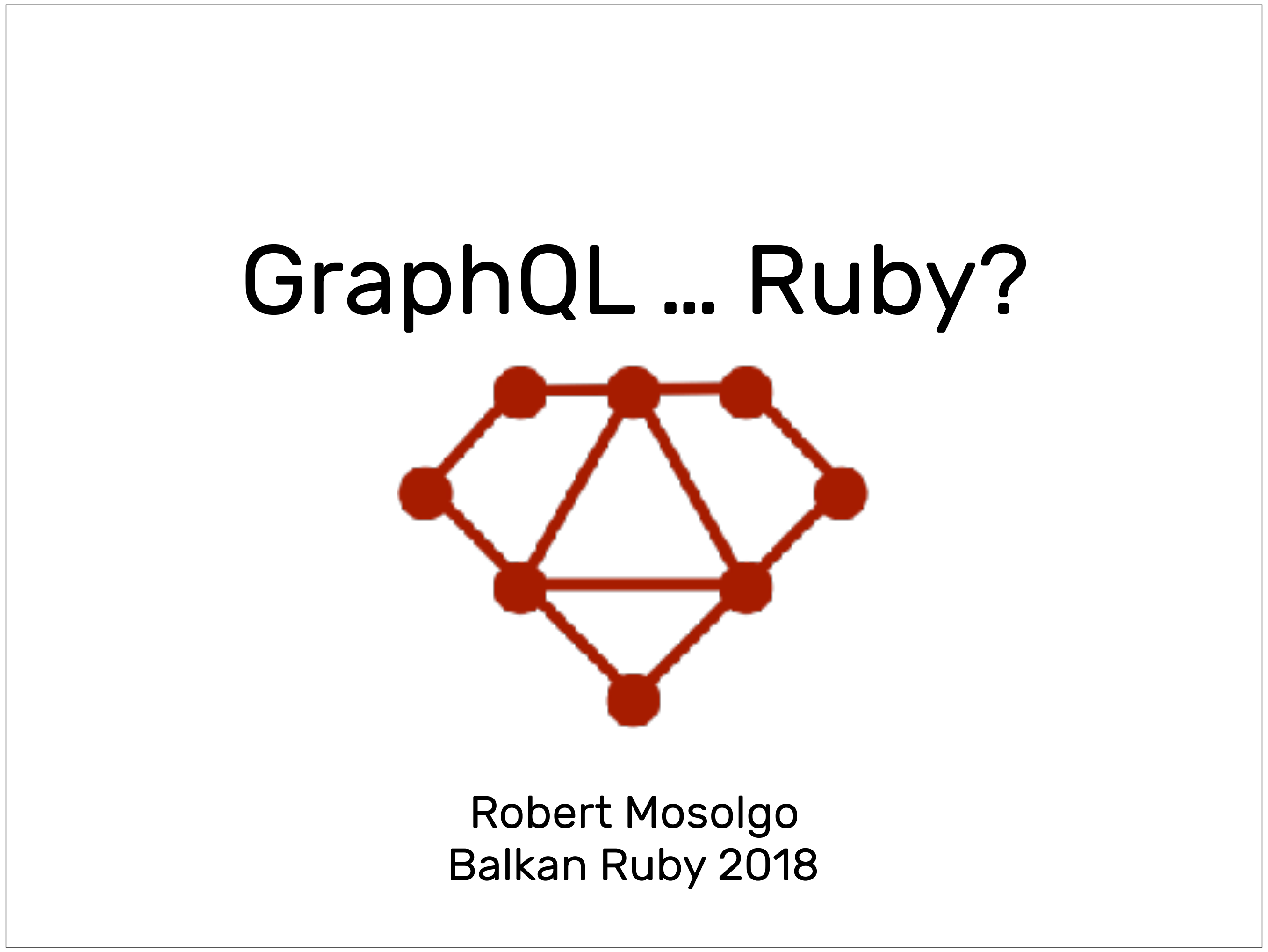


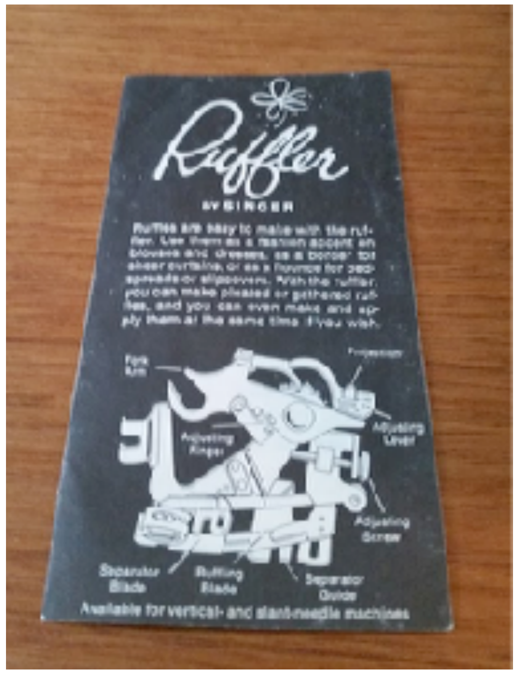
GraphQL … Ruby?

Robert Mosolgo
Balkan Ruby 2018

I’m Robert Mosolgo, to introduce myself I’m going to tell you about my new hobby.

Sewing. Here's the machine I'm working with now, it's a classic, more about that later.

Here are some of my projects lately, the window treatment, the bench, those curtains.
You have to pattern-match the fabric, and it’s hard to keep it lined up as you go.



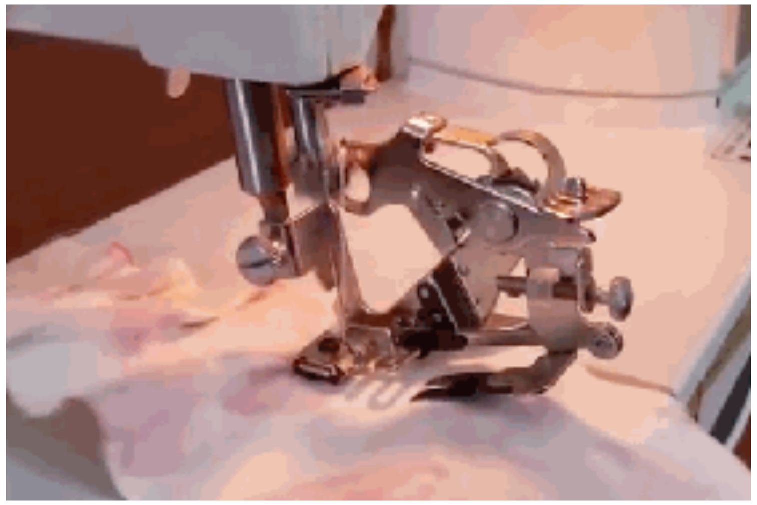
Ruffler

BY SINGER

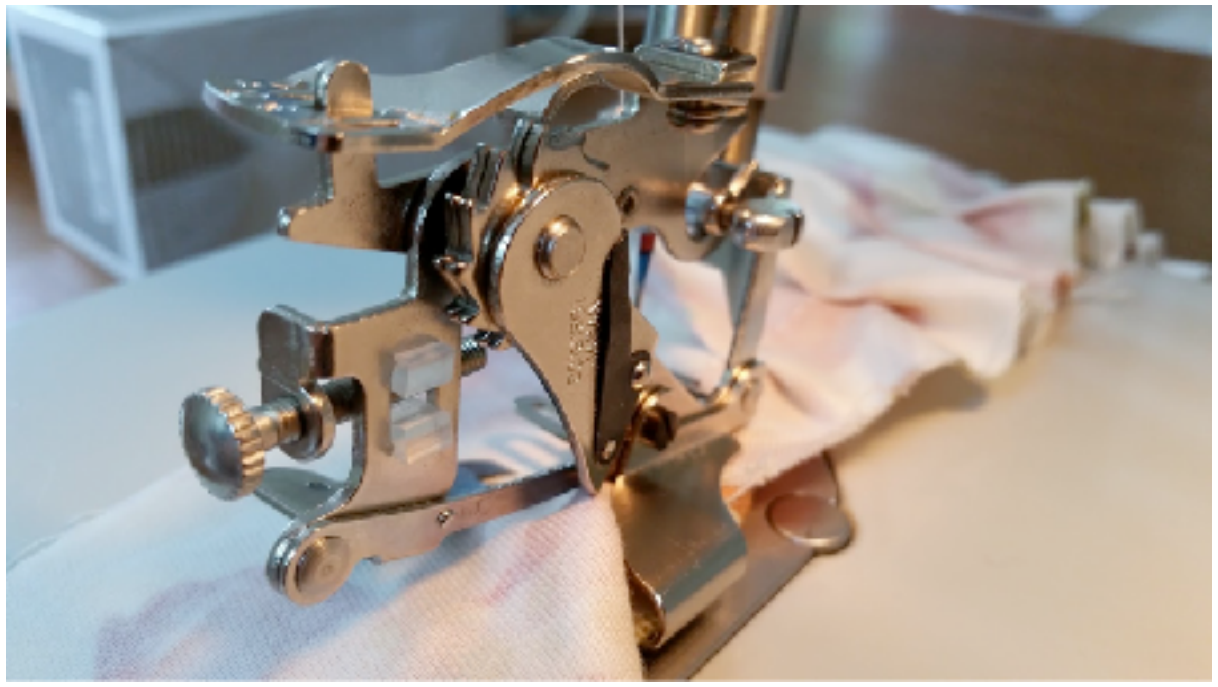
Ruffles are easy to make with the ruffler. Use them as a fashion accent on blouses and dresses, as a border for sheer curtains, or as a flounce for bedspreads or slipcovers. With the ruffler, you can make pleated or gathered ruffles, and you can even make and apply them at the same time if you wish.

Fork Arm
Projection
Adjusting Finger
Adjusting Lever
Adjusting Screw
Separator Blade
Ruffling Blade
Separator Guide

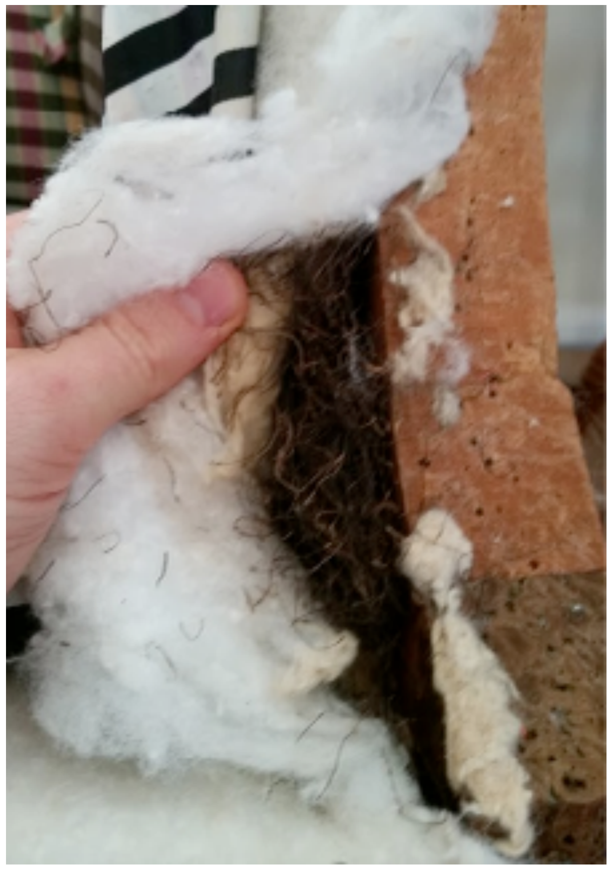
Available for vertical- and slant-needle machines



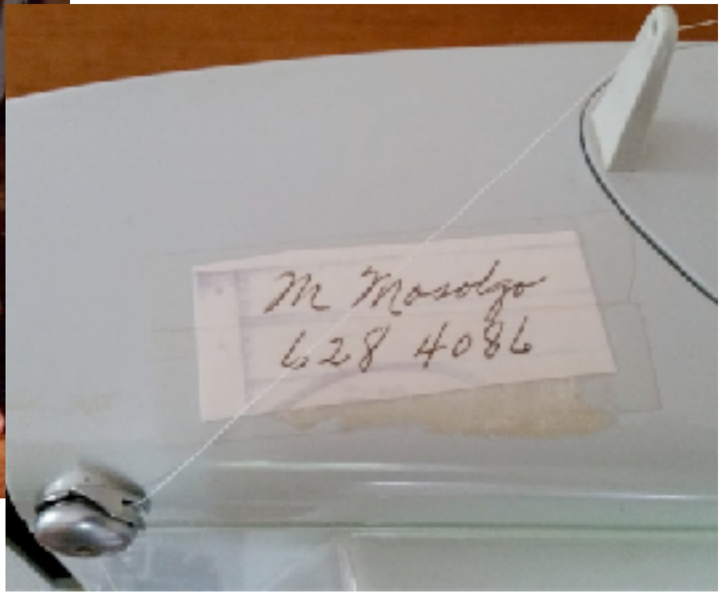
I’m really enjoying the mechanical aspect of it. Here’s the Ruffler — what a device! Also, this technology is hands-on, it’s easy to see how great the machine is, how long it would have taken by hand.



Here’s another view of the Ruffler. Each stitch clicks that wheel, and it increments. Then, on the deeper cuts, it drives the ruffling blade to tuck in that fabric. Amazing.

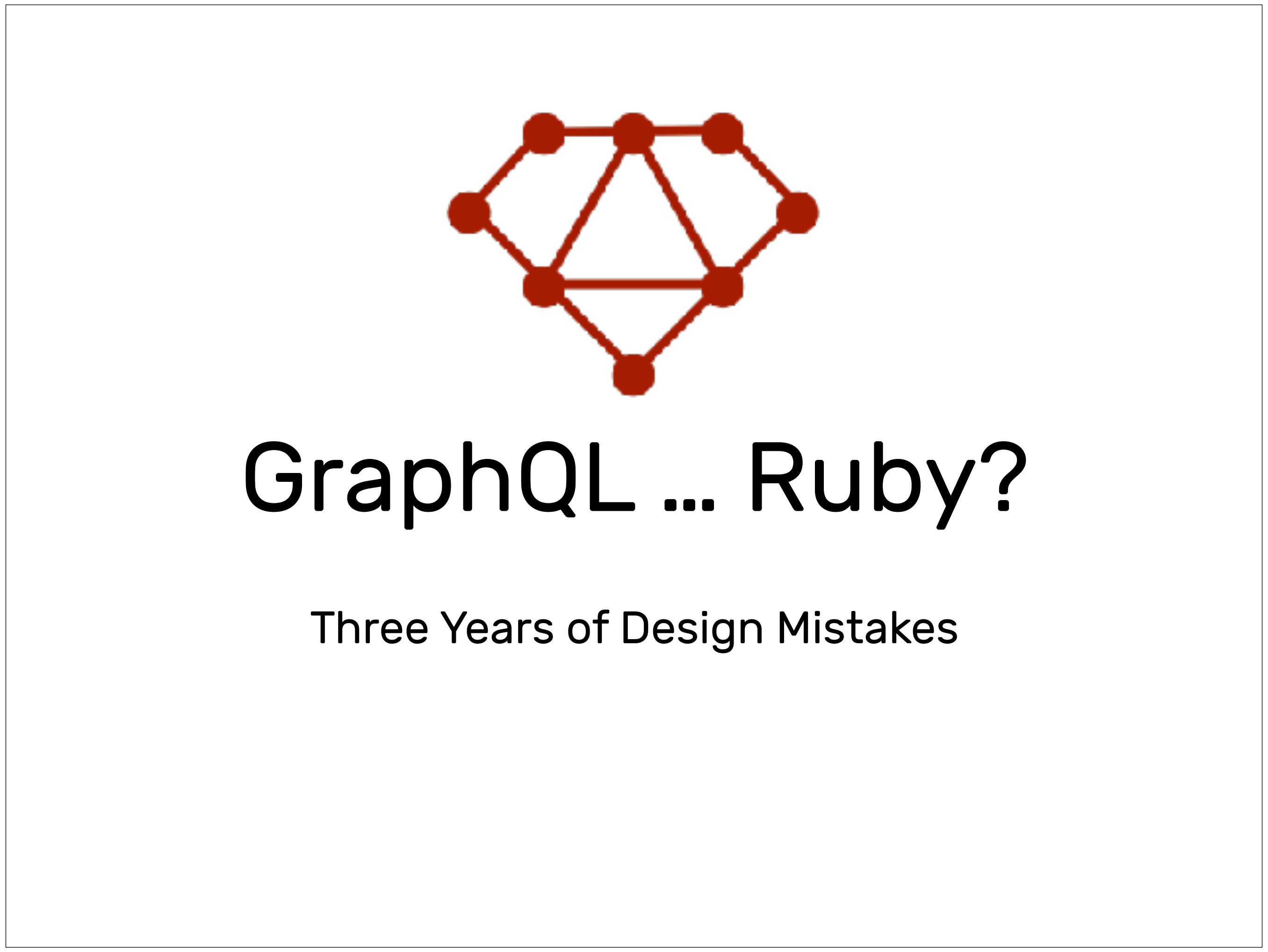


There’s also a cool historical aspect to this. Here’s a chair that I thought I’d reupholster. It’s on hold at the moment. But stripping it down, look what I found: seriously retro padding, that’s HAIR! Apparently horse hair or pig hair was used because it maintained its springiness. Wool and cotton are put on top because it’s softer. And newer upholstery is visible there too — polyester batting.



The history is also personal. Here’s my Mamaw, my grandmother. It’s her sewing machine, but sadly she’s dead now, so I can’t learn to sew from her, instead I’m watching youtube videos. That’s a shame.

Enough about sewing, let’s talk about GraphQL!



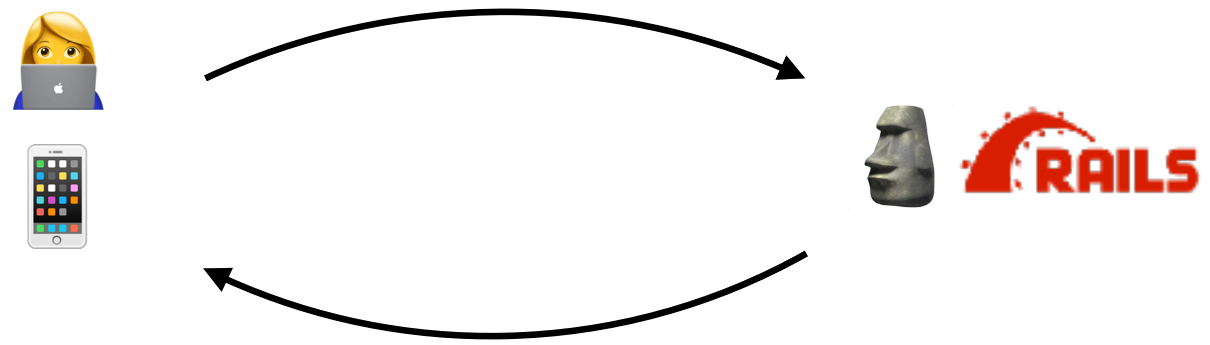
GraphQL … Ruby?

Three Years of Design Mistakes

So, in this talk, I'm going to introduce GraphQL, then talk about my experience implementing it, building the library, and some lessons I learned along the way.

What is GraphQL?

```
POST /graphql
  { ... }
```



```
{ “data”: ... }
```

GraphQL is for client-server communication, and it works like this …

Two parts: the language, for the incoming documents, and the runtime, for how it’s fulfilled. Let’s look more closely.

Query Language

```
{
  users(first: 5) {
    login
    repositories(first: 5) {
        name
    }
  }
}
```

So, let’s talk about the query language. Here’s a GraphQL Query, as a reminder, it usually comes in an HTTP Post request.

Query Language

```
{ # selection
  users(first: 5) {
    login
    repositories(first: 5) {
        name
    }
  }
}
```

The first part of the query is a *selection*. A bit like SELECT in SQL, it says we want to fetch some data.

Query Language

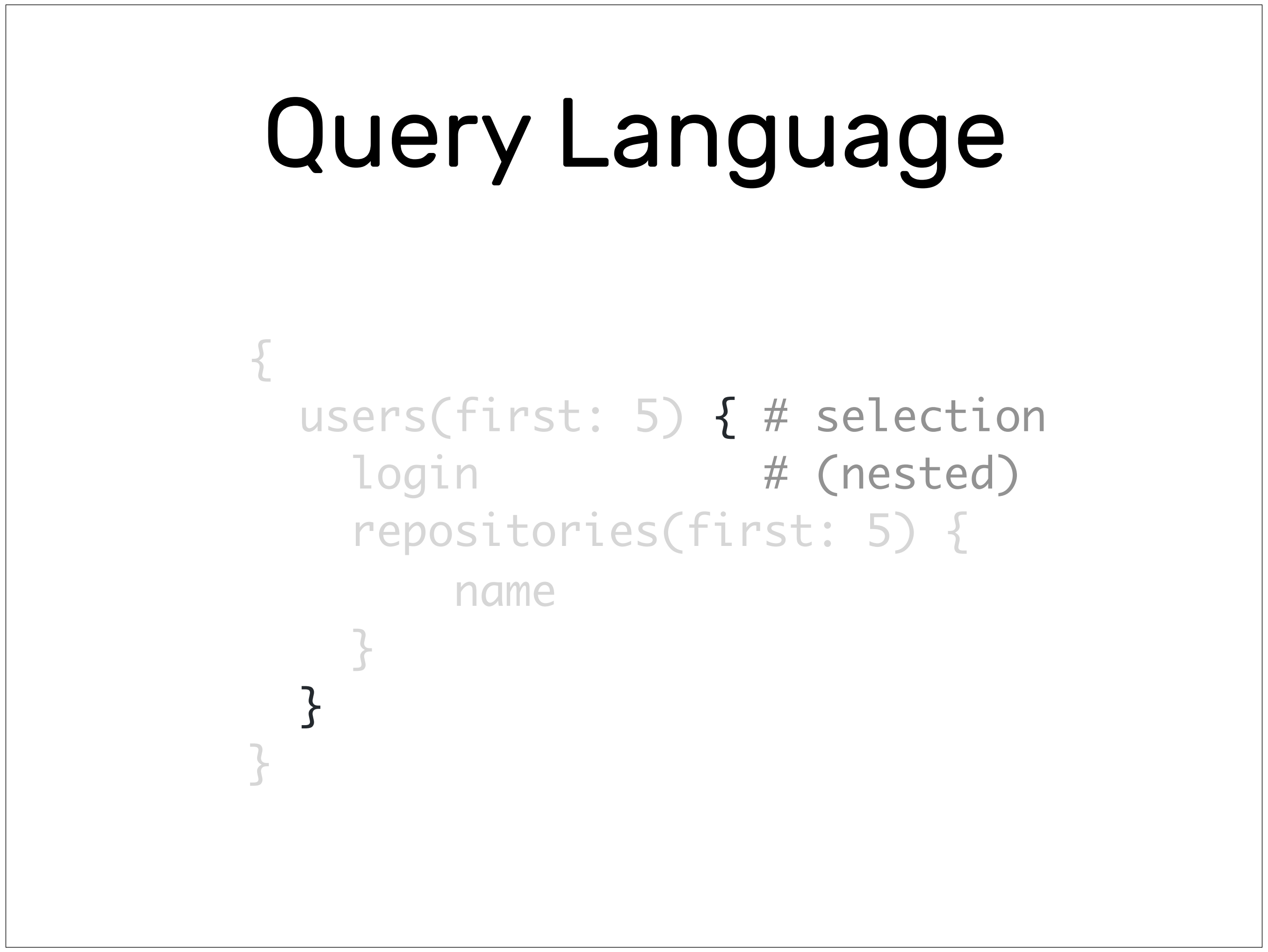
```
{
  users(first: 5) { # field
    login
    repositories(first: 5) {
        name
    }
  }
}
```

Next is a *field*. It names the data that we actually want, a bit like telling a database to select a column.

Query Language

```
{
  users(first: 5) { # arguments
    login
    repositories(first: 5) {
        name
    }
  }
}
```

Fields can have arguments that help the server get data. This is like SQL WHERE, LIMIT, and ORDER, all together.



Query Language

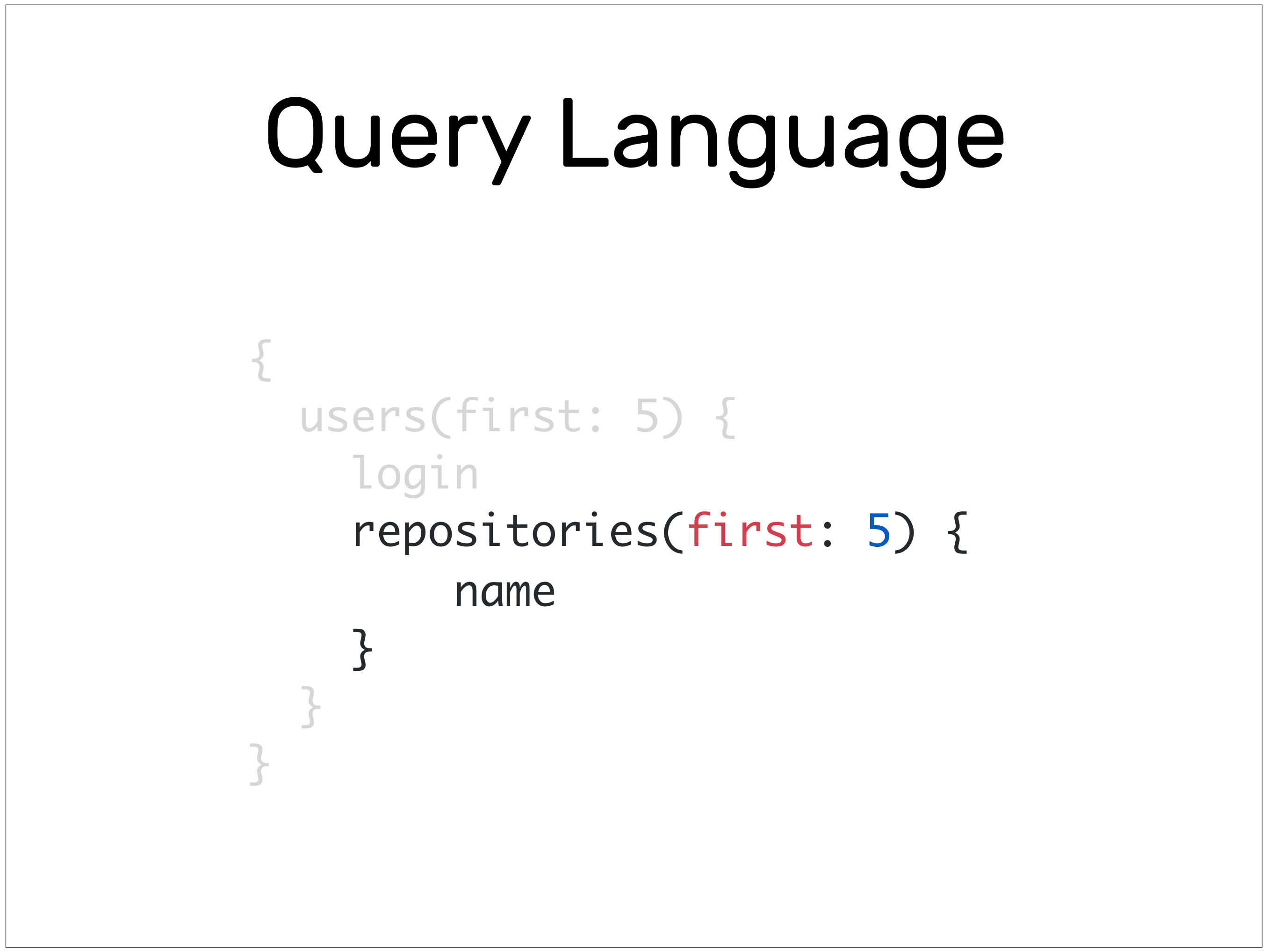
```
{
  users(first: 5) { # selection
    login           # (nested)
    repositories(first: 5) {
        name
    }
  }
}
```

Now, since the users field will return a list of users, we can request some data for each user in the list. We express this with another selection, a *nested* selection.

Query Language

```
{
  users(first: 5) {
    login # field (scalar)
    repositories(first: 5) {
        name
    }
  }
}
```

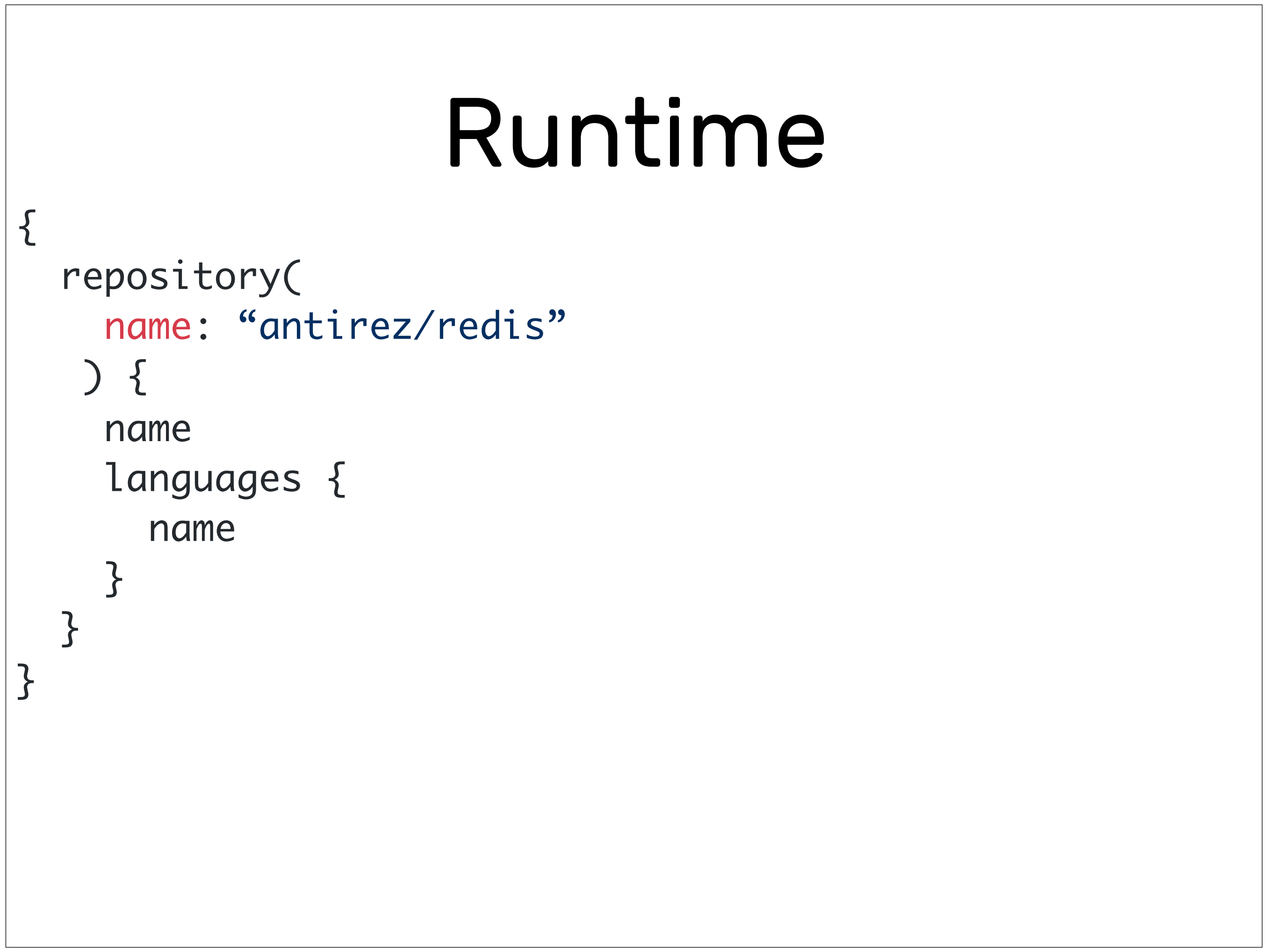
Inside the selection, we can request data about a User, for example, their login. This field returns a String, so there's nothing else to select. We'll get a string here.



Query Language

```
{
  users(first: 5) {
    login
    repositories(first: 5) {
        name
    }
  }
}
```

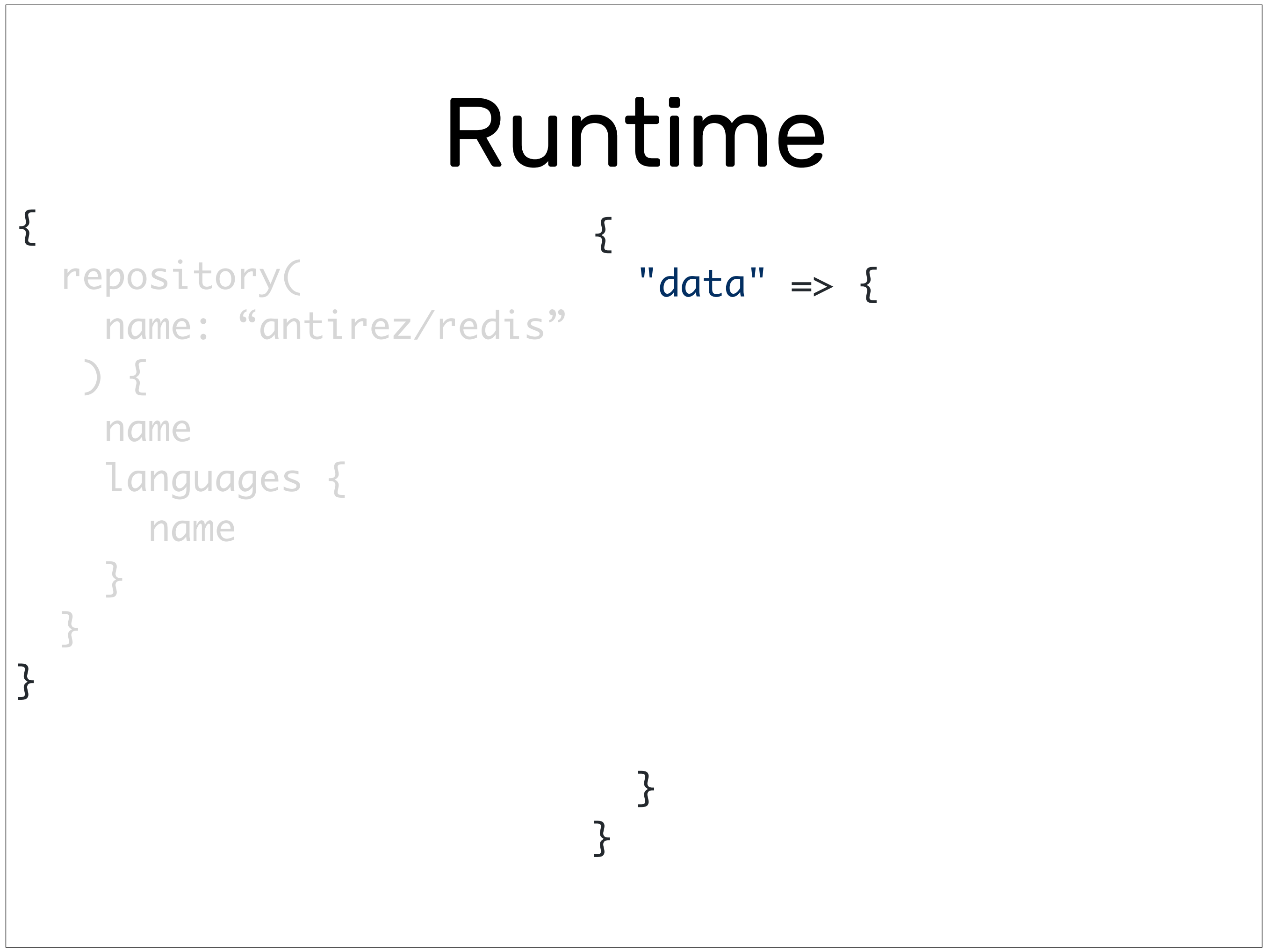
And again, we request repositories for each user, and the name of each repository. That’s the simplest example of GraphQL, like, “Hello World.”



Runtime

```
{
  repository(
    name: “antirez/redis”
   ) {
    name
    languages {
      name
    }
  }
}
```

So, given a query, how does GraphQL create a response? That’s called the *runtime*.

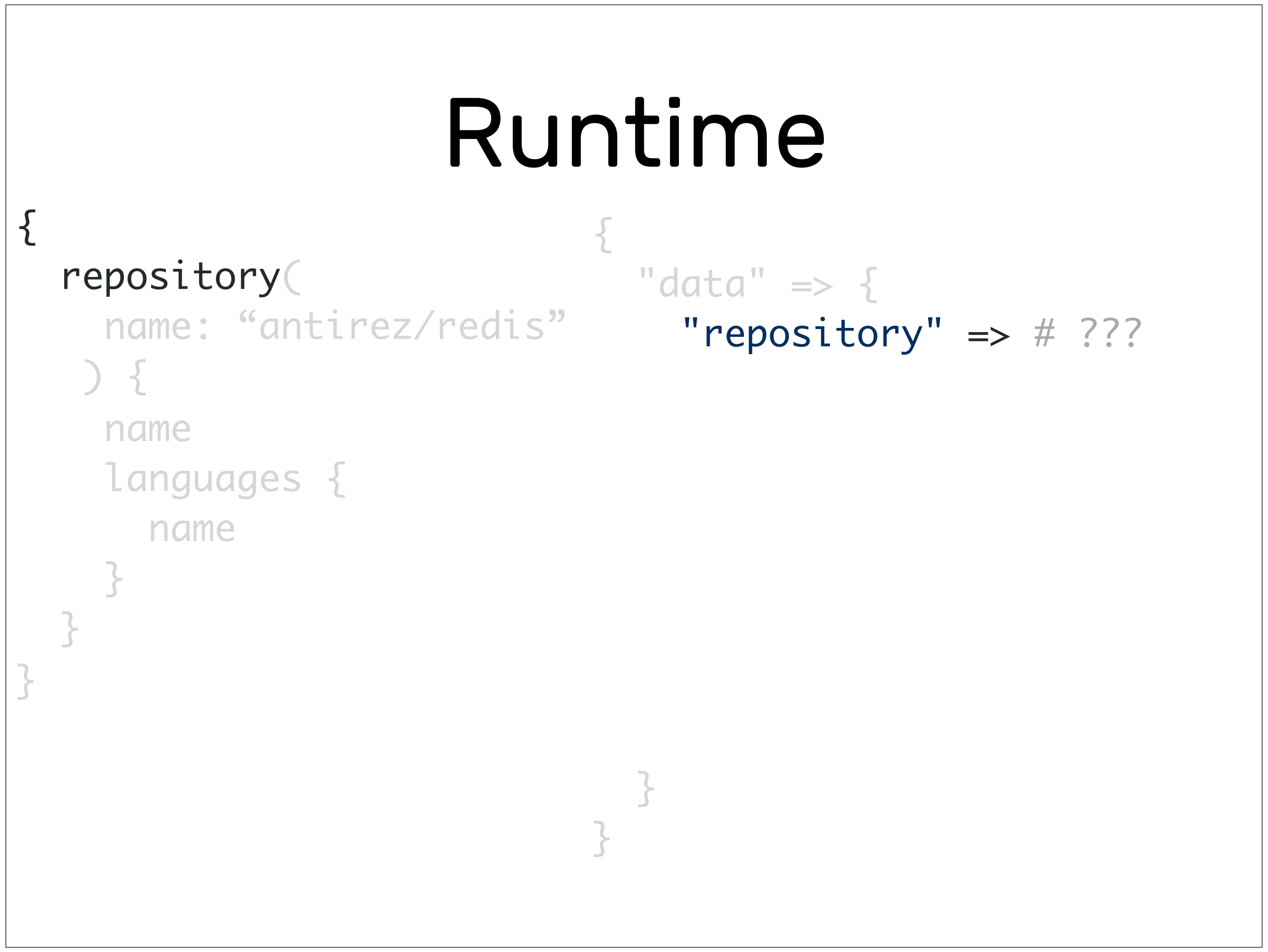


Runtime

```
{
  repository(
    name: “antirez/redis”
   ) {
    name
    languages {
      name
    }
  }
}
```

```
{
  "data" => {



  }
}
```

First, every query puts its response in the “data” key of the JSON response.

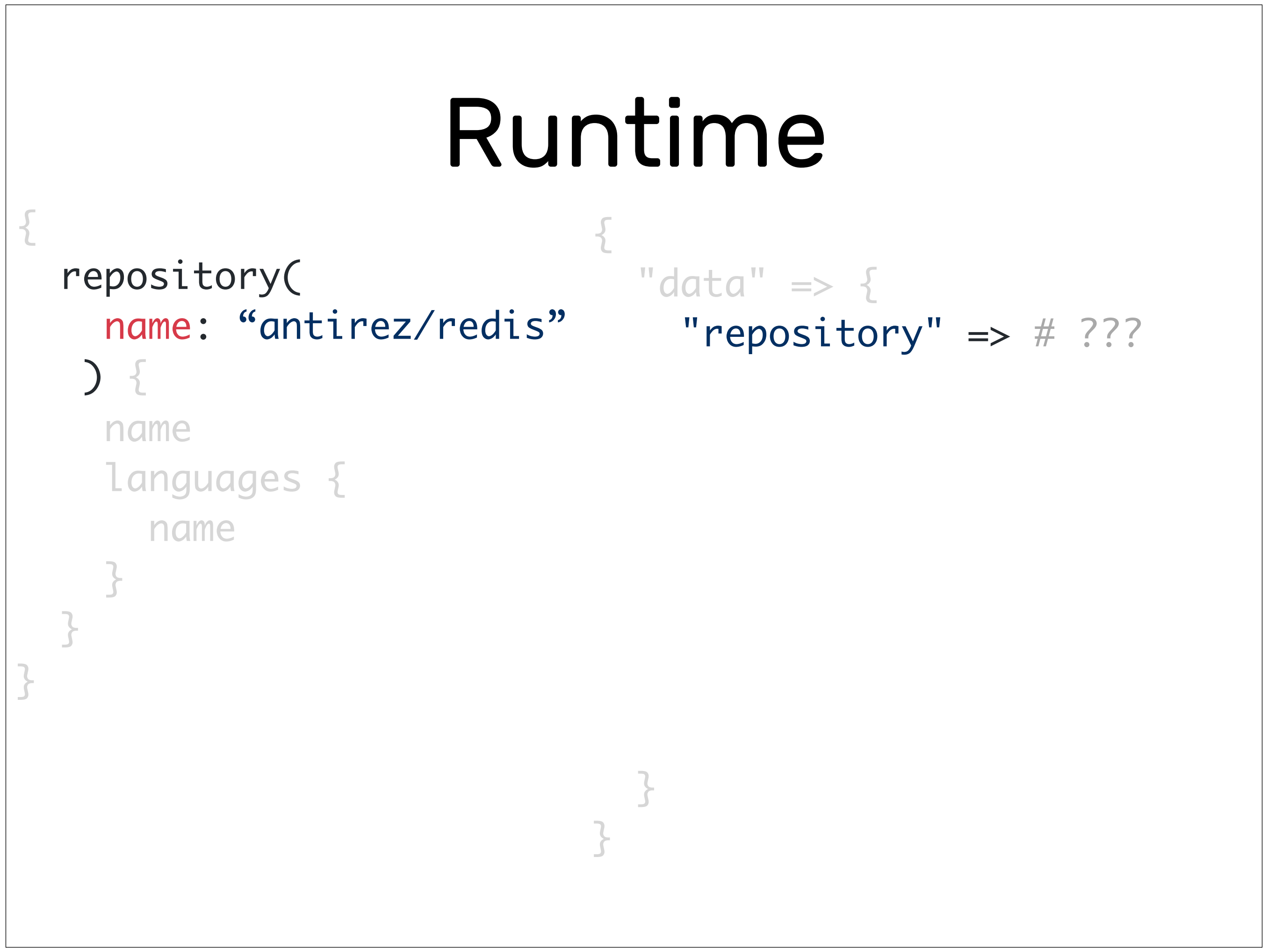


Runtime

```
{
  repository(
    name: “antirez/redis”
   ) {
    name
    languages {
      name
    }
  }
}
```

```
{
  "data" => {
    "repository" => # ???
  }
}
```

Next, the first field is executed. Since the field was called repository, there’s a matching field in the JSON response. But what goes there?

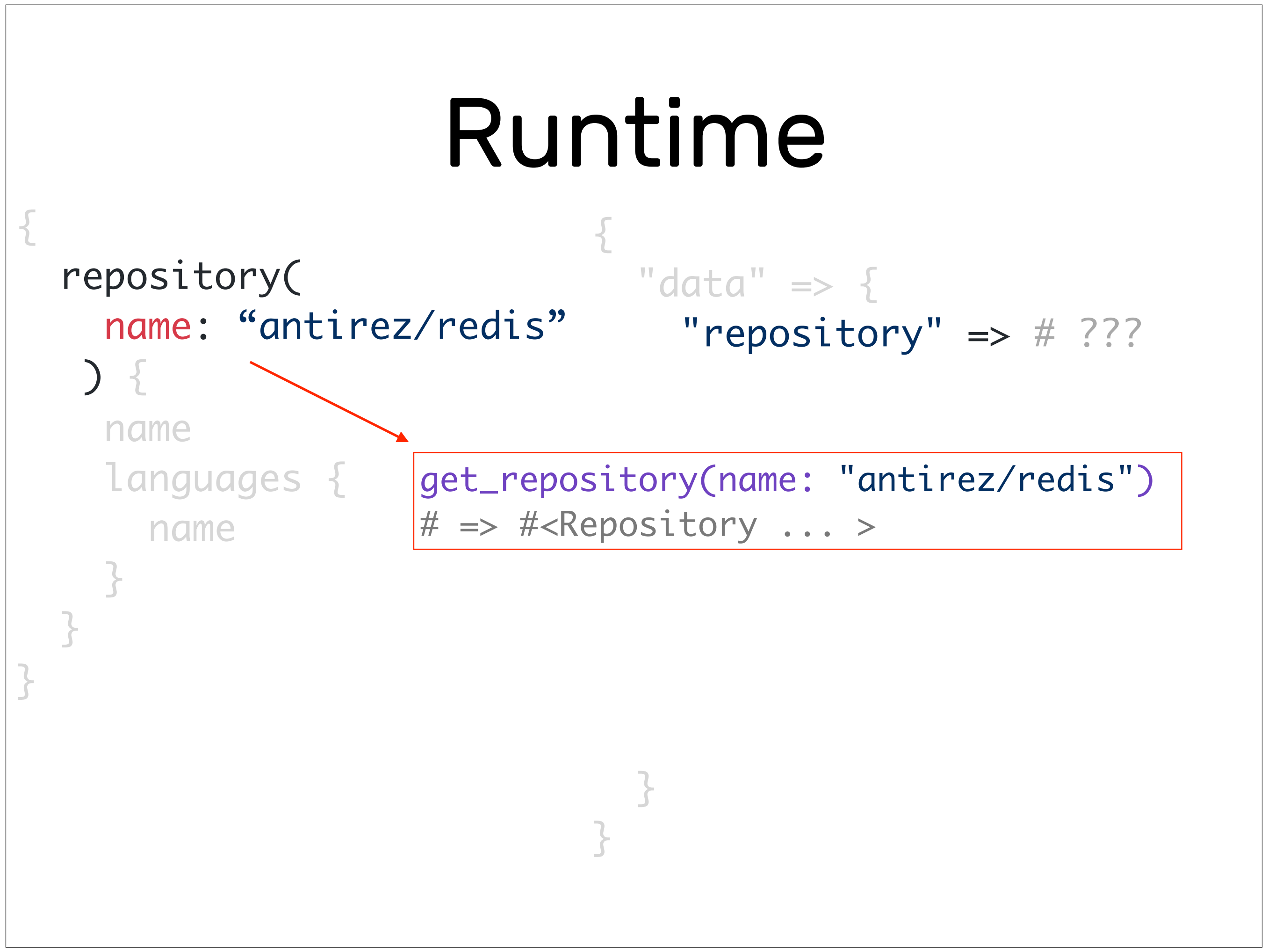


Runtime

```
{
  repository(
    name: “antirez/redis”
   ) {
    name
    languages {
      name
    }
  }
}
```

```
{
  "data" => {
    "repository" => # ???
  }
}
```

This is where the magic happens, in GraphQL, it’s called a “resolve function” or “resolver”.



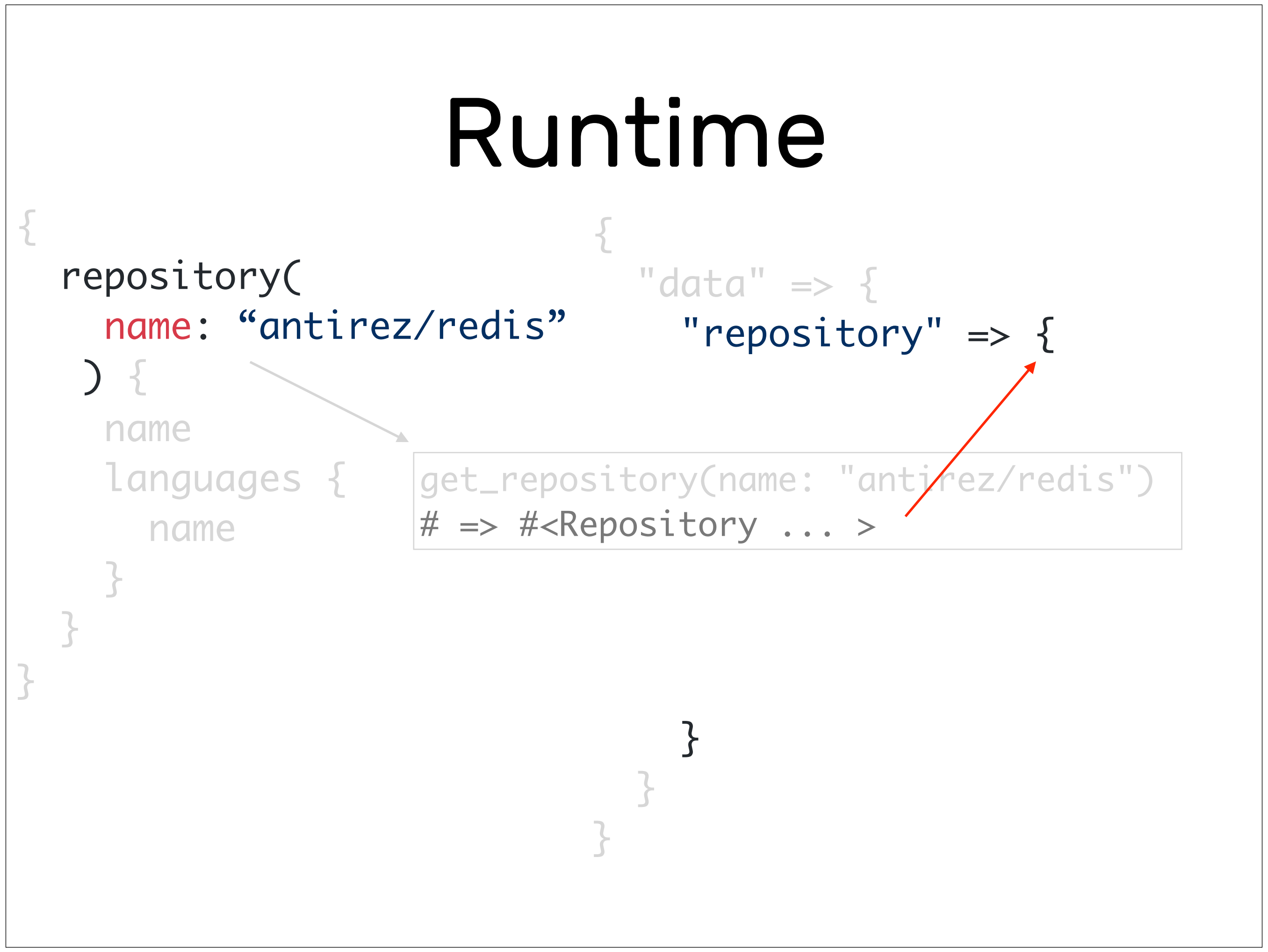
Runtime

```
{
  repository(
    name: “antirez/redis”
   ) {
    name
    languages {
      name
    }
  }
}
```

```
{
  "data" => {
    "repository" => # ???



  }
}
```

```
get_repository(name: "antirez/redis")
# => #<Repository ... >
```

We have some Ruby code that corresponds to the field, so we call it.



Runtime

```
{
  repository(
    name: “antirez/redis”
  ) {
    name
    languages {
      name
    }
  }
}
```

```
get_repository(name: "antirez/redis")
# => #<Repository ... >
```

```
{
  "data" => {
    "repository" => {

    }
  }
}
```

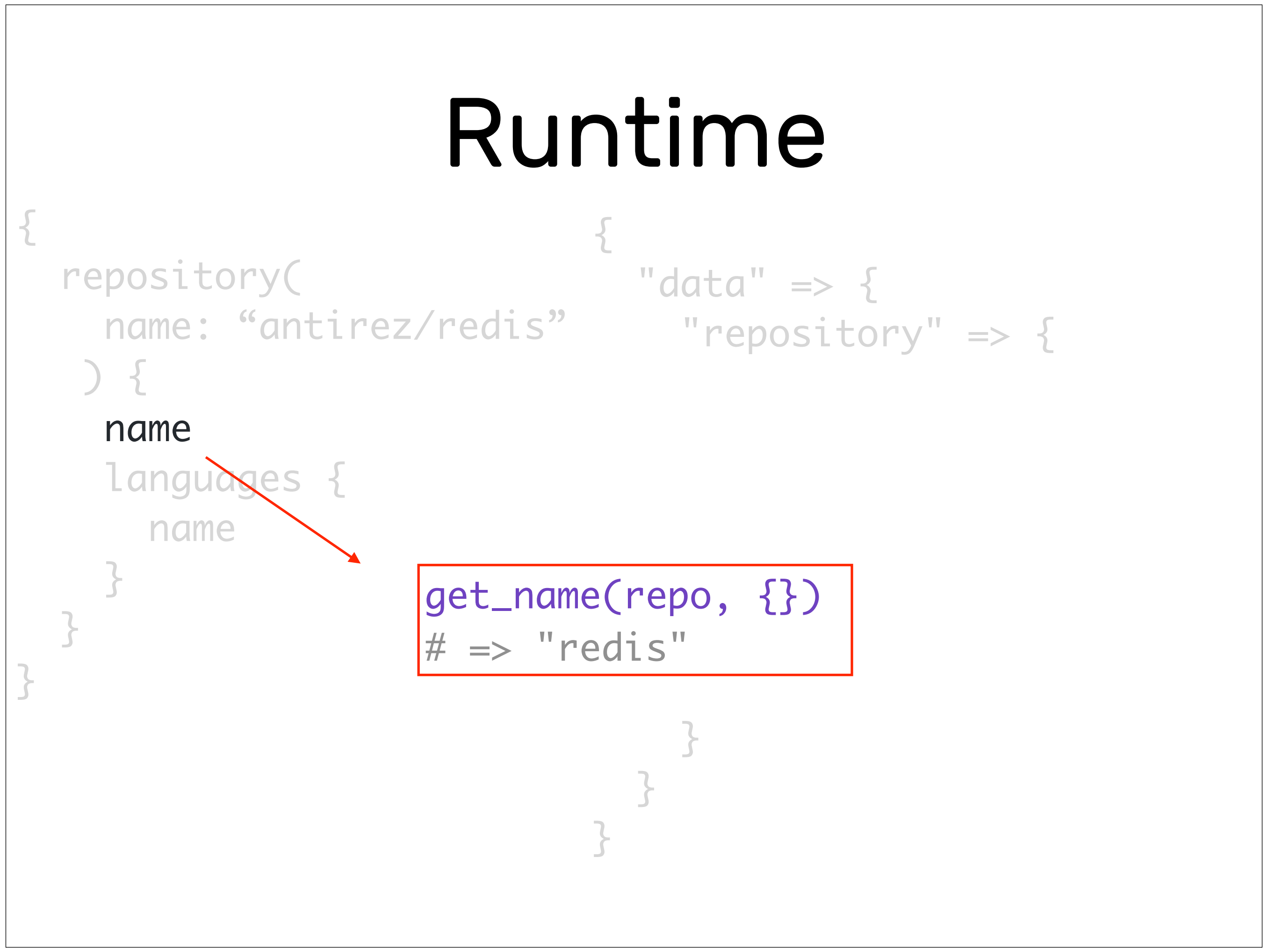
Since we got an object, a Repository, we’re going to add an object to the response. (Otherwise, we would put `nil` in the response).

Runtime

```
{
  repository(
    name: “antirez/redis”
   ) {
    name
    languages {
      name
    }
  }
}
```

```
{
  "data" => {
    "repository" => {




    }
  }
}
```

Now, we keep resolving fields in the same way



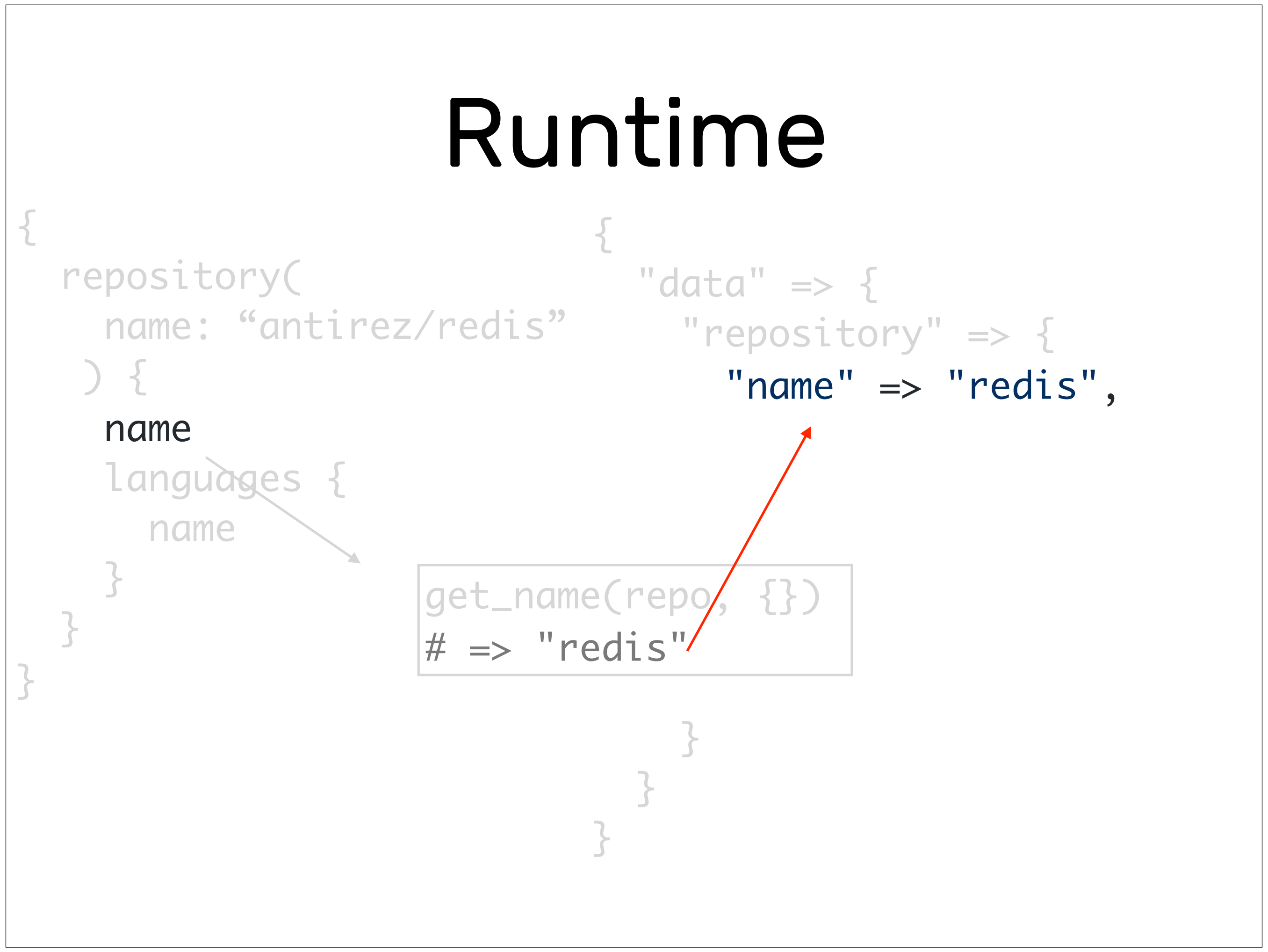
Runtime

```
{
  repository(
    name: “antirez/redis”
   ) {
    name
    languages {
      name
    }
  }
}
```

```
{
  "data" => {
    "repository" => {
      }
    }
}
```

```
get_name(repo, {})
# => "redis"
```

Now, we keep resolving fields in the same way



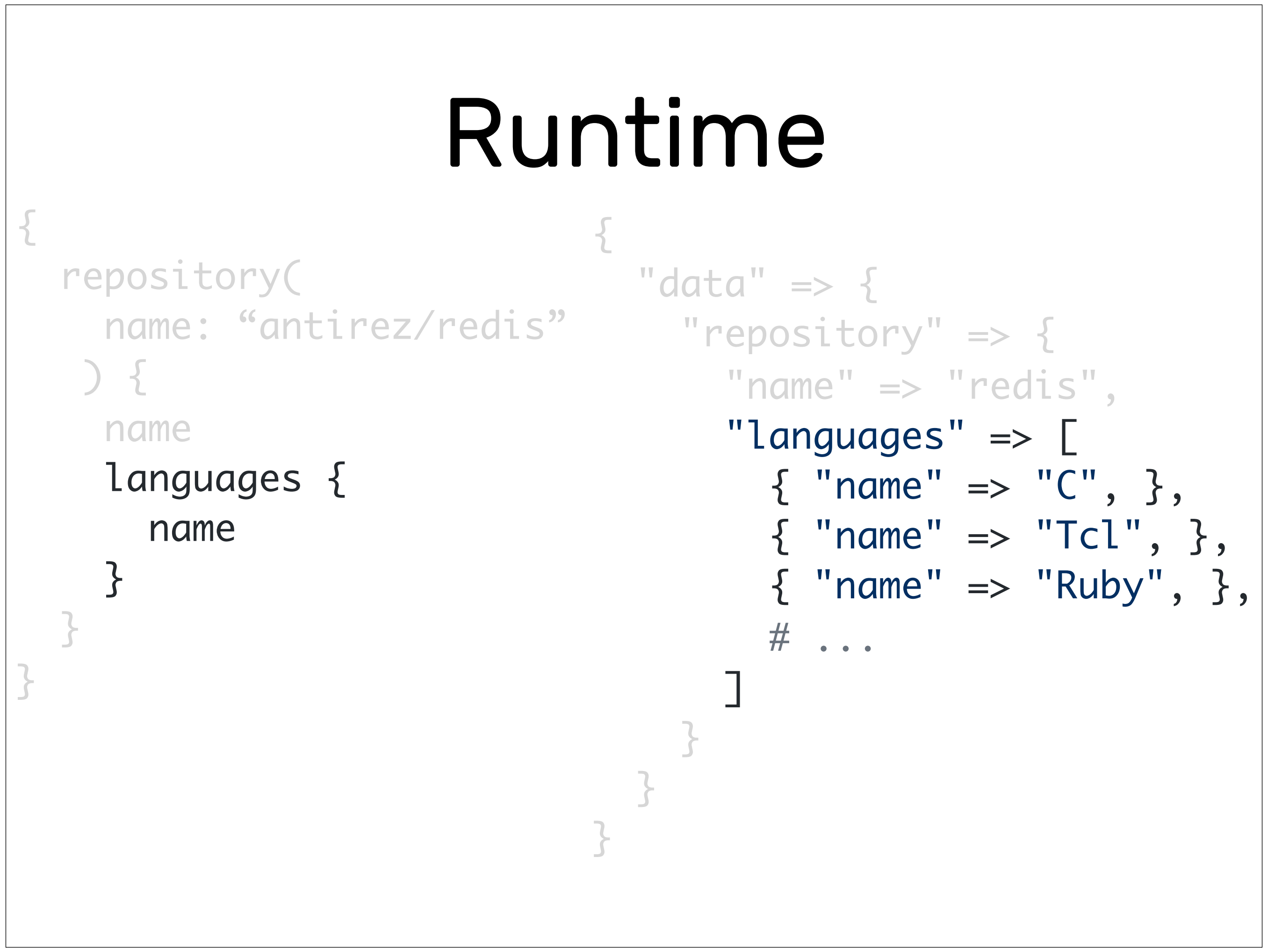
Runtime

```
{
  repository(
    name: “antirez/redis”
   ) {
    name
    languages {
      name
    }
  }
}
```

```
get_name(repo, {})
# => "redis"
```

```
{
  "data" => {
    "repository" => {
      "name" => "redis",


    }
  }
}
```

Now, we keep resolving fields in the same way

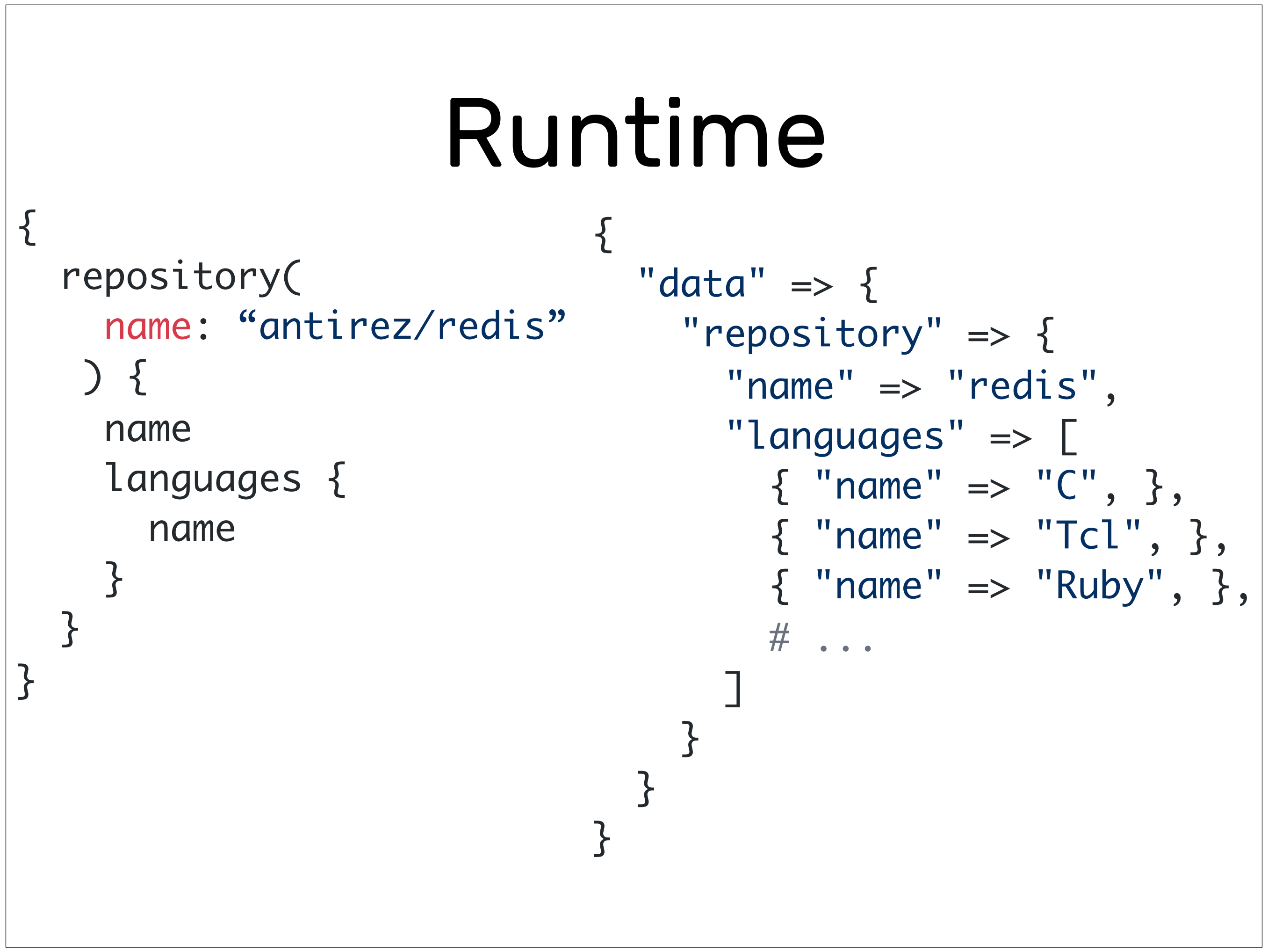


Runtime

```
{
  repository(
    name: “antirez/redis”
   ) {
    name
    languages {
      name
    }
  }
}
```

```
{
  "data" => {
    "repository" => {
      "name" => "redis",
      "languages" => [
        { "name" => "C", },
        { "name" => "Tcl", },
        { "name" => "Ruby", },
        # ...
      ]
    }
  }
}
```

And so on, each field is resolved.

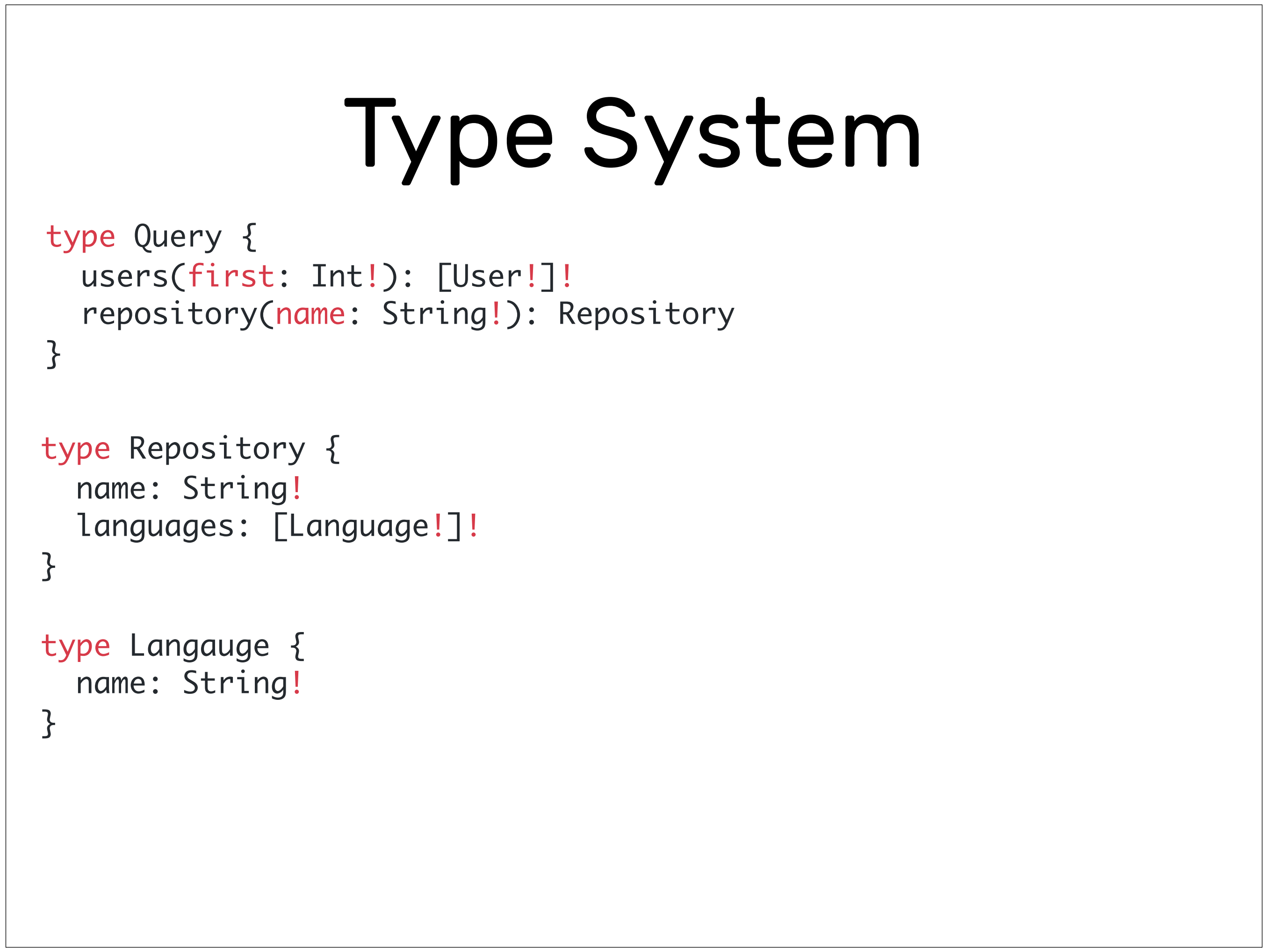


Runtime

```
{
  repository(
    name: “antirez/redis”
   ) {
    name
    languages {
      name
    }
  }
}
```

```
{
  "data" => {
    "repository" => {
      "name" => "redis",
      "languages" => [
        { "name" => "C", },
        { "name" => "Tcl", },
        { "name" => "Ruby", },
        # ...
      ]
    }
  }
}
```

That’s how we get the response — and how it matches the request!

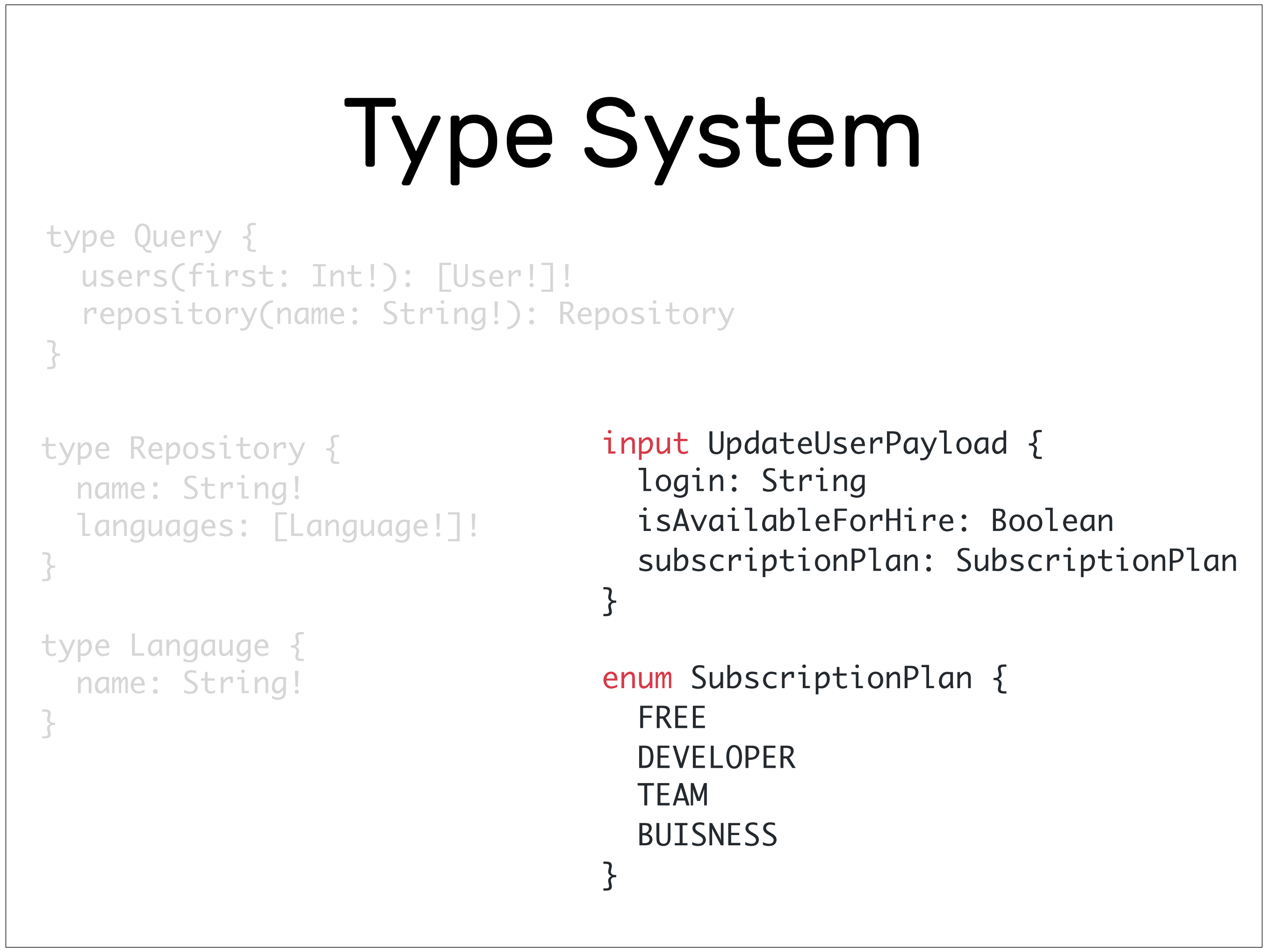


Type System

```
type Query {
  users(first: Int!): [User!]!
  repository(name: String!): Repository
}

type Repository {
  name: String!
  languages: [Language!]!
}

type Langauge {
  name: String!
}
```

How does the runtime know what to do? You build a schema for your app. This is the basis of execution, and it's also used for validating incoming queries and type-checking client inputs. This type system is defined using Ruby, which we'll talk about next.

Each type is defined, and there's one called the *root* type which is the entry point for your schema. By convention, it's called "Query", and it's the starting point for all queries.

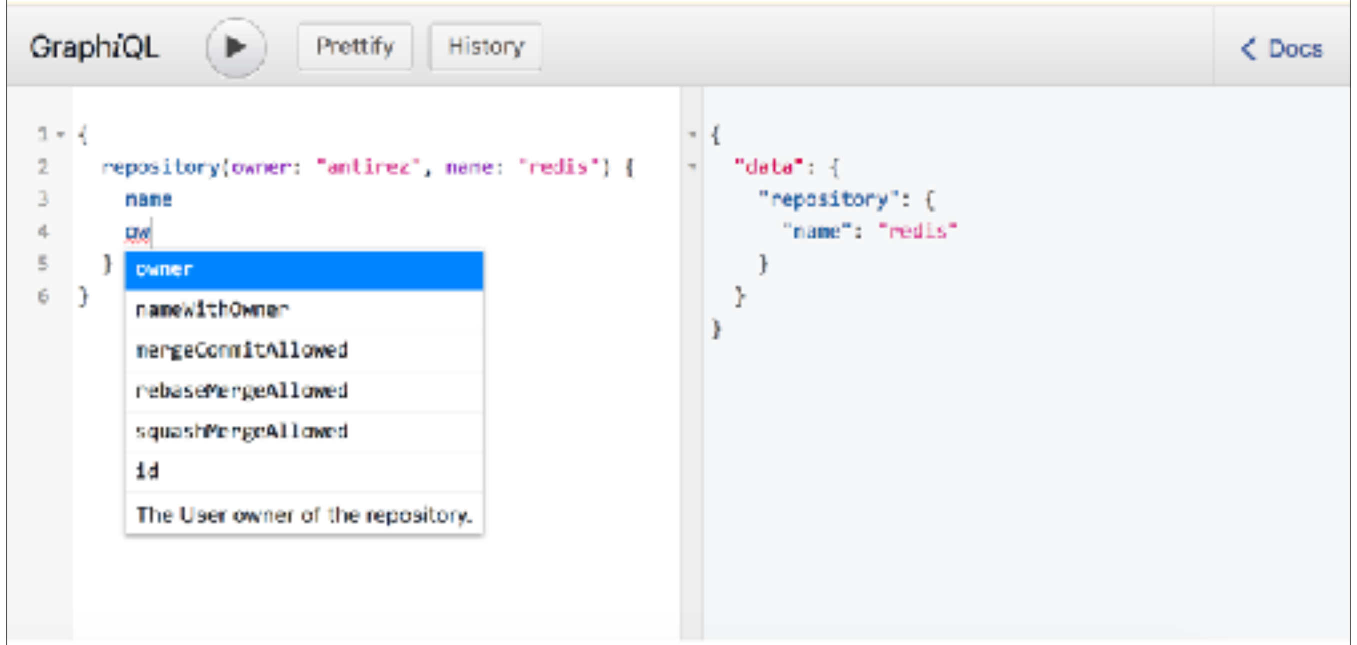


Type System

```
type Query {
  users(first: Int!): [User!]!
  repository(name: String!): Repository
}

type Repository {
  name: String!
  languages: [Language!]!
}

type Langauge {
  name: String!
}
```

```
input UpdateUserPayload {
  login: String
  isAvailableForHire: Boolean
  subscriptionPlan: SubscriptionPlan
}

enum SubscriptionPlan {
  FREE
  DEVELOPER
  TEAM
  BUISNESS
}
```

Inputs are also strongly typed — this means for mutation requests like create and update, all the inputs are validated before GraphQL executes the request.



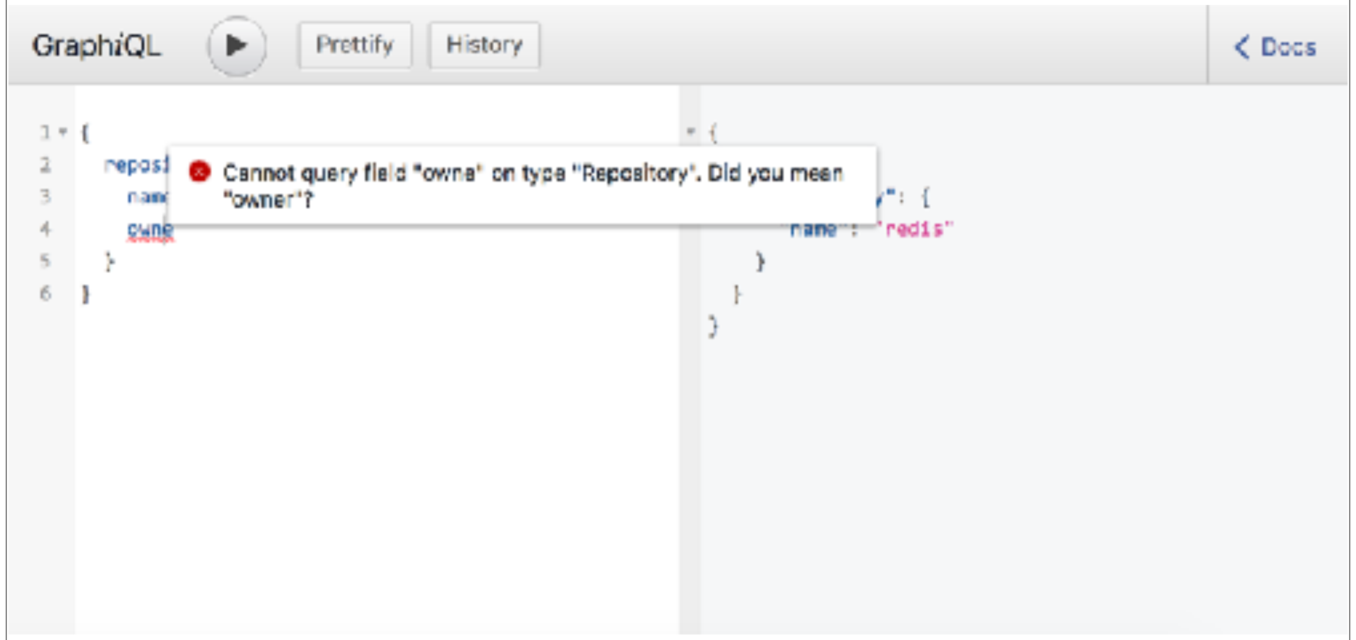
```
{
  repository(owner: "antirez", name: "redis") {
    name
    ow
  }
}
```

```
owner
nameWithOwner
mergeCommitAllowed
rebaseMergeAllowed
squashMergeAllowed
id
```

The User owner of the repository.

```
{
  "data": {
    "repository": {
      "name": "redis"
    }
  }
}
```

This type system can be leveraged to support really nice tooling.

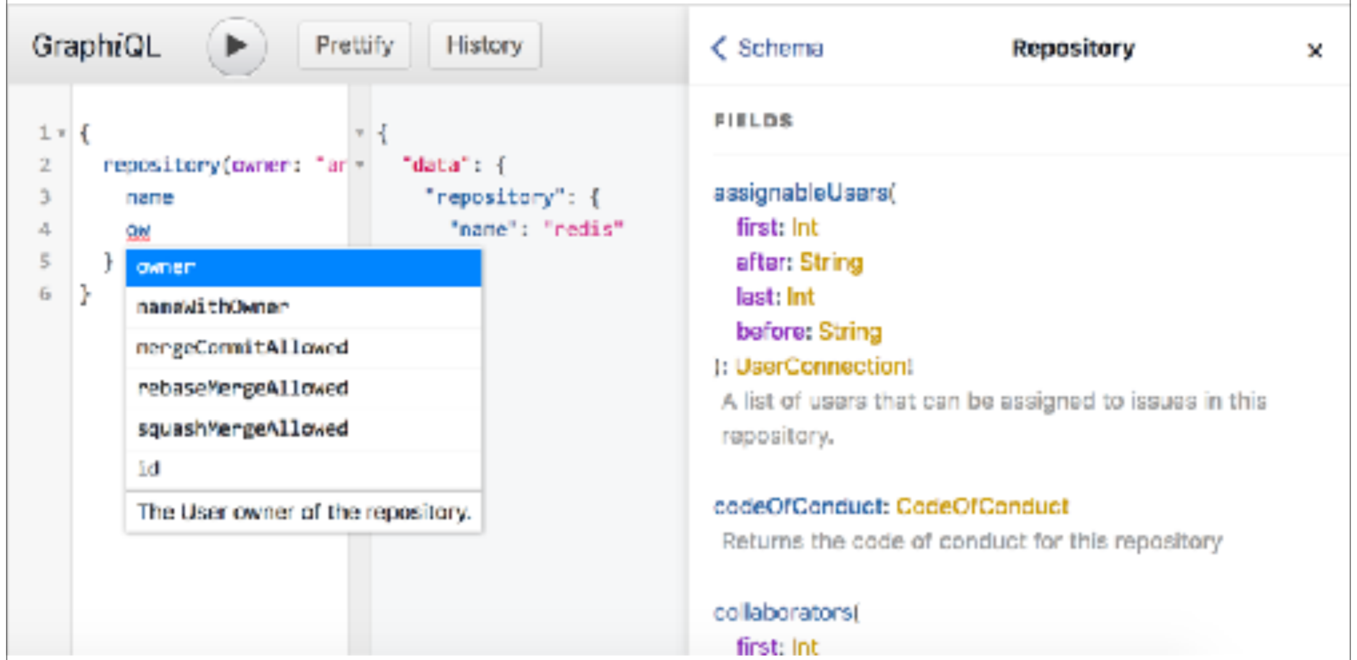


```
{
  reposi
    nam
    owne
  }
}
```

Cannot query field "owna" on type "Repository". Did you mean "owner"?

```
{
        y": {
      "name": "redis"
    }
  }
}
```

And warns you if there are errors!



```
{
  repository(owner: "ar
    name
    ow
  }
}
```

```
owner
nameWithOwner
mergeCommitAllowed
rebaseMergeAllowed
squashMergeAllowed
id
```

The User owner of the repository.

```
{
  "data": {
    "repository": {
      "name": "redis"
```

FIELDS

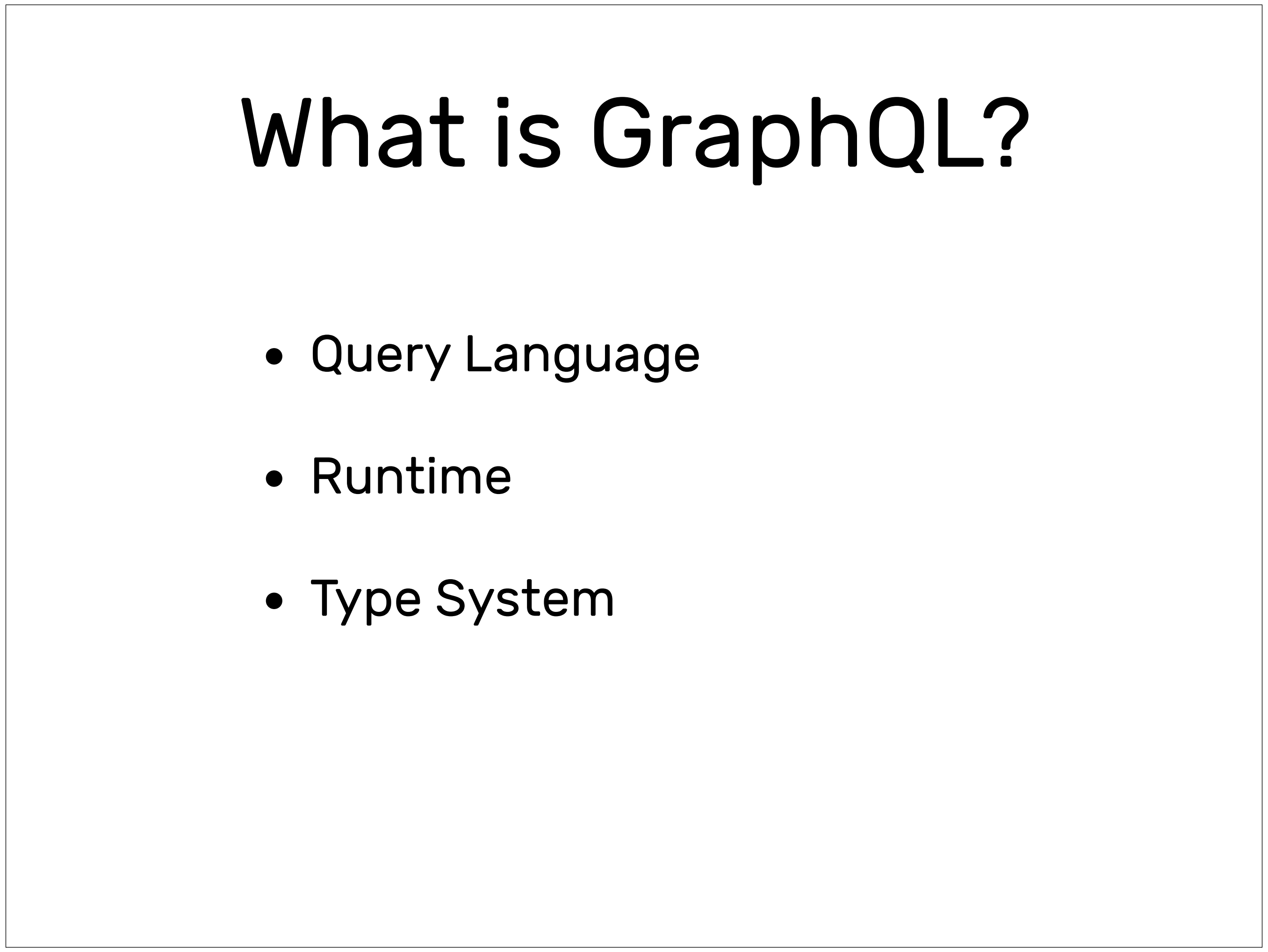
assignableUsers(
  first: Int
  after: String
  last: Int
  before: String
): UserConnection!
A list of users that can be assigned to issues in this repository.

codeOfConduct: CodeOfConduct
Returns the code of conduct for this repository

collaborators(
  first: Int

https://developer.github.com/v4

Also, it has built-in documentation.

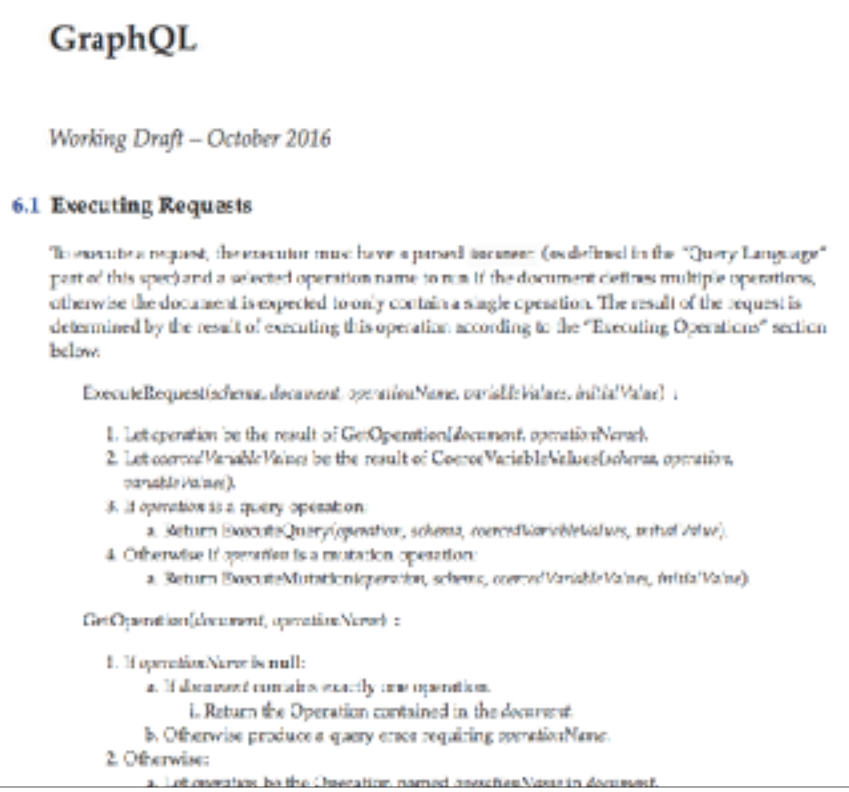


What is GraphQL?

- Query Language
- Runtime
- Type System

So, to summarize, we talked about how GraphQL includes a query language, which describes a request for data. And we talked about the Runtime, which is how GraphQL requests are fulfilled, by calling Ruby code in your app. Finally, we talked about the type system, that gives structure to your API and supports tooling.

The Spec



GraphQL

Working Draft – October 2016

6.1 Executing Requests

To execute a request, the executor must have a parsed document (as defined in the "Query Language" part of this spec) and a selected operation name to run if the document defines multiple operations, otherwise the document is expected to only contain a single operation. The result of the request is determined by the result of executing this operation according to the "Executing Operations" section below.

```
ExecuteRequest(schema, document, operationName, variableValues, initialValue) :

    1. Let operation be the result of GetOperation(document, operationName).
    2. Let coercedVariableValues be the result of CoerceVariableValues(schema, operation,
       variableValues).
    3. If operation is a query operation:
        a. Return ExecuteQuery(operation, schema, coercedVariableValues, initialValue).
    4. Otherwise if operation is a mutation operation:
        a. Return ExecuteMutation(operation, schema, coercedVariableValues, initialValue).

GetOperation(document, operationName) :

    1. If operationName is null:
        a. If document contains exactly one operation.
            i. Return the Operation contained in the document.
        b. Otherwise produce a query error requiring operationName.
    2. Otherwise:
        a. Let operation be the Operation named operationName in document.
```

I've skipped some features in the language and runtime, but the best part is, you can find it all in the GraphQL specification. GraphQL-Ruby implements the specification, and there are implementations in lots of languages. By the way, the spec was created by folks at Facebook, that's where GraphQL was invented and they continue to steward the design of it.

Rails Conf 2018 PITTSBURGH

Agenda

- What is GraphQL
- Using in your app
- Potential pitfalls

Nick Quaranto, “The GraphQL Way”
RailsConf 2018

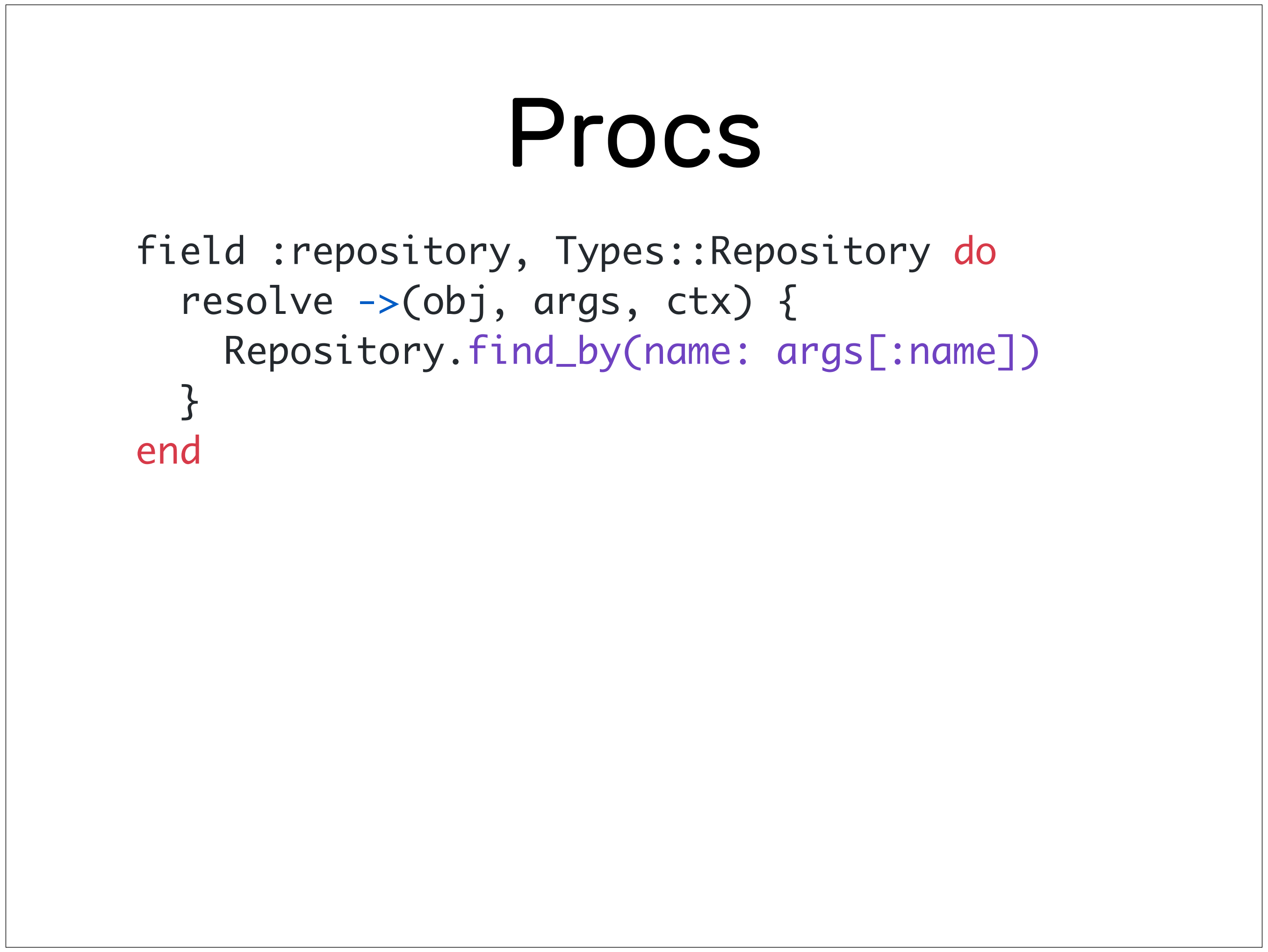
This is not a talk about building a GraphQL system for your app. If you’re interested in more of that, I recommend this talk from RailsConf. It was a great introduction.

GraphQL … Ruby?

Three Years of Design Mistakes

GraphQL-flavored Ruby

So, how was GraphQL-Ruby written? Well, I was a bit younger, and very excited about GraphQL. I thought, doesn't everyone want to learn the details about GraphQL, and build a GraphQL system using Ruby? So, I took GraphQL primitives and forced them into Ruby. Now I call it, GraphQL-flavored Ruby, where normal Ruby code has an unusual taste to it.

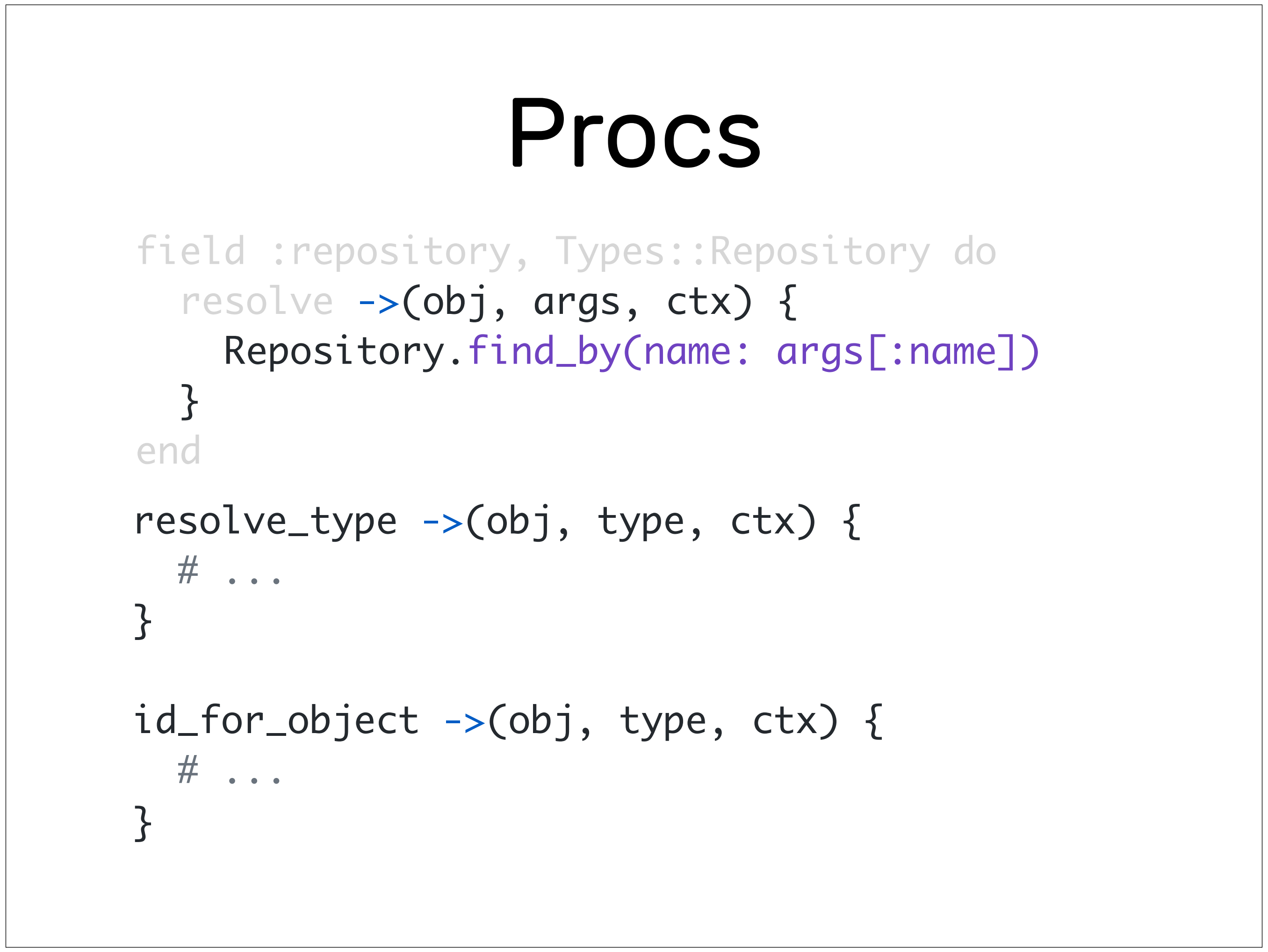


Procs

```
field :repository, Types::Repository do
  resolve ->(obj, args, ctx) {
    Repository.find_by(name: args[:name])
  }
end
```

I mentioned that in GraphQL, fields have something called "resolver function". So, why not try to use Ruby's anonymous functions, procs?

Until the last few months, this is what a field definition looked like in GraphQL Ruby. Do you see the proc?

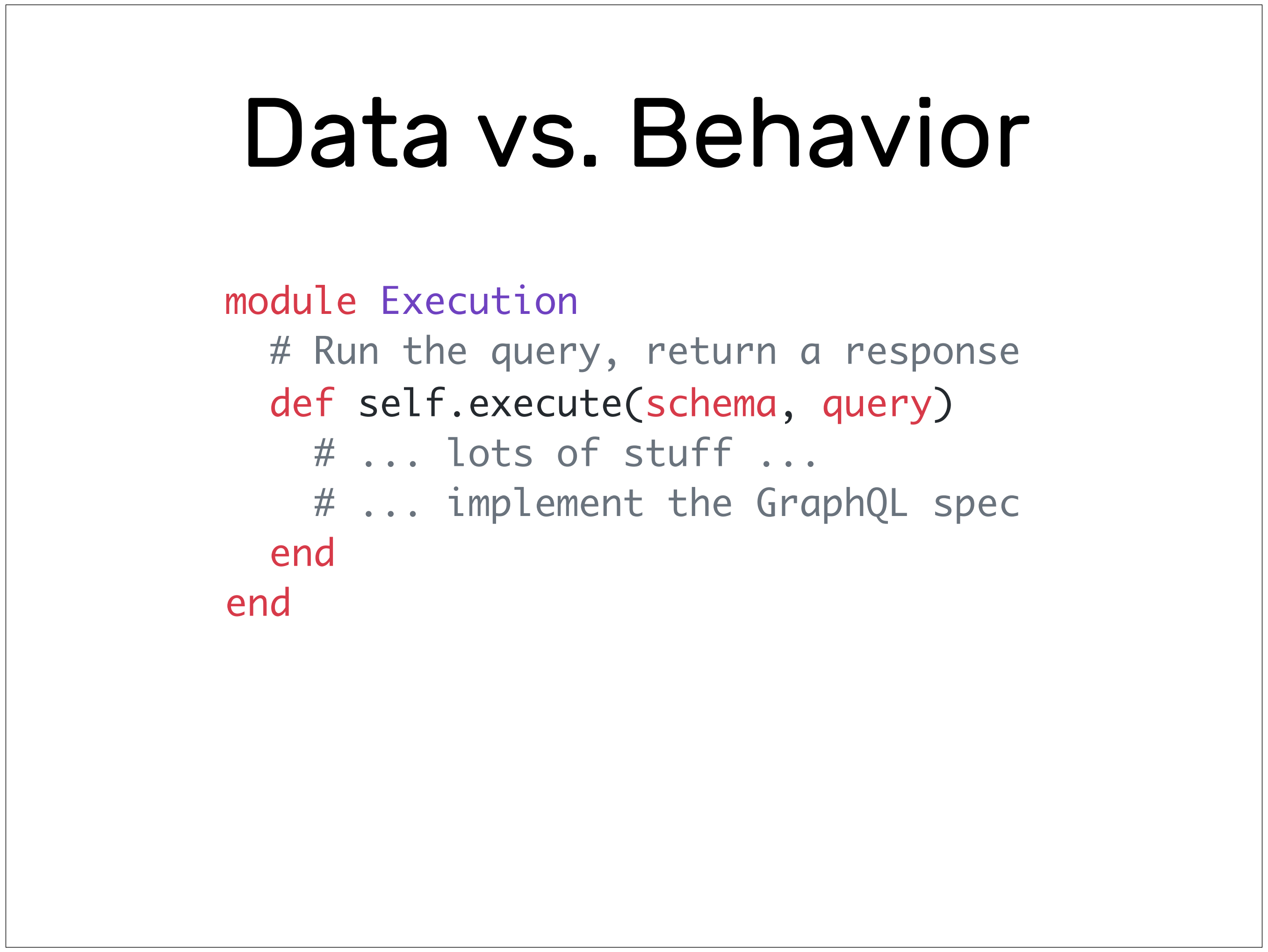


Procs

```
field :repository, Types::Repository do
  resolve ->(obj, args, ctx) {
    Repository.find_by(name: args[:name])
  }
end

resolve_type ->(obj, type, ctx) {
  # ...
}

id_for_object ->(obj, type, ctx) {
  # ...
}
```

It’s right there. There are also a bunch of other pros in the GraphQL-Ruby API.

I learned that there are several good reasons not to use Procs. Unfamiliar, slow, unusual context (different self).

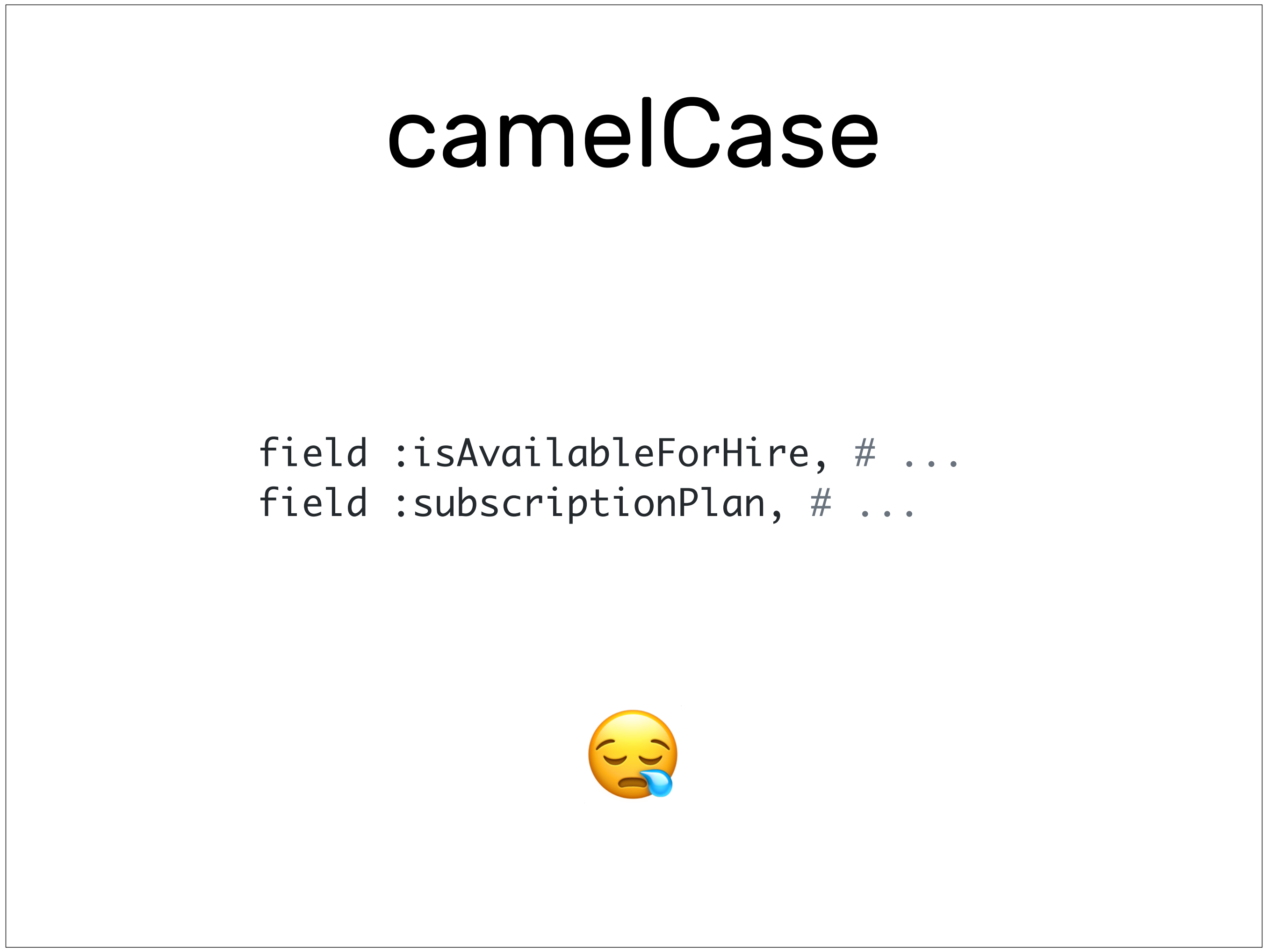


Data vs. Behavior

```ruby
module Execution
  # Run the query, return a response
  def self.execute(schema, query)
    # ... lots of stuff ...
    # ... implement the GraphQL spec
  end
end
```

This is an idea of functional design. On the one hand, you have data structures, and on the other hand, you have algorithms that consume those data structures. This is really nice organization, and it fits well with a literal reading of the GraphQL specification, where the runtime is described as an algorithm.

But it's not a good fit for Ruby: we want to hack, and functional design is not hackable! What I mean is, we want super-classes, and wrapper methods, and overridable hooks. Think about Rails features that work that way.

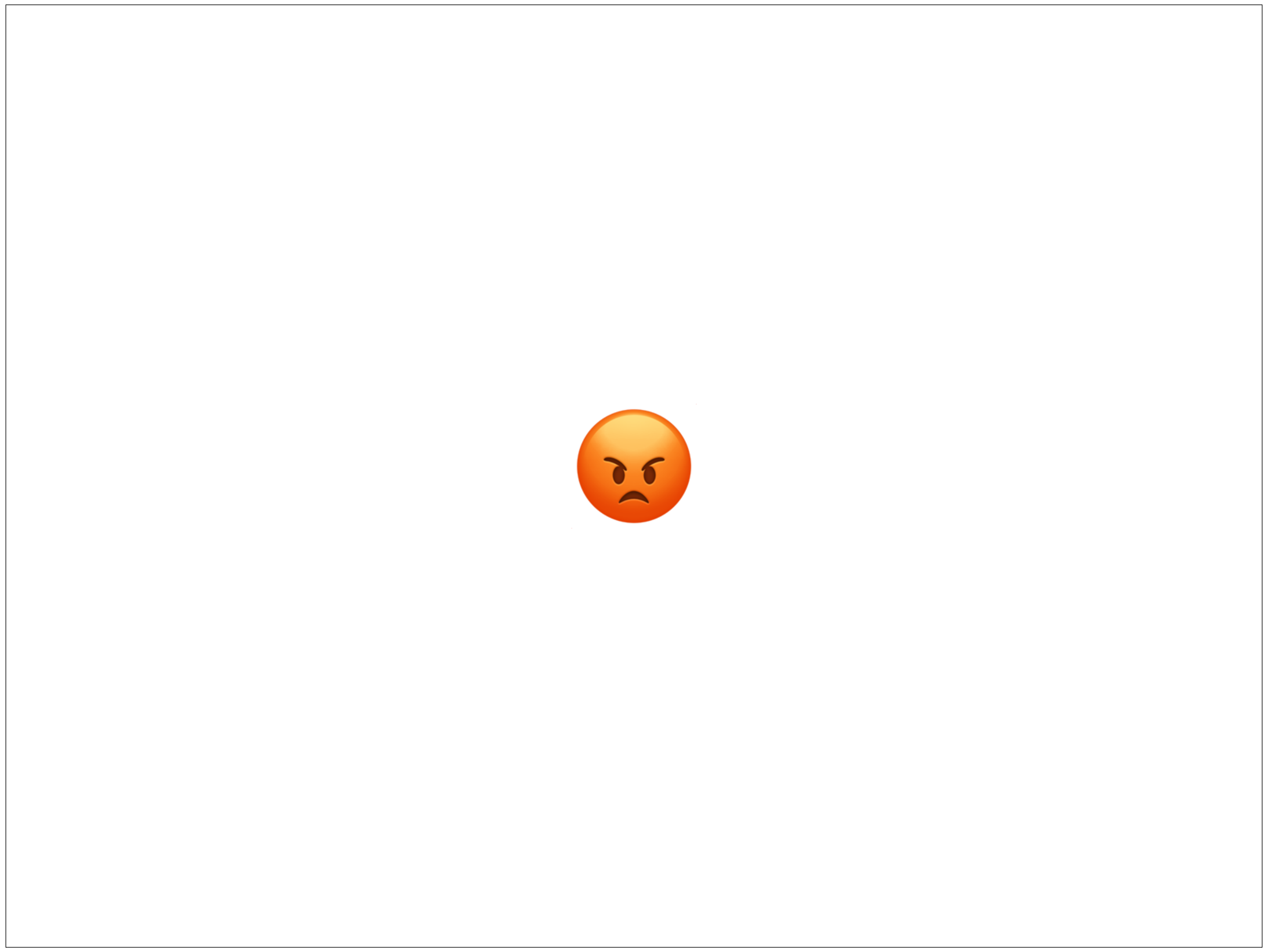


camelCase

```
field :isAvailableForHire, # ...
field :subscriptionPlan, # ...
```

😪

Finally, the most grievous offense of all. GraphQL conventionally uses camelcase, but Ruby uses underscore case. In my haste, I put camelcase symbols in Ruby! It’s not a big deal, but every day it makes you cringe, and it adds up!



😡

GraphQL-flavored Ruby was not productive. It makes you feel off-balance, and throws weird errors, and … camelcase!

GraphQL Friction #414

Open

I really learned about these issues after starting a new job, at GitHub. GitHub took a big bet on GraphQL and the folks there, my teammates now, learned a lot more about managing and deploying GraphQL than I did. So I listened, and asked some questions, and we worked together to try a new paradigm for GraphQL.

Ruby-flavored GraphQL?

So, what if we turn the old idea upside-down? Let’s take normal Ruby paradigms *first*, and use them to build a GraphQL API. What kind of paradigms are we looking for?

- Classes and inheritance
- Mutability and state
- Imperative, hackable execution

Inheritance

```ruby
class Types::User < Types::BaseObject








end
```

First, it means we should use classes, not singleton type objects. Ruby has really good support for code-sharing with inheritance. Inheritance has some problems, but it also has some advantages, and as Ruby developers, we’re used to using those.

Inheritance

```ruby
class Types::User < Types::BaseObject
  field :repos, [Types::Repositories], null: false do
    argument :first, Integer, required: true
  end







end
```

Inside a class, you can make fields…

Inheritance

```
class Types::User < Types::BaseObject
  field :repos, [Types::Repositories], null: false do
    argument :first, Integer, required: true
  end

  def repos(first:)
    object.repositories.first(first)
  end


end
```

And the resolve functions are just methods. Arguments are … arguments!

Inheritance

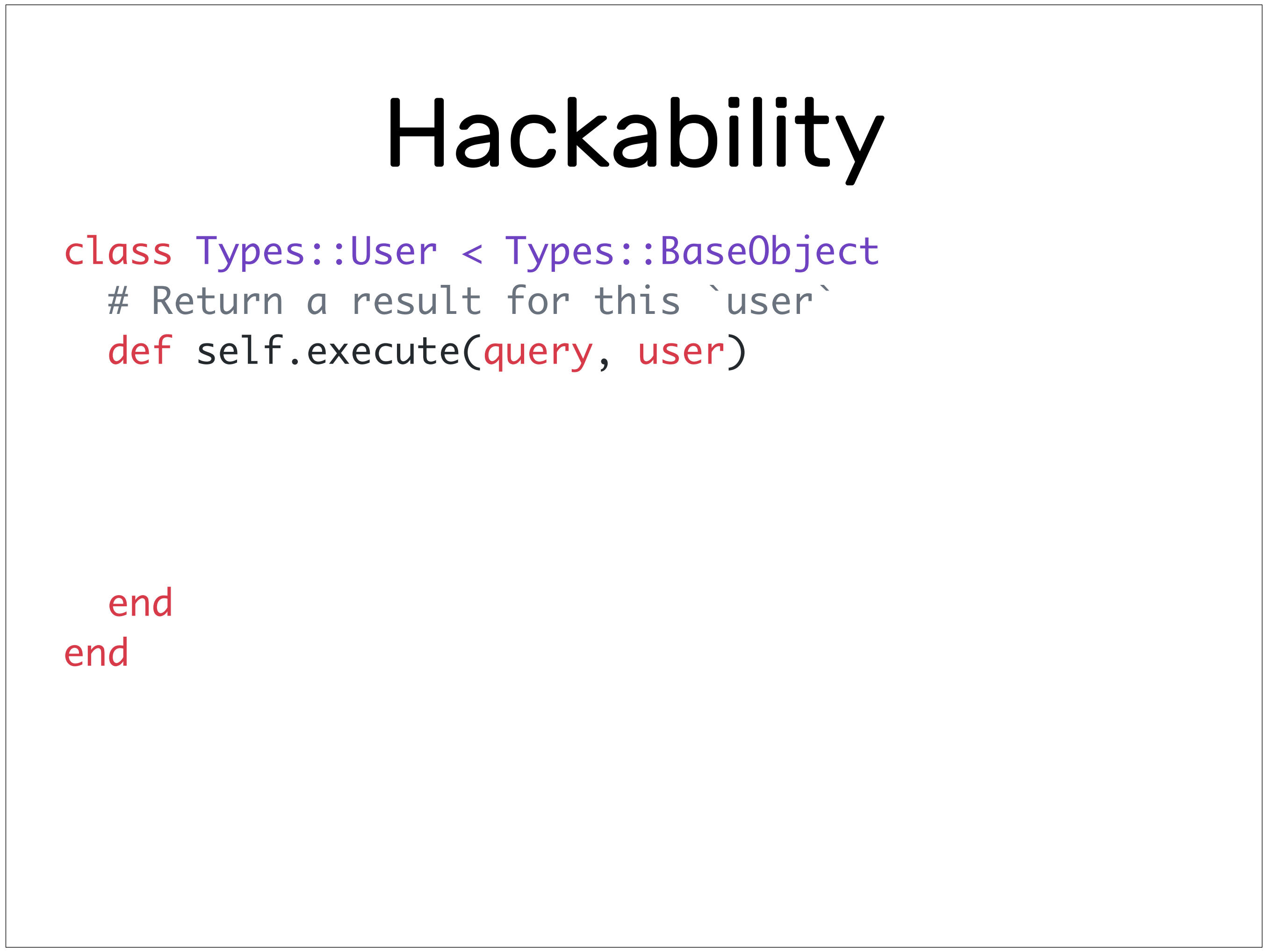
```
class Types::User < Types::BaseObject
  field :repos, [Types::Repositories], null: false do
    argument :first, Integer, required: true
  end

  def repos(first:)
    object.repositories.first(first)
  end
  # RuntimeError ...
  # from types/user.rb:7:in `repos'
end
```

Again, the benefits here are familiarity, debugability, and probably a small performance gain. Since we're using methods, you can use inheritance to share resolve functions, and `super` to call the default behavior, etc.

State

```
class Types::Repository < Types::BaseObject
  field :committers, [Committer], null: false

  def committers
    @committers ||= GitClient.fetch_committers(object)
  end
end
```

The GraphQL spec describes a mostly-stateless algorithm for the Runtime. But we don’t have a lot of habits for stateless coding and Ruby’s support for it is not good. Previously, GraphQL-Ruby had long-lived singleton instances which were prone to state-related bugs. Now, we use some short-lived objects whose instance variables may be used without the risk of foot-gunning.



Hackability

```ruby
class Types::User < Types::BaseObject
  # Return a result for this `user`
  def self.execute(query, user)




  end
end
```

The current GraphQL execution code is a function that takes a schema and a query as input, and returns a response. What if you want to tweak the way that GraphQL executes? The function is opaque: you can’t split it or extend it. It’s like a black box. So I’d like to put execution code *in the type definitions*, so you can change how each type behaves. This is applying OOP to GraphQL! Also, by putting execution code in little bits, it makes it more approachable to learners.

Hackability

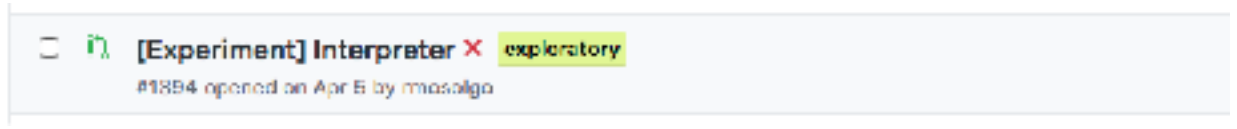
```
class Types::User < Types::BaseObject
  # Return a result for this `user`
  def self.execute(query, user)
    result_hash, duration = with_timing { super }
    # Record some metrics
    Stats.push(“User", result_hash.keys, duration)
    result_hash
  end
end
```

You could use this approach to customize execution, or add some instrumentation, or maybe something else. And I think it will be a bit more approachable to people who want to contribute the library, since it's manageable chunks.

It's also a bit more OOP, since the class holds both the *structure* of the type and the *behavior*, at least, how it's executed.

Hackability

```
class Types::User < Types::BaseObject
  # Return a result for this `user`
  def self.execute(query, user)
    result_hash, duration = with_timing { super }
    # Record some metrics
    Stats.push(“User", result_hash.keys, duration)
    result_hash
  end
end
```


[Experiment] Interpreter ✗ exploratory
#1394 opened on Apr 5 by rmosolgo

This idea is coming together, you can find it on a branch on GitHub.

Underscores

```
field :is_available_for_hire, # ...
field :subscription_plan, # ...
```

😌

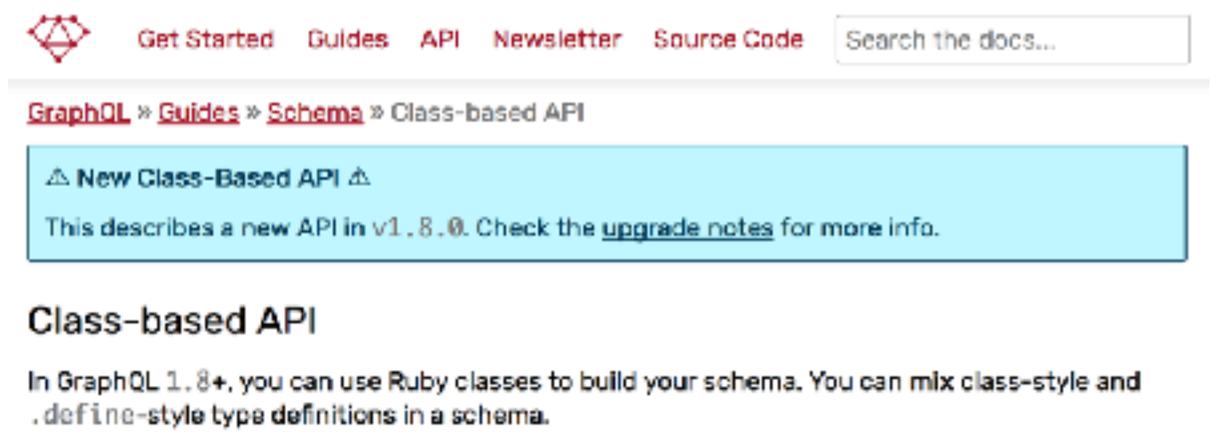
Oh, and one more bonus: all is right in the world, you can write underscores, and they'll be transformed to camelized by default! What a relief.

Ruby-flavored GraphQL

- Leverage Ruby Paradigms:
- Inheritance
- State
- Hackability
- Underscores 😎

v1.8

https://rubygems.org/gems/graphql/versions/1.8.0



GraphQL » Guides » Schema » Class-based API

⚠ New Class-Based API ⚠

This describes a new API in `v1.8.0`. Check the upgrade notes for more info.

Class-based API

In GraphQL `1.8`+, you can use Ruby classes to build your schema. You can mix class-style and `.define`-style type definitions in a schema.

So, some of this is already done! You can get the latest version from rubygems, 1.8, with the new API, and you can find the docs online.

The execution stuff should be coming soon, it’s on my team’s roadmap for this summer!

Thanks!

http://graphql-ruby.org

https://github.com/rmosolgo/graphql-ruby

https://developer.github.com/v4/explorer

So, that’s it for my talk. I also brought some GraphQL-Ruby stickers, so please come say hi if you’d like one!