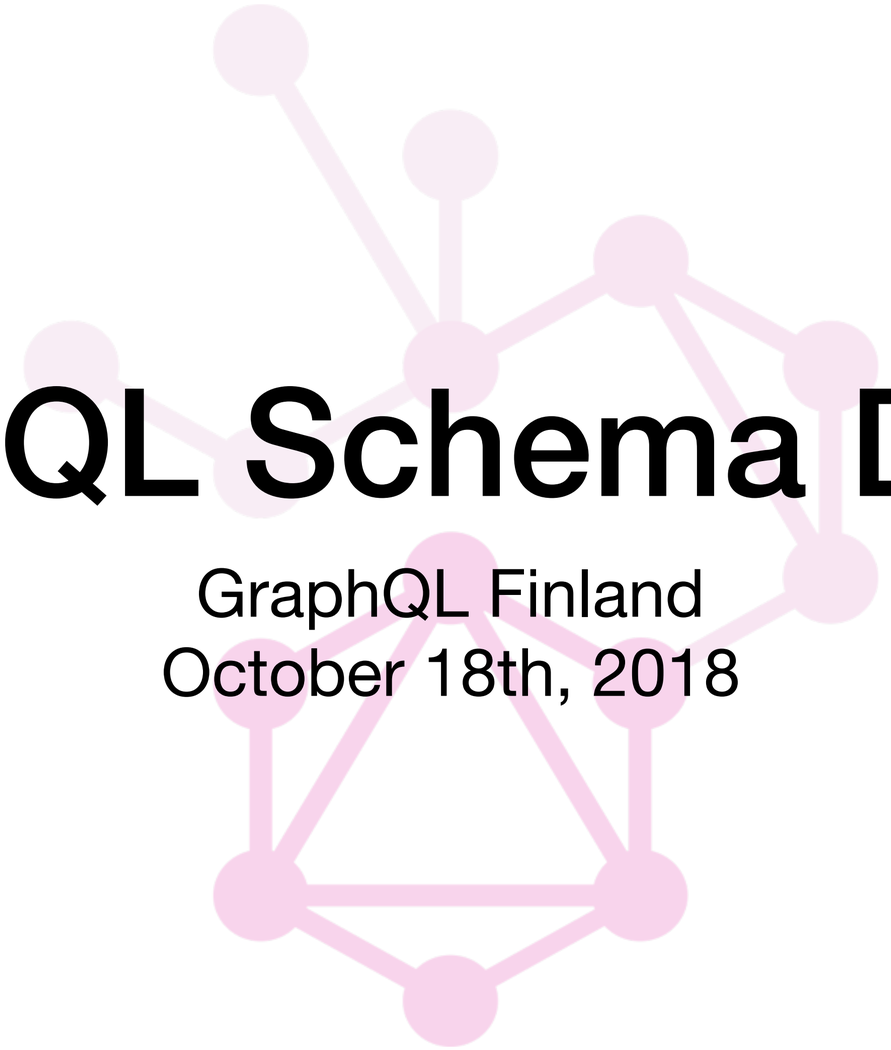


GraphQL Schema Design

GraphQL Finland
October 18th, 2018

@__xuorig__

Marc-André Giroux

Platform Engineer @ GitHub

@__xuorig__

Workshop Overview

- Overview of why, and how to design a great GraphQL API
- Specific tips and tricks for a well designed GraphQL API
- GraphQL in production: best practices, tools, etc.
- If you can and want: Talk about your design problems! Can we solve them together?

Exercises

- Don't be afraid to pair in groups of 2-3-4
- 15-20mins long, take a break ☕ when you're done!
- Design is subjective, no right or wrong answers, but lets discuss the tradeoffs!

Why GraphQL?

Strong schema: Server expresses possibilities

Declarative: Clients select just what they need

Client Centric

Schema Design: The Why

- Build future-proof APIs: best way to mitigate the breaking changes we don't know about yet!
- Build APIs that are easier to reason about.
- Become a better reviewer and understand your domain better.

What GraphQL isn't

not a publicly exposed version
of your database

not a security concern

not your REST resources as a typed schema

not a copy of your UI data on the server

GraphQL lets us model an interface to our business domain

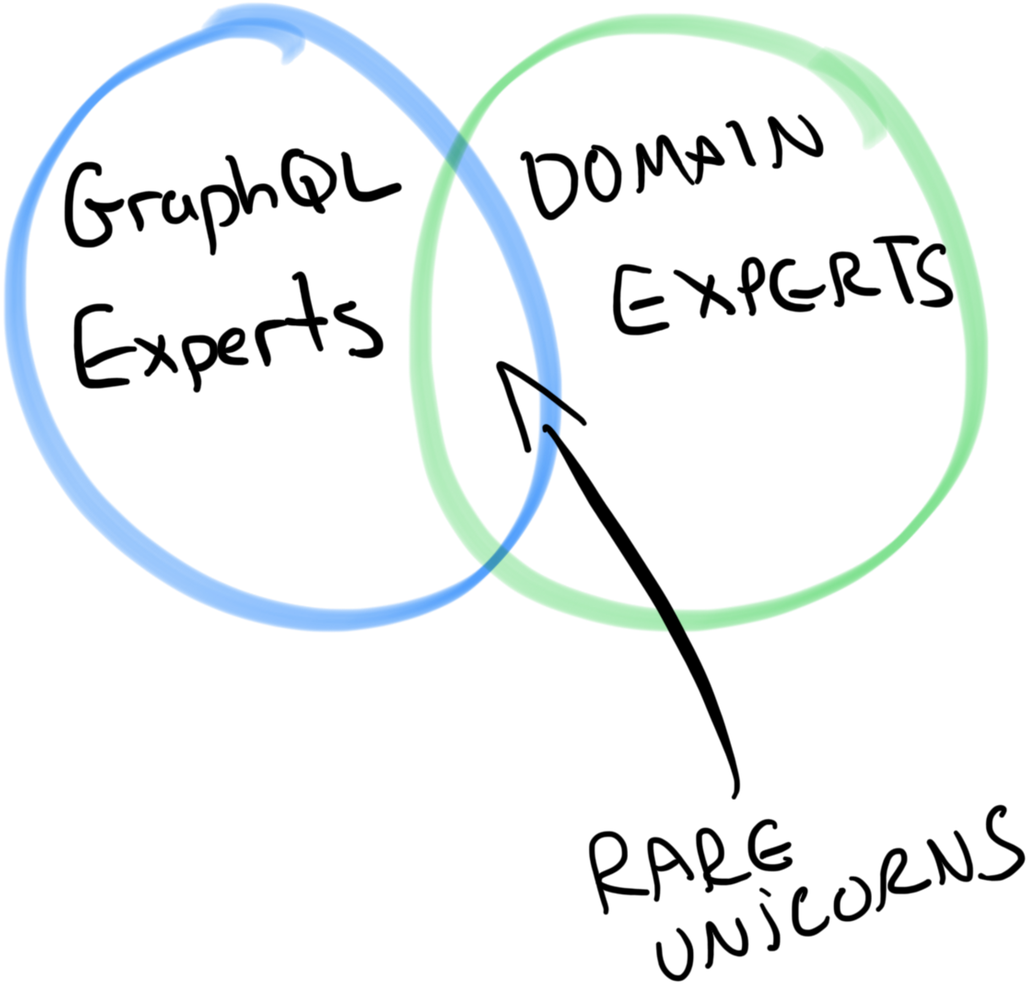
It's often an opportunity to start from fresh! (and maybe get it right this time 🙈)

So lets forget our implementation details, our current APIs, or the way our UI is built!

What does it take to design a great GraphQL API?

1. Know your domain (and sub-domains)

2. Know GraphQL really well (duh)


GraphQL
Experts
DOMAIN
EXPERTS
RARE
UNICORNS

Unless you're one of these unicorns, reach out to domain experts!

Schema Design: The How

Warning!

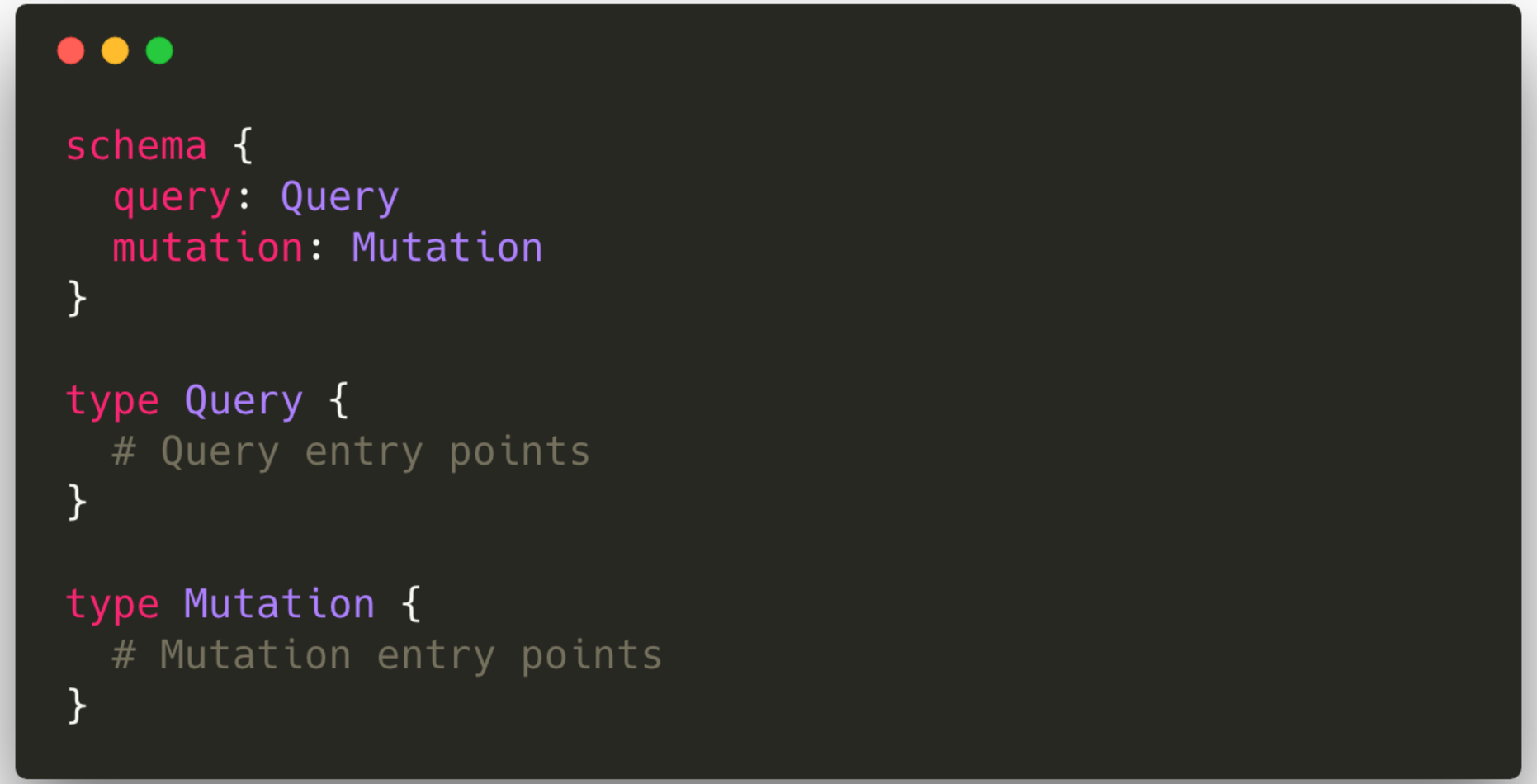
- Like pretty much everything else, GraphQL schema design is a game of tradeoffs! Some of the things we'll cover may work or not for you, which is why it's so important to keep an open mind and really understand your domain first.

The Schema Definition Language (SDL / IDL)

```
type Cat {
  name: String!
  race: CatRace!
}

enum CatRace {
  SIAMESE
  PERSIAN
  MAINECOON
}

type Query {
  cats: [Cat!]!
}

schema {
  query Query
}
```

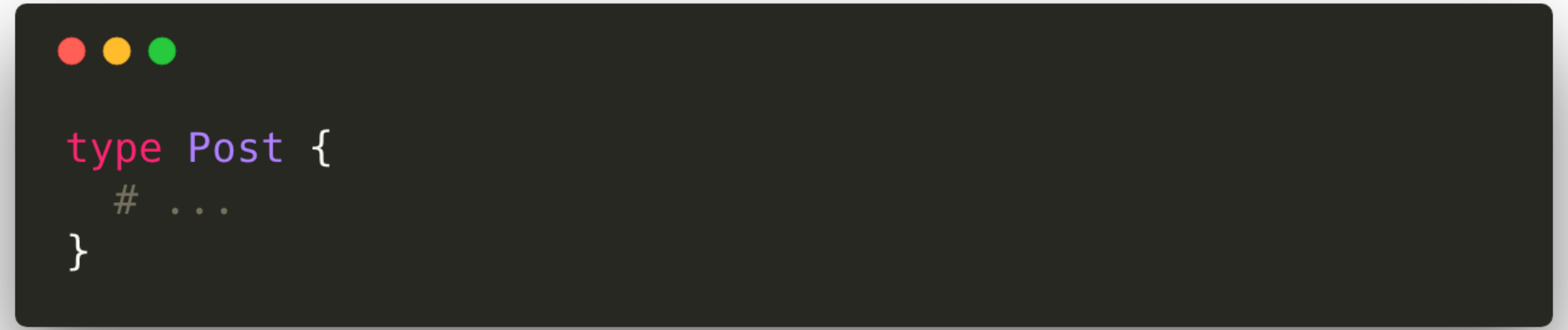
SDL: A Primer

Roots

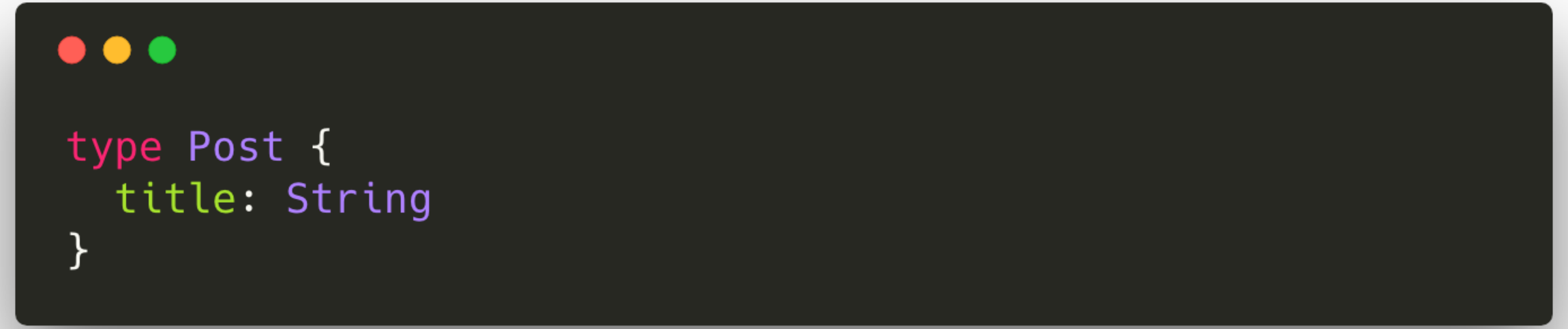


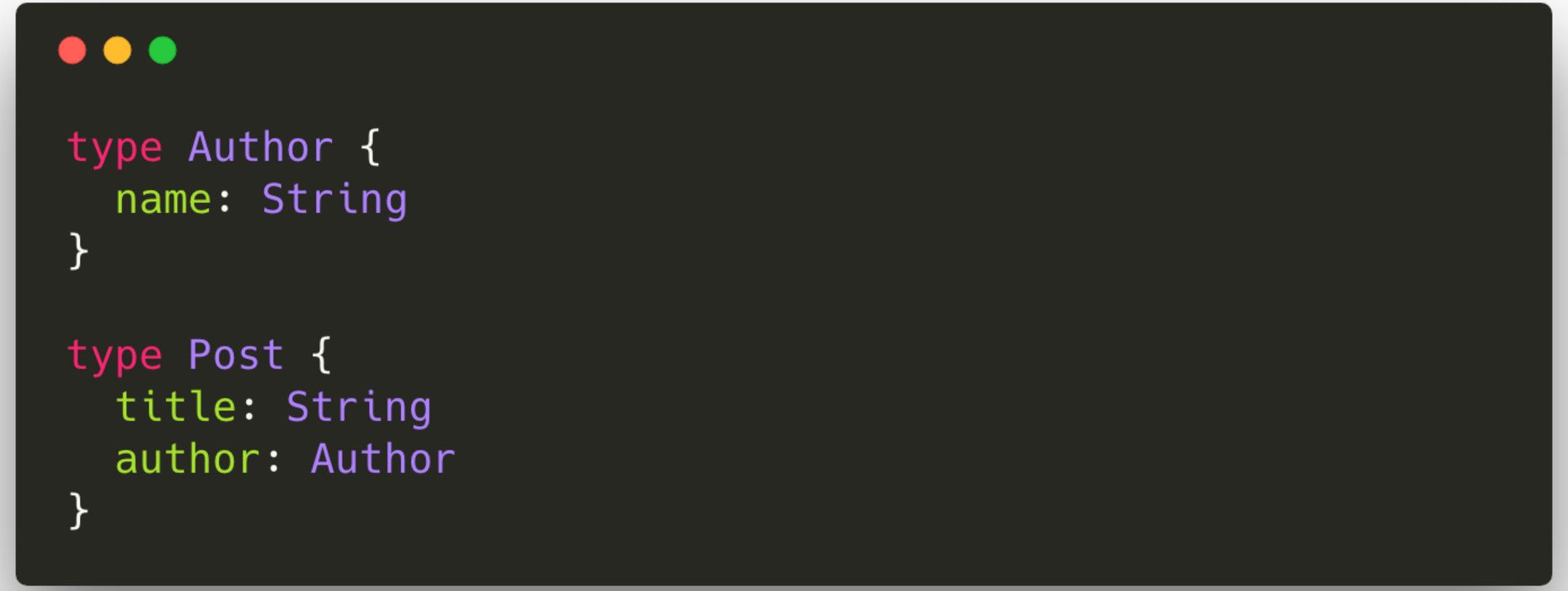
```
schema {
  query: Query
  mutation: Mutation
}

type Query {
  # Query entry points
}

type Mutation {
  # Mutation entry points
}
```

Types


type Post {
# ...
}

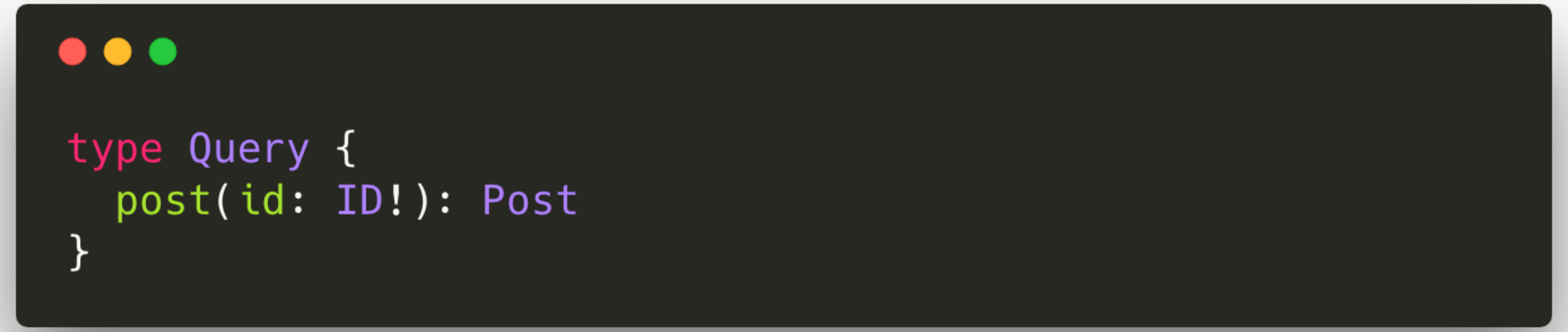
Fields


type Post {
title: String
}

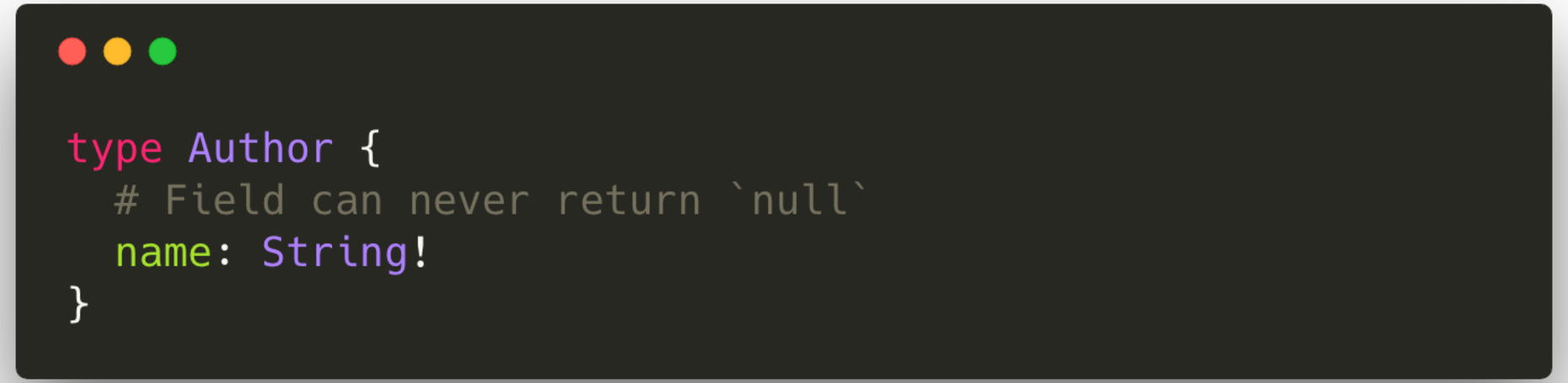


```
type Author {
  name: String
}

type Post {
  title: String
  author: Author
}
```

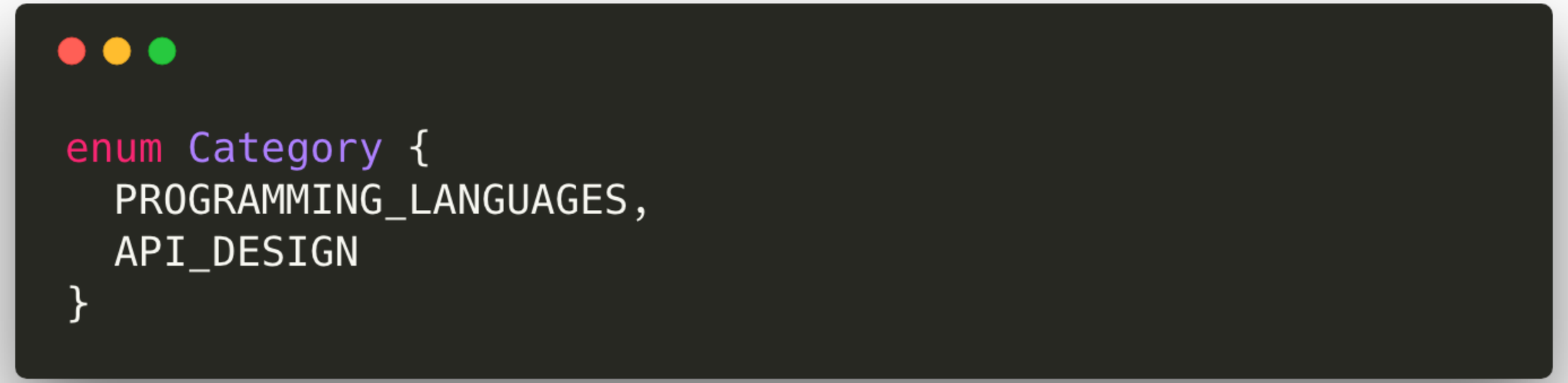
Arguments


type Query {
post(id: ID!): Post
}

Nullability

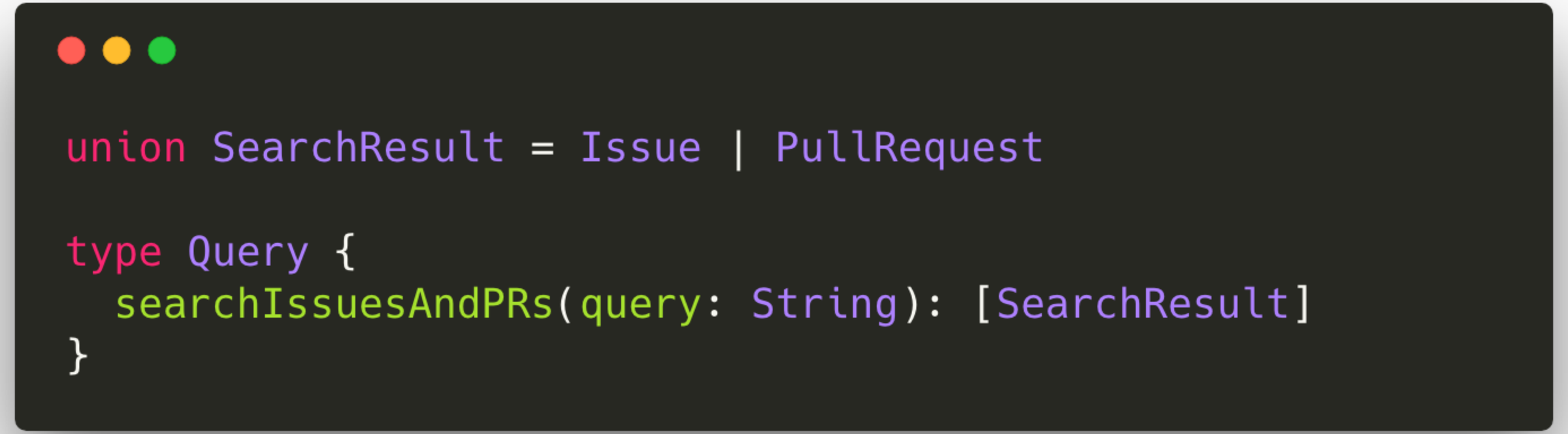

type Author {
# Field can never return `null`
name: String!
}

Enums


enum Category {
PROGRAMMING_LANGUAGES,
API_DESIGN
}

Interfaces & Unions

```
interface Closable {
  isClosed: Boolean!
}

type PullRequest implements Closable {
  isClosed: Boolean!
  title: String!
}
```


union SearchResult = Issue | PullRequest
type Query {
searchIssuesAndPRs(query: String): [SearchResult]
}

Directives

```
type Query {
  post(id: ID!): Post @deprecated(reason: "Stop using this!")
}
```

EX.1

IDL First Design

- Much easier to review
- Universal language
- In most cases implementation can always change, but not the schema
- Good way to encourage design first approach

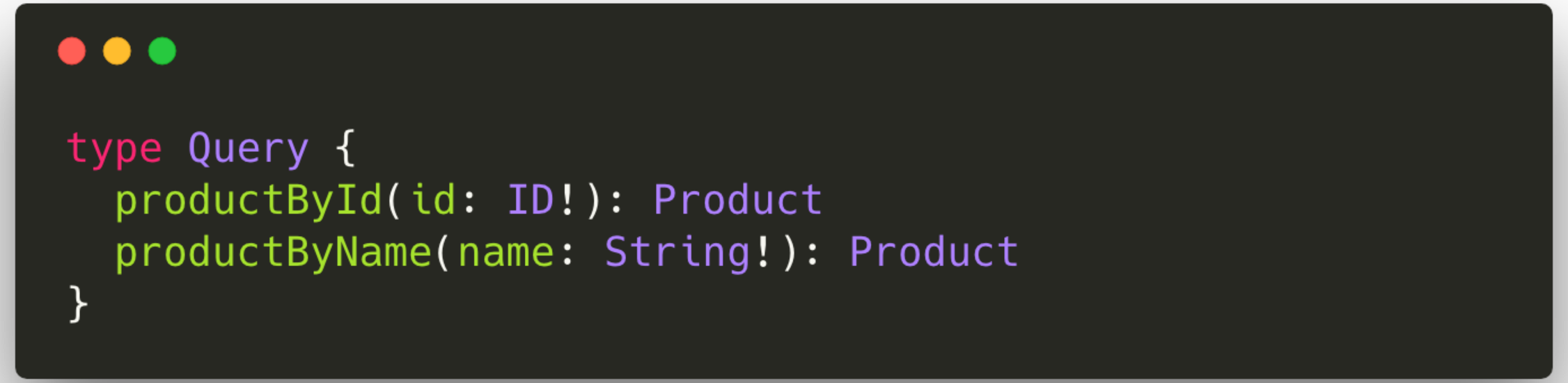
Use the schema, Luke.

EX.2.1

Explicit > Implicit


type Query {
product(id: ID, name: String): Product
}

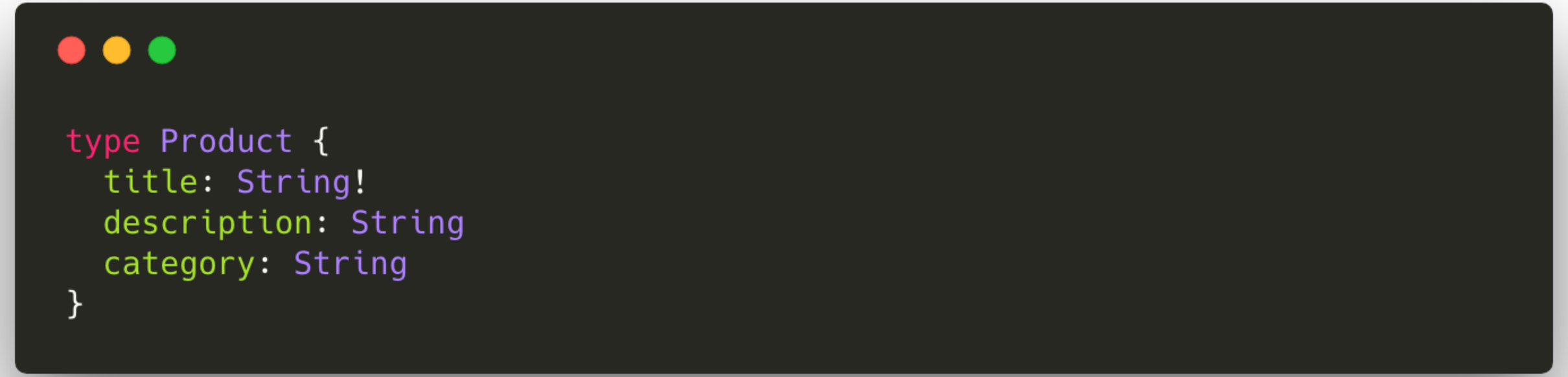
- Schema much weaker
- Harder to use by clients (Need to read more docs, trial and error, etc)
- Resolver logic has to be more complex



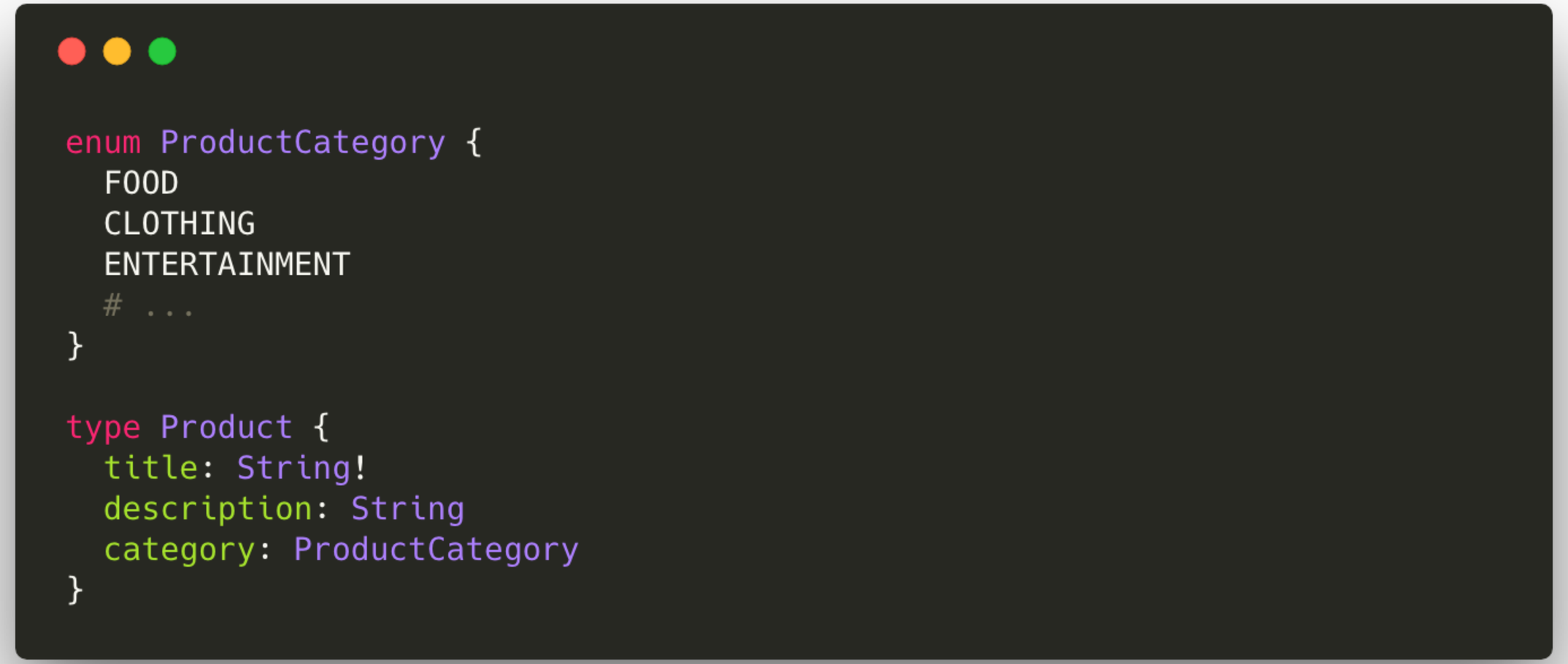
```
type Query {
  productById(id: ID!): Product
  productByName(name: String!): Product
}
```

- More declarative & explicit.
- Very simple to use for clients.
- Resolver logic only takes care of a single use case

EX.2.2


type Product {
title: String!
description: String
category: String
}

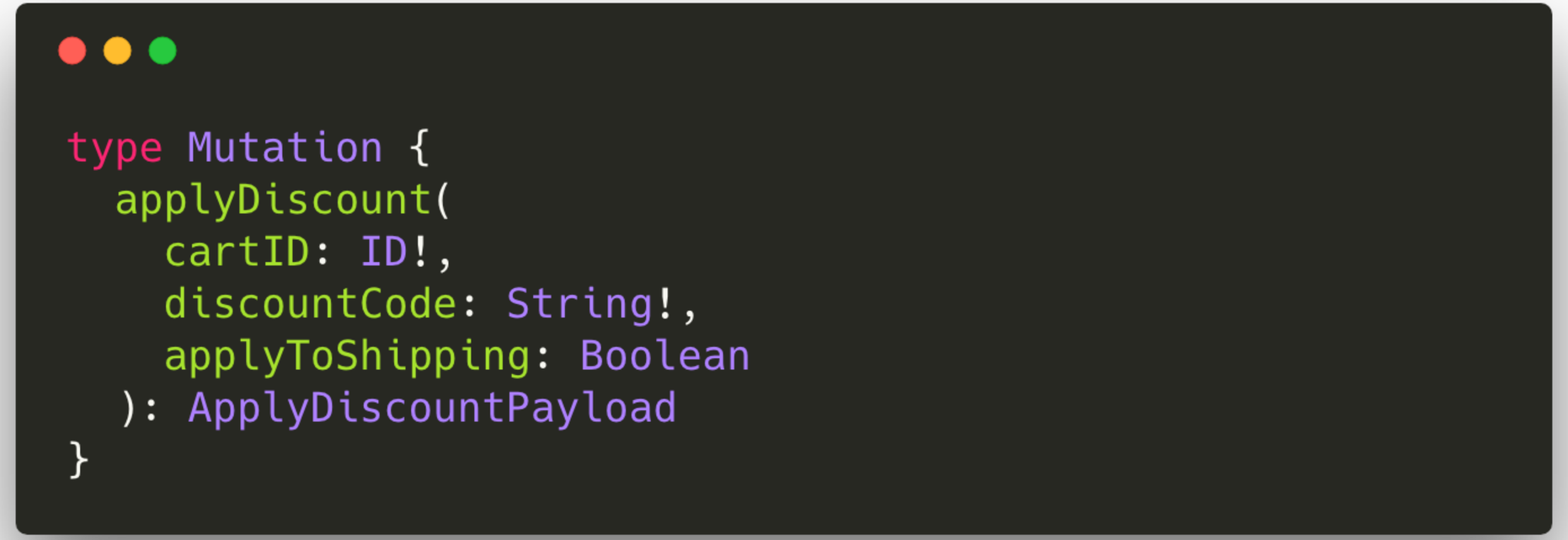
- Clients needs to read documentation / guess possible values
- Server side validation can become tricky



```
enum ProductCategory {
  FOOD
  CLOTHING
  ENTERTAINMENT
  # ...
}

type Product {
  title: String!
  description: String
  category: ProductCategory
}
```

- Schema exposes all possibilities, no more guess work
- Gotchas: Dynamic values, adding values. Tradeoffs!

EX.2.3

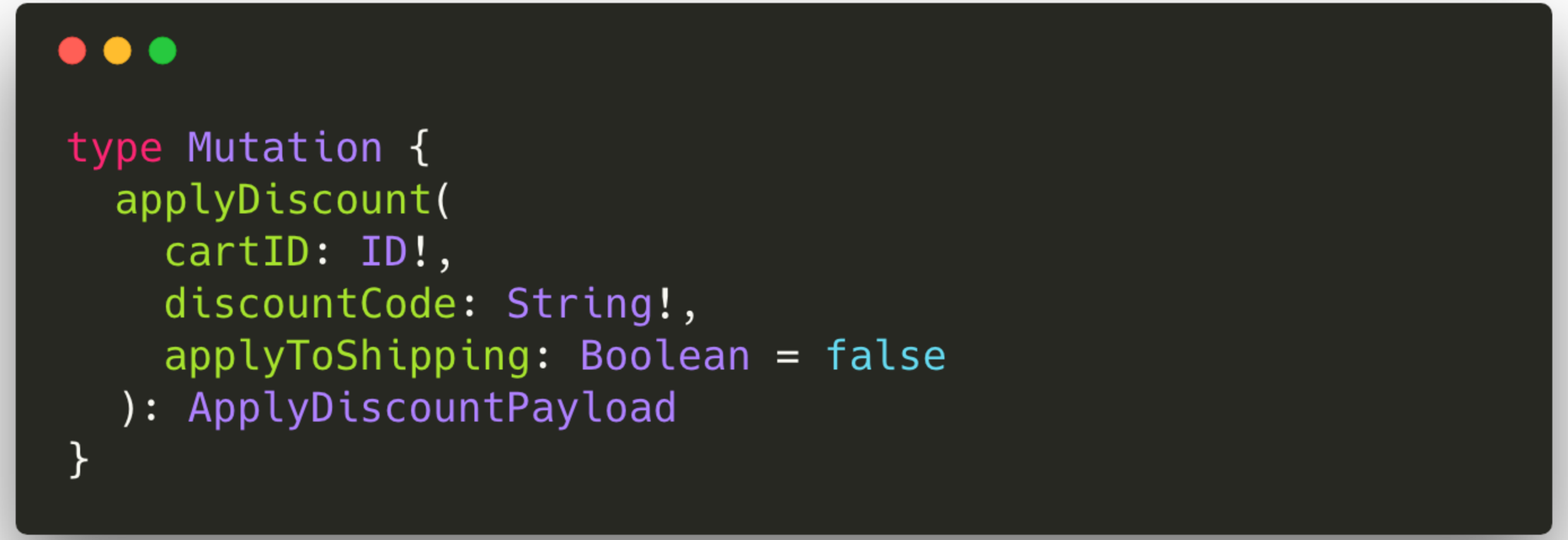


```
type Mutation {
  applyDiscount(
    cartID: ID!,
    discountCode: String!,
    applyToShipping: Boolean
  ): ApplyDiscountPayload
}
```

- if optional: What happens when you omit it, the default case? (Read docs, try it)

- If required: we don't necessarily want to require that if there is a common use case (before shipping)

Use default values for optional arguments, especially for booleans (what does `null` mean?)

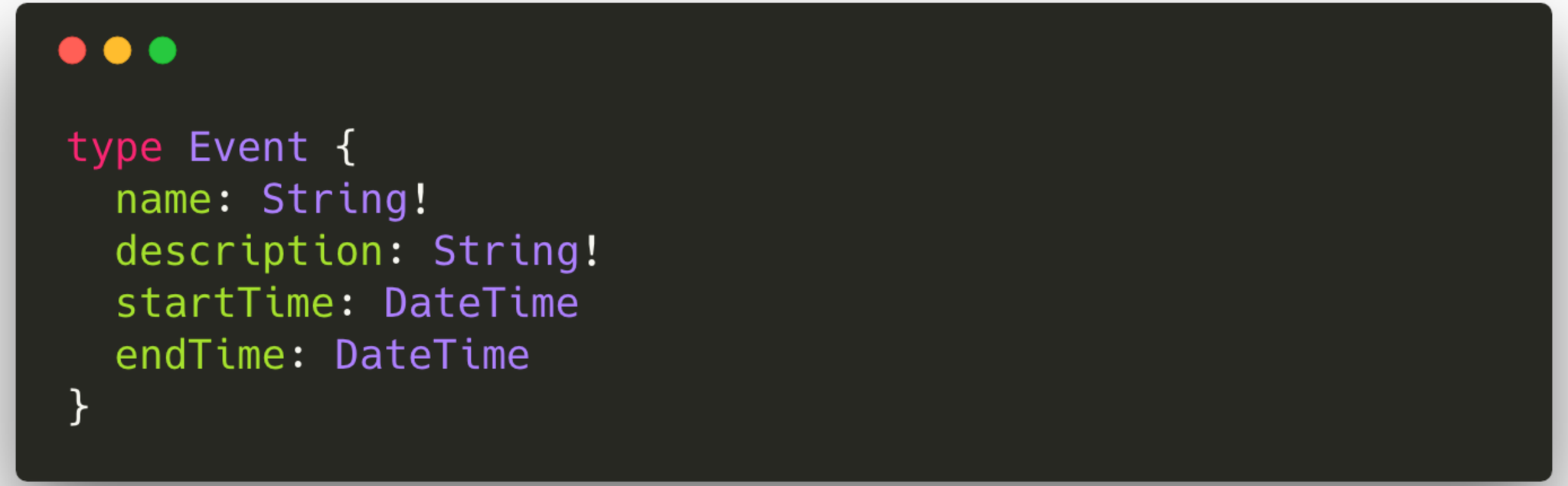


```
type Mutation {
  applyDiscount(
    cartID: ID!,
    discountCode: String!,
    applyToShipping: Boolean = false
  ): ApplyDiscountPayload
}
```

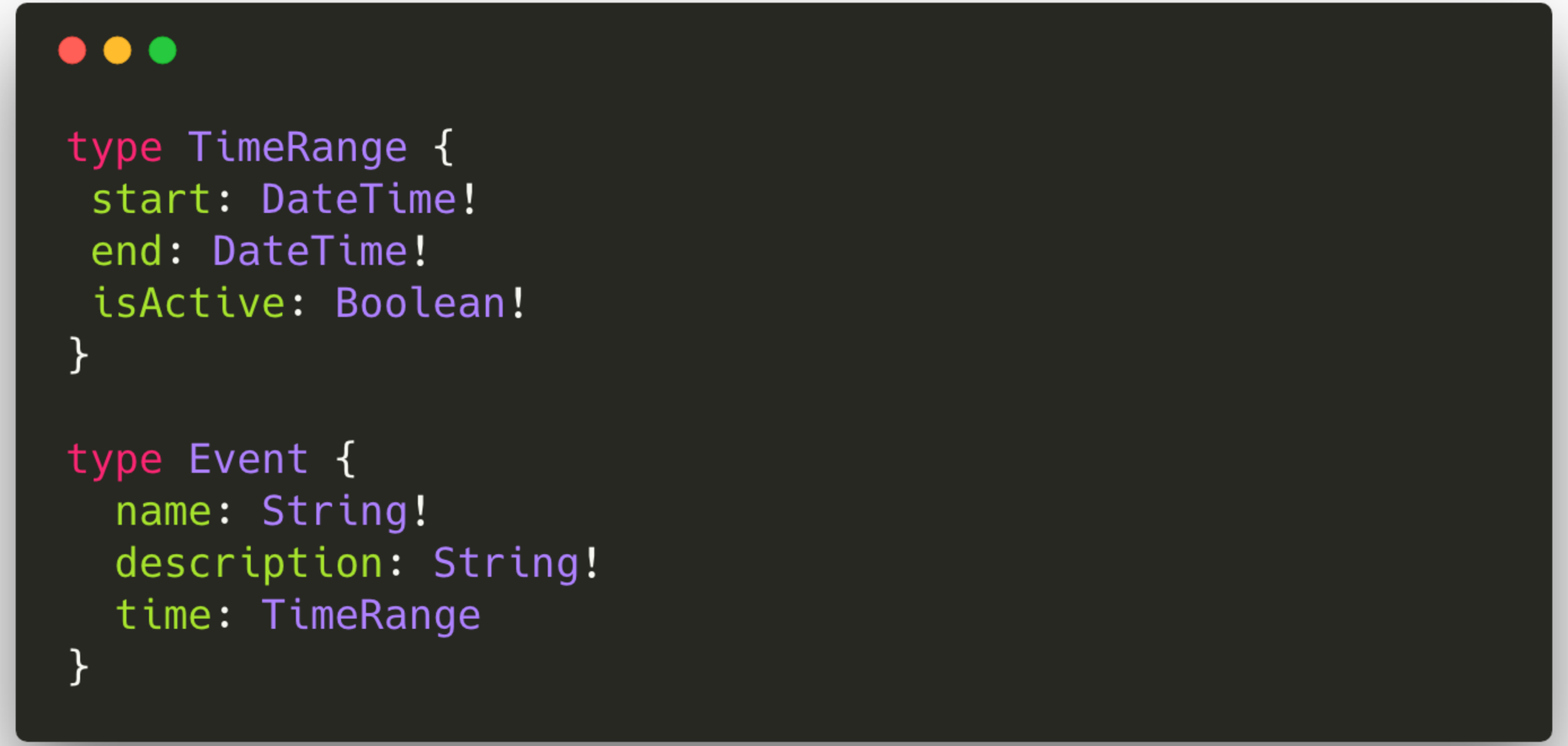
In general: Prefer highly optimized fields over complex generic fields

Adding fields comes at no cost

When in doubt, think domain use cases


type Event {
name: String!
description: String!
startTime: DateTime
endTime: DateTime
}

Don't be afraid to represent things using complex objects



```
type TimeRange {
 start: DateTime!
 end: DateTime!
 isActive: Boolean!
}

type Event {
  name: String!
  description: String!
  time: TimeRange
}
```

- Subtle difference, but make more sense in certain scenario. Before, all fields would have been on the event, all of them nullable.

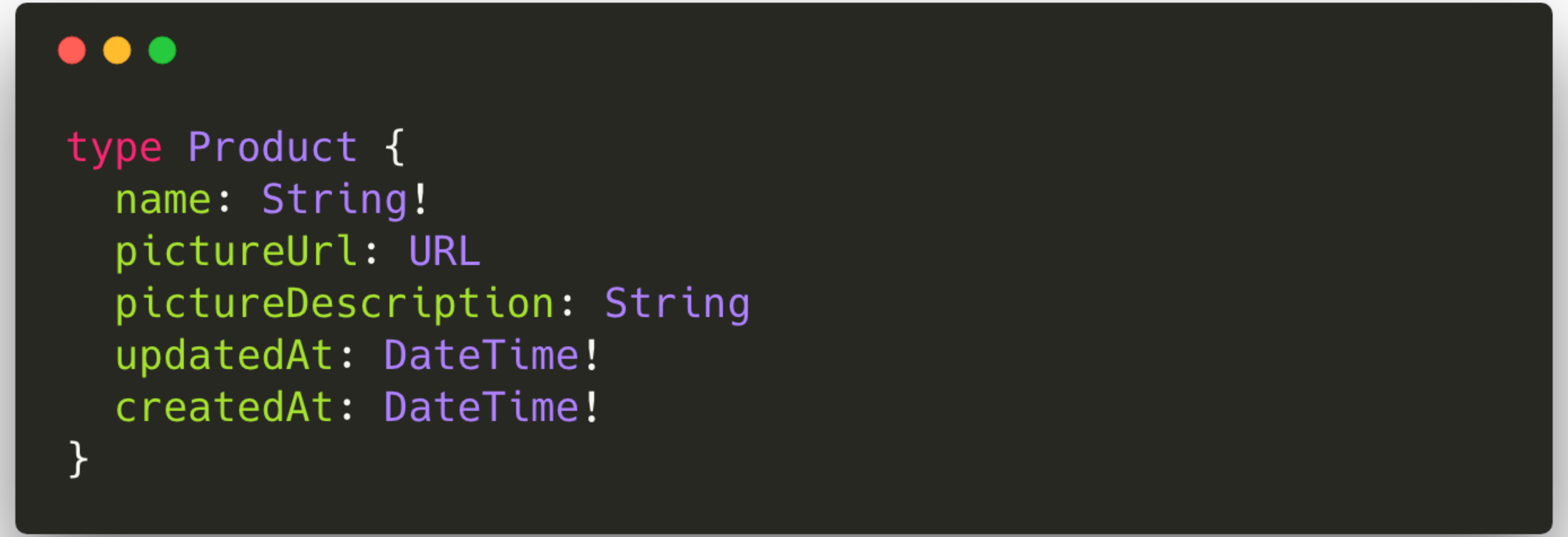
- With a new object, the whole time object may be null if the event hasnt been scheduled yet, but start end, and isActive must be present if it *has* been scheduled.

- Adding more fields to the TimeRange is easy, and does not make the `Event` type more complex.

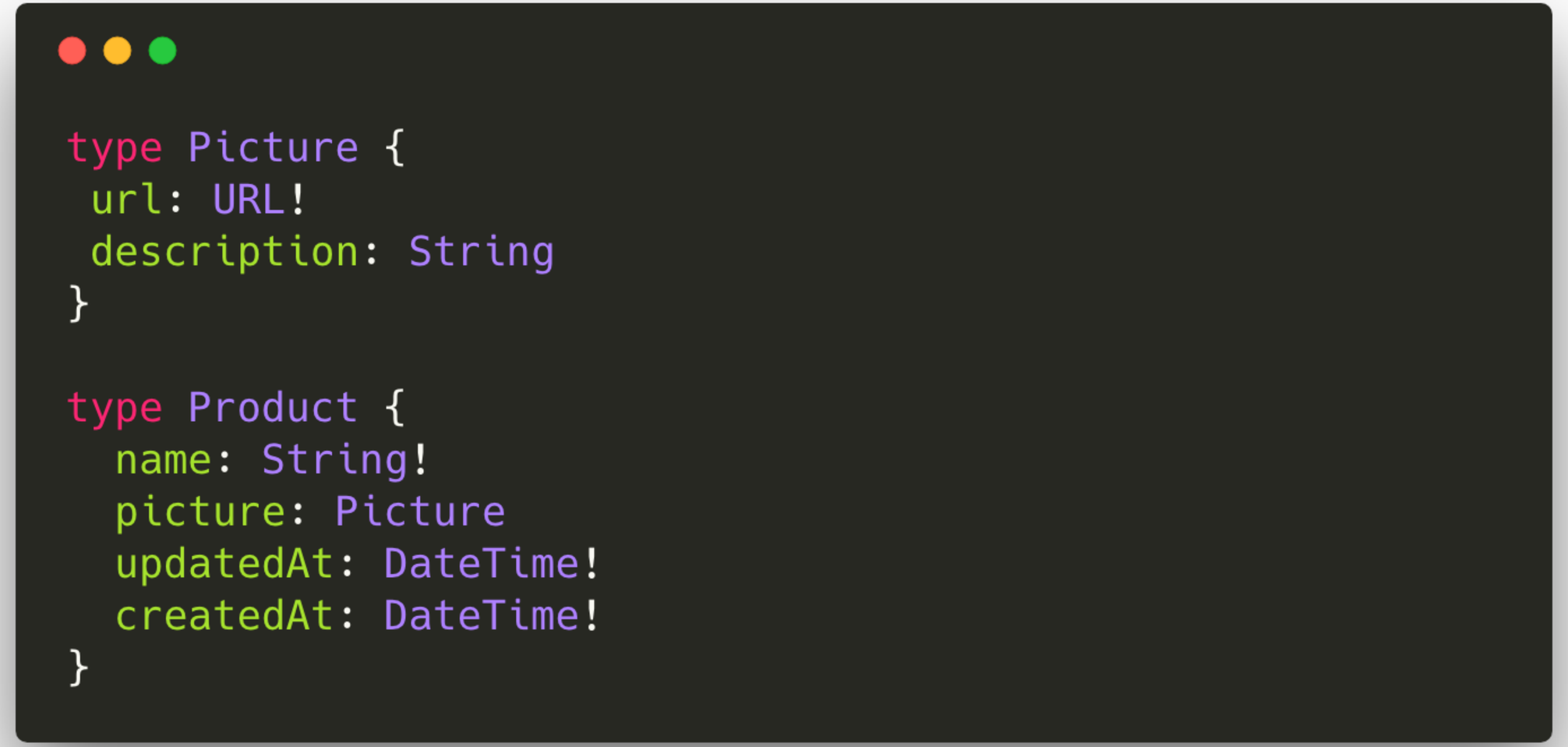
Common mistake: Copying the database schema

```sql
CREATE TABLE `products` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(120) NOT NULL,
  `picture_url` varchar(120) NOT NULL,
  `picture_description` varchar(120) NOT NULL,
  `created_at` datetime NOT NULL,
  `updated_at` datetime NOT NULL,
 );
```


bin/db-to-gql structure.sql schema.graphql



```
type Product {
  name: String!
  pictureUrl: URL
  pictureDescription: String
  updatedAt: DateTime!
  createdAt: DateTime!
}
```



```
type Picture {
 url: URL!
 description: String
}

type Product {
  name: String!
  picture: Picture
  updatedAt: DateTime!
  createdAt: DateTime!
}
```

- Makes more semantic sense, as we explored earlier
- We can now reuse the `Picture` type.
- More discoverable for clients!

Use these tools if needed, but don't blindly trust them!

Using Abstract Types

EX.3.0 & EX.3.1

Interface vs union types: When to use which?

Interfaces

```
interface Discountable {
  price: Money!
  discountCode: String!
}

type Product implements Discountable { ... }
type ShippingRate implements Discountable { ... }
type Gift implements Discountable { ... }
```

Unions

```
union BagOfThings = Product | GiftCard | Discount

type Query {
 searchThings: [BagOfThings!]!
}
```

- Use interface types when types should all conform to a specific behavioral contract.

- Don't use "marker interfaces" as metadata or to copy your existing OOP design.

- Use unions when a field may return one of many unrelated objects with no clear common behavior.

EX.3.2

Don't use the GraphQL schema to share resolvers!

- However tempting it is to use an interface type to "share" fields between types, make sure it make sense to do it, that the shared fields indeed form a "contract" and that the objects behave the same in some ways.

- Naming is a smell: if the only naming that comes to mind sounds like "ObjectDetails", "ObjectFields", or something similar, maybe these are just objects with similar fields with no real contract.

- Use your programming language to achieve sharing through inheritance / composition or write everything twice :)

Domain Driven > Data Driven

EX.4.0

Nullability

EX.7.0

- Using nullability is great to have a stronger schema
- On one end, if everything is nullable, clients need to check for `null` everywhere
- On the other end, if everything is non-nullable, anything that goes wrong risks "bubbling up" and returning less than expected

- What's the balance? Carefuly thinking about what should be nullable or not.

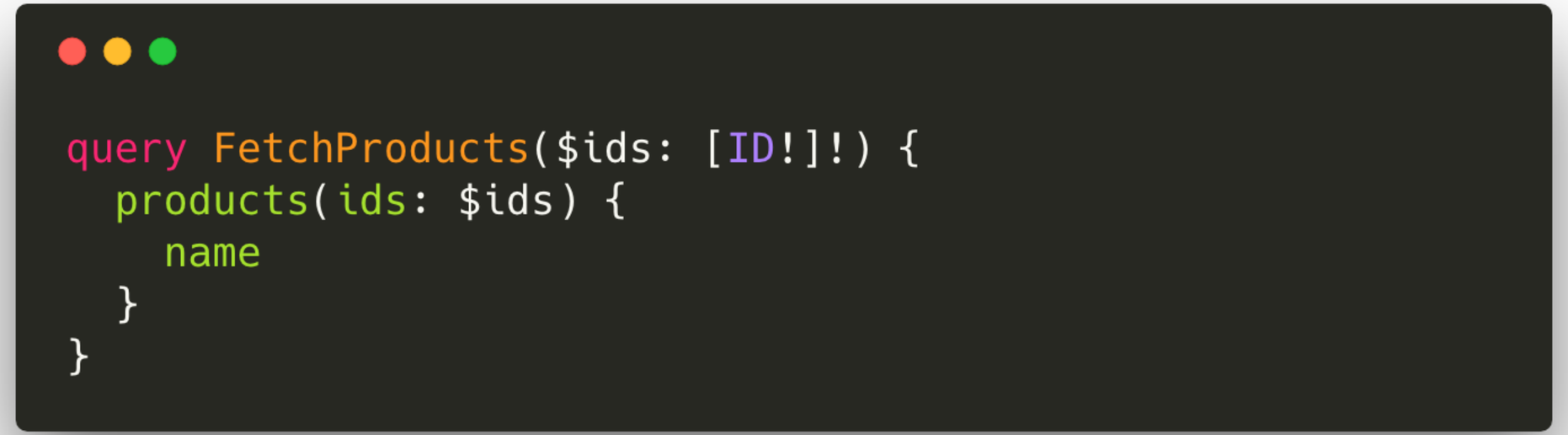
- DB column can be null? Coming from an external service? Is there any chance the field can return null if the stars are aligned? Make it nullable.

- If it simply does not make sense for the object/field to exist without a value, non-nullable.

- Rule of thumb: Scalars repesenting simple data attributes can often be non-null, fields representing "associations" returning object types should often be nullable.

Static Queries & Static Friendly Schema

```
// Building a dynamic query to fetch multiple products

product_queries = products.map(->(product) {
  return "product(id: ${product.id}) { name }"
});

query = gql`
  query {
    ${product_queries.join("\n")}
  }
`;

fetch(query);
```


query FetchProducts($ids: [ID!]!) {
products(ids: $ids) {
name
}
}

EX.5.0

Why Static?

- GraphQL's a query language: use it!
- Tooling: if all our queries are static, this enables a lot of cool tools like for example code generation, editor tooling, etc
- Way easier to reason about usually
- Server side debugging becomes way easier
- Enables persisted queries

Guidelines for a static friendly schema

- Offer a plural version of fields and mutations
- `node` and `nodes` are good examples of this
- Can we provide just the plural version? Tradeoff, not as fun to use for clients needing only the singular.

Input Coercion

When expected as an input, list values are accepted only when each item in the list can be accepted by the list's item type.

If the value passed as an input to a list type is *not* a list and not the **null** value, then the result of input coercion is a list of size one, where the single item value is the result of input coercion for the list's item type on the provided value (note this may apply recursively for nested lists).

This allow inputs which accept one or many arguments (sometimes referred to as "var args") to declare their input type as a list while for the common case of a single value, a client can just pass that value directly rather than constructing the list.

Following are examples of input coercion with various list types and values:

Expected Type	Provided Value	Coerced Value
`[Int]`	`[1, 2, 3]`	`[1, 2, 3]`
`[Int]`	`[1, "b", true]`	Error: Incorrect item value

Input Coercion

When expected as an input, list values are accepted only when each item in the list can be accepted by the list's item type.

If the value passed as an input to a list type is *not* a list and not the **null** value, then the result of input coercion is a list of size one, where the single item value is the result of input coercion for the list's item type on the provided value (note this may apply recursively for nested lists).

This allow inputs which accept one or many arguments (sometimes referred to as "var args") to declare their input type as a list while for the common case of a single value, a client can just pass that value directly rather than constructing the list.

Following are examples of input coercion with various list types and values:

Expected Type	Provided Value	Coerced Value
`[Int]`	`[1, 2, 3]`	`[1, 2, 3]`
`[Int]`	`[1, "b", true]`	Error: Incorrect item value

```
type Query {
  products(ids: [ID!]!): [Product]!
}

query {
  # This works!
  products(ids: "JUST ONE ID") {
   name
  }
}
```

Designing Great Mutations

EX.6.0

Design for behaviours or actions over data

Anemic Mutations

```
type Mutation {
  updateCart(input: UpdateCartInput): UpdateCartPayload!
}

input UpdateCartInput {
  email: Email
  shippingAddress: Address
  items: [ItemInput!]
  userAcceptsMarkteting: Boolean
  creditCard: CreditCard
  billingAddress: Address
  # ...
}
```

- Hard to reason about for clients. What data needs to change to "add an item"?
- Errors are more difficult to handle, what if multiple things change at once, what about conflicting states? Possible but not always the best experience.
- The API is not very discoverable. How can I achieve X, Y, and Z?

What are the actual use cases behind this data?

- Adding an Item to a Cart
- Removing an Item from a Cart
- Changing the Shipping Address

```
type Mutation {
  updateShippingAddress(cartID: ID!, address: Address!):
UpdateShippingAddressPayload
  addItem(cartID: ID!, item: Item!, quantity: Int!): AddItemPayload
  removeItem(cartID: ID!, itemID: ID!): RemoveItemPayload
}
```

- Clients know right away what is possible, and how to do it
- Stronger schema, less nullability on inputs, less server side validation needed.
- Errors become a lot more clear to the client, harder to get into weird states.
- Bonus: Easier to map to events than looking at what data changed (Subscriptions?)

EX.6.1

Caveats to this approach:

Atomicity

- What if my UI allows modifying the shipping address, adding, and removing items all at once?

- Mutations are executed in order, but if one fails, the other ones might succeed. No "Transactions".

- With this design, clients most likely will need to do multiple operations to achieve this.

Atomicity vs Granularity: A tradeoff?


WHY NOT BOTH?
memegenerator.net

If needed, why not both?

- Also a tradeoff, bigger schema, more options for clients, not always an advantage.

- Besides the cognitive load for clients, adding fields has no cost in GraphQL, we can use that to our advantage.

Mutation Input Tips

- Sharing input types or not: yet another tradeoff! Sharing has the benefit of reusability, especially for typed clients. However, make sure that inputs are not going to diverge when sharing. (Example: Update & Create Inputs)

- My personal taste: Avoid sharing too much. (Adding types has no real cost, and the evolution benefits are pretty noticable)

- Just like for object types, don't be scared to add more input types.

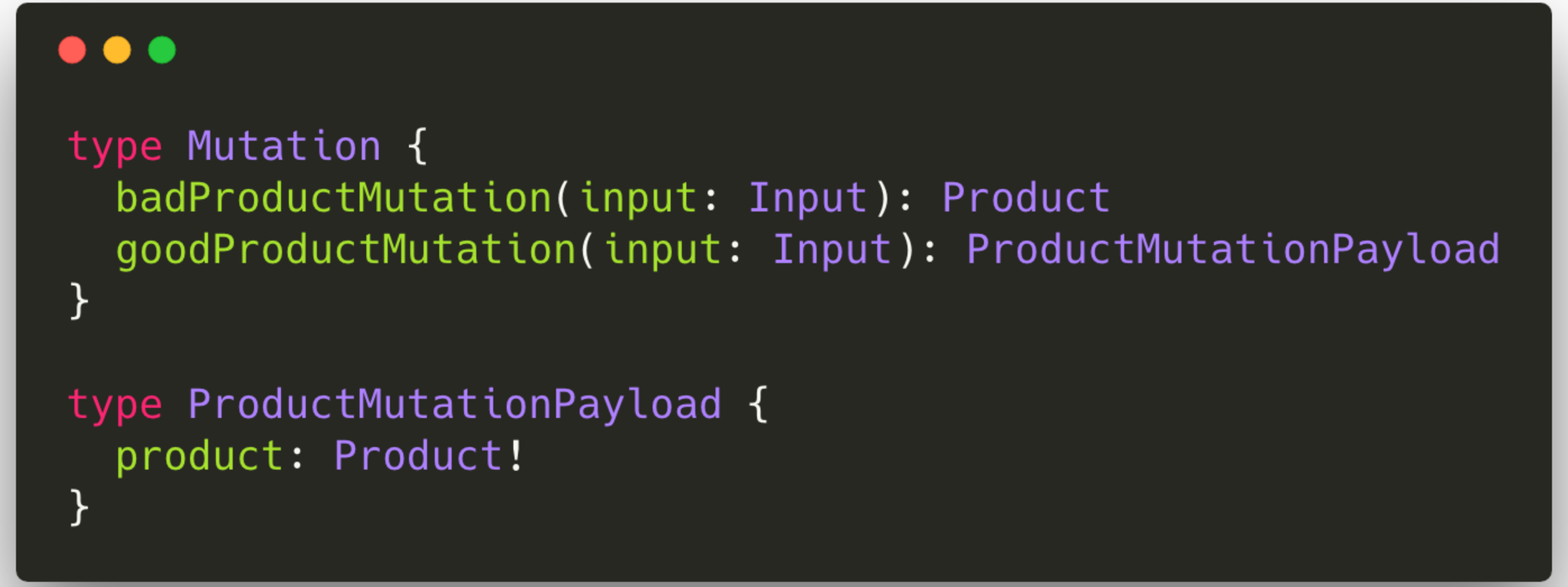
- A more nested input for clients sometimes, but it might save you when it's time to evolve your schema. It also makes more sense to group certain inputs inside an input object type sometimes.

- One Input only, relay convention?
- Do it only if it makes sense. Arguments can be used to create a nice API too. Example:

```
type Mutation {
  createProduct(input: ProductInput): CreateProductPayload
  updateProduct(id: ID!, input: ProductInput): CreateProductPayload
}

input ProductInput {
  name: String!
  description: String!
  price: Float!
}
```

- id at the root, reusing the ProductInput for both create and update.

Mutation Payload Types



```graphql
type Mutation {
  badProductMutation(input: Input): Product
  goodProductMutation(input: Input): ProductMutationPayload
}

type ProductMutationPayload {
  product: Product!
}
```

EX.6.2

Payload Types: What do clients need?

- Entry point(s) to refetch the mutated objects.
- Fields to respect client conventions (Relay 🙉)
- Success / Failure

Many Approaches in the Wild

Representing Success or Failure

Errors

GraphQL Errors

7.1.2 Errors

The `errors` entry in the response is a non-empty list of errors, where each error is a map.

If no errors were encountered during the requested operation, the `errors` entry should not be present in the result.

If the `data` entry in the response is not present, the `errors` entry in the response must not be empty. It must contain at least one error. The errors it contains should indicate why no data was able to be returned.

If the `data` entry in the response is present (including if it is the value **null**), the `errors` entry in the response may contain any errors that occurred during execution. If errors occurred during execution, it should contain those errors.

Error result format

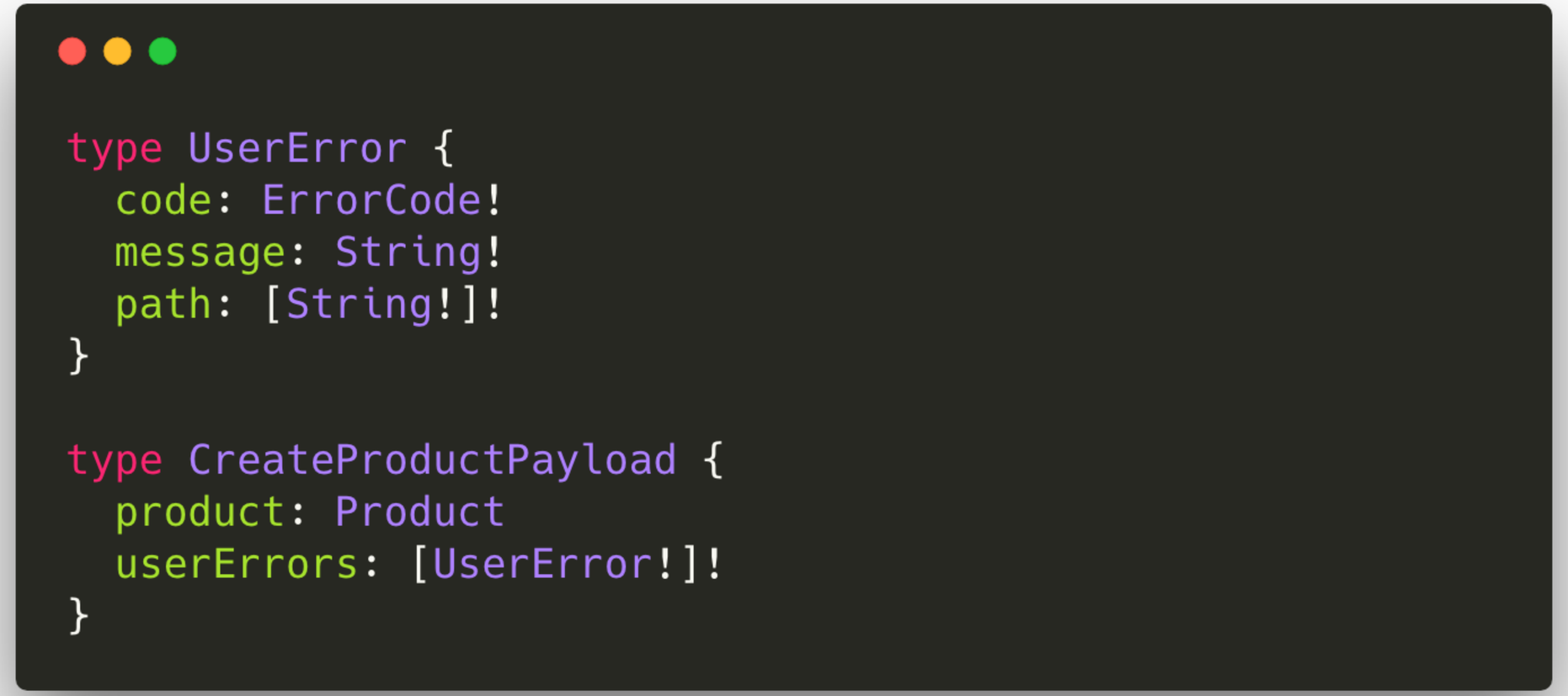
Every error must contain an entry with the key `message` with a string description of the error intended for the developer as a guide to understand and correct the error.

If an error can be associated to a particular point in the requested GraphQL document, it should contain an entry with the key `locations` with a list of locations, where each location is a map with the keys `line` and `column`, both positive numbers starting from `1` which describe the beginning of an associated syntax element.

If an error can be associated to a particular field in the GraphQL result, it must contain an entry with the key `path` that details the path of the response field which experienced the error. This allows clients to identify whether a `null` result is intentional or caused by a runtime error.

- GraphQL Errors have no schema. They're not actually part of the schema.
- This means they're harder to evolve. Harder to discover for clients.
- It also means that any custom fields we have might get overriden by the spec at any time.
- You can use "extensions" for that, but it doesn't seem like the most pleasant experience to clients!
- A field must return `null` if it has errors, what if we still want to return some data on mutation failure?

Errors as Data



```graphql
type UserError {
  code: ErrorCode!
  message: String!
  path: [String!]!
}

type CreateProductPayload {
  product: Product
  userErrors: [UserError!]!
}
```

- We have control over the schema for errors.
- Easier to evolve and very discoverable for clients.
- Since there is not top level error, we can return other data on failure. The current state of the object for example, or any other metadata.
- No risk of clashing with the spec errors in the future.

Experimental: Using Union Types to represent success

```
type Mutation {
  createProduct(input: Input!): CreateProductPayload
}

union CreateProductPayload = CreateProductSuccess | CreateProductFailure

type CreateProductSuccess {
  product: Product!
  # ...
}

type CreateProductFailure {
  userErrors: [UserError!]!
  # ...
}
```

```
mutation {
  createProduct(input: { name: "test" }) {
    ... on CreateProductSuccess {
      product { name }
    }
    ... on CreateProductFailure {
      userErrors { message }
    }
  }
}
```

- Stronger guarantees: if success, Product is never null!
- More complex, need these fragments everytime

Schema Evolution

Designing For Evolution

Naming Things 

- Sometimes, being overly specific might save us later. (i.e ShopLiveEvent vs Event)

- Deprecating just for a name is a breaking change that doesn't give value to clients, avoid it at all cost.

- As covered before: use types!
- Use custom scalars with care (hard to evolve, no idea how they are used, and no way to deprecate parts of them)
- Know the domain as best as you can.

Evolving the Schema: GitHub's Approach

Continuous Evolution?

Evolve your API without versions

Add new fields and types to your GraphQL API without impacting existing queries. Aging fields can be deprecated and hidden from tools. By using a single evolving version, GraphQL APIs give apps continuous access to new features and encourage cleaner, more maintainable server code.

graphql.org

EVOLVE'13 | Keynote | Roy Fielding

[Docs] [txt|pdf|xml|html] [Tracker] [Email] [Diff1] [Diff2] [Nits]

Versions: 00 01 02 03 04 05 06

Network Working Group — E. Wilde
Internet-Draft — CA Technologies
Intended status: Standards Track — November 12, 2017
Expires: May 16, 2018

The Sunset HTTP Header
draft-wilde-sunset-header-04

Abstract

This specification defines the Sunset HTTP response header field, which indicates that a URI is likely to become unresponsive at a specified point in the future. It also defines a sunset link relation type that allows linking to resources providing information about an upcoming resource or service sunset.

https://tools.ietf.org/html/draft-wilde-sunset-header-04

API evolution is the concept of striving to maintain the "I" in API, the request/response body, query parameters, general functionality, etc., only breaking them when you absolutely, absolutely, have to.

- PHIL STURGEON

Our Mission 🚀

1. Avoid breaking changes the best we can.
2. Make it easy for engineers to evolve our schemas.
3. Make it easy for users to evolve their clients.

Avoiding Breaking Changes

Accidental Breaking Changes 😰

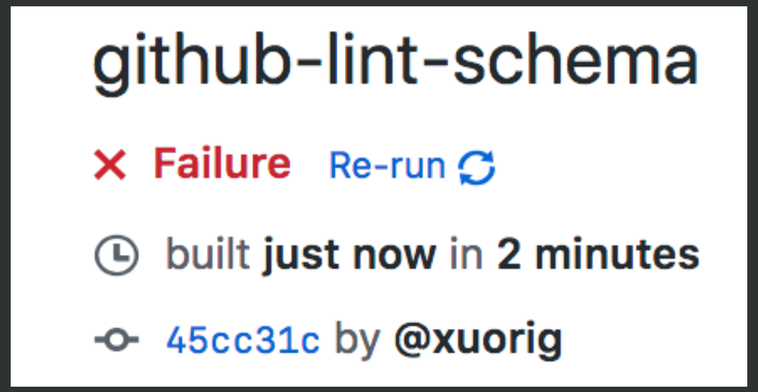
Checked-in GraphQL IDL

```
$ bin/dump-graphql-schema
> config/schema.public.graphql

$ git add config/schema.public.graphql
```

Checked-in GraphQL IDL

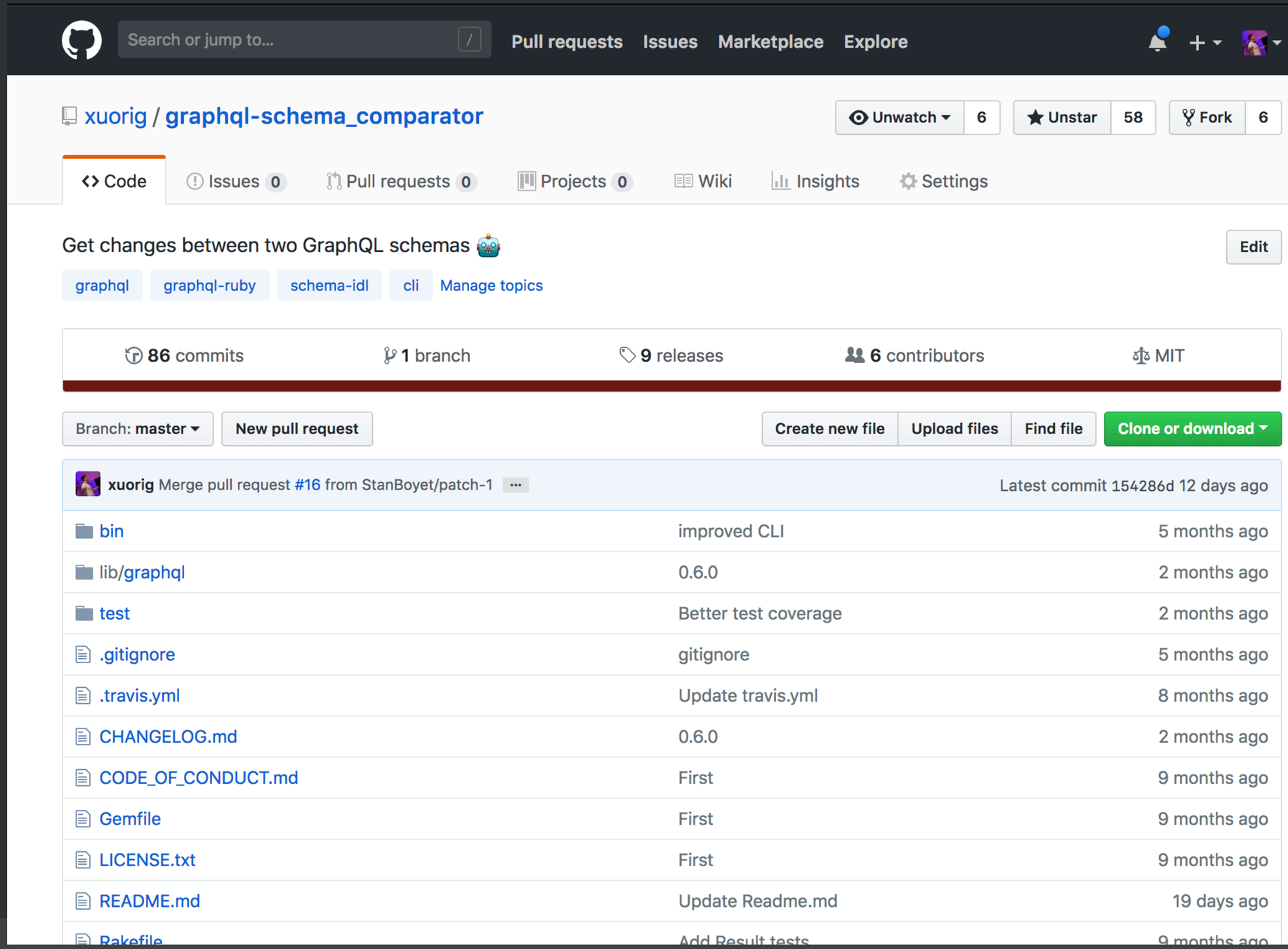
```
8531  # A user is an individual's account on GitHub that owns repositories and can make new content.
8532  type User implements Actor & Node & RepositoryOwner & UniformResourceLocatable {
8533    # A URL pointing to the user's public avatar.
8534    avatarUrl(
8535      # The size of the resulting square image.
8536      size: Int
8537    ): URI!
8538
8539    # The user's public profile bio.
8540    bio: String
8541
8542    # The user's public profile bio as HTML.
8543    bioHTML: HTML!
8544
8545    # A list of commit comments made by this user.
8546    commitComments(
8547      # Returns the elements in the list that come after the specified global ID.
8548      after: String
8549
8550      # Returns the elements in the list that come before the specified global ID.
8551      before: String
8552
8553      # Returns the first _n_ elements from the list.
8554      first: Int
8555
8556      # Returns the last _n_ elements from the list.
8557      last: Int
8558    ): CommitCommentConnection!
```

Checked-in GraphQL IDL




2 lib/platform/objects/pull_request_review_comment.rb
View
@@ -58,8 +58,6 @@ def self.async_viewer_can_see?(permission, object)
58 58 pr_review_comment.async_update_path_uri
59 59 end
60 60
61 - field :pull_request, Objects::PullRequest, method: :async_pull_request, description: "The pull request associ
62 -
63 61 field :pull_request_review, PullRequestReview, method: :async_pull_request_review, description: "The pull req
64 62
65 63 field :commit, Commit, description: "Identifies the commit associated with the comment.", null: false

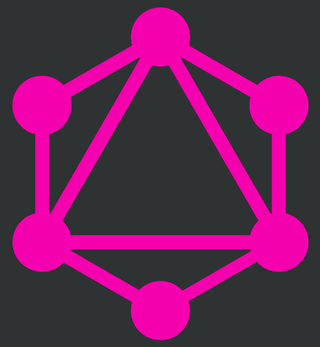

3 config/schema.public.graphql
View
@@ -7278,9 +7278,6 @@ type PullRequestReviewComment implements Comment & Deletable & Node & Reactable
7278 7278 # Identifies when the comment was published at.
7279 7279 publishedAt: DateTime
7280 7280
7281 - # The pull request associated with this review comment.
7282 - pullRequest: PullRequest!
7283 -
7284 7281 # The pull request review associated with this review comment.
7285 7282 pullRequestReview: PullRequestReview
7286 7283

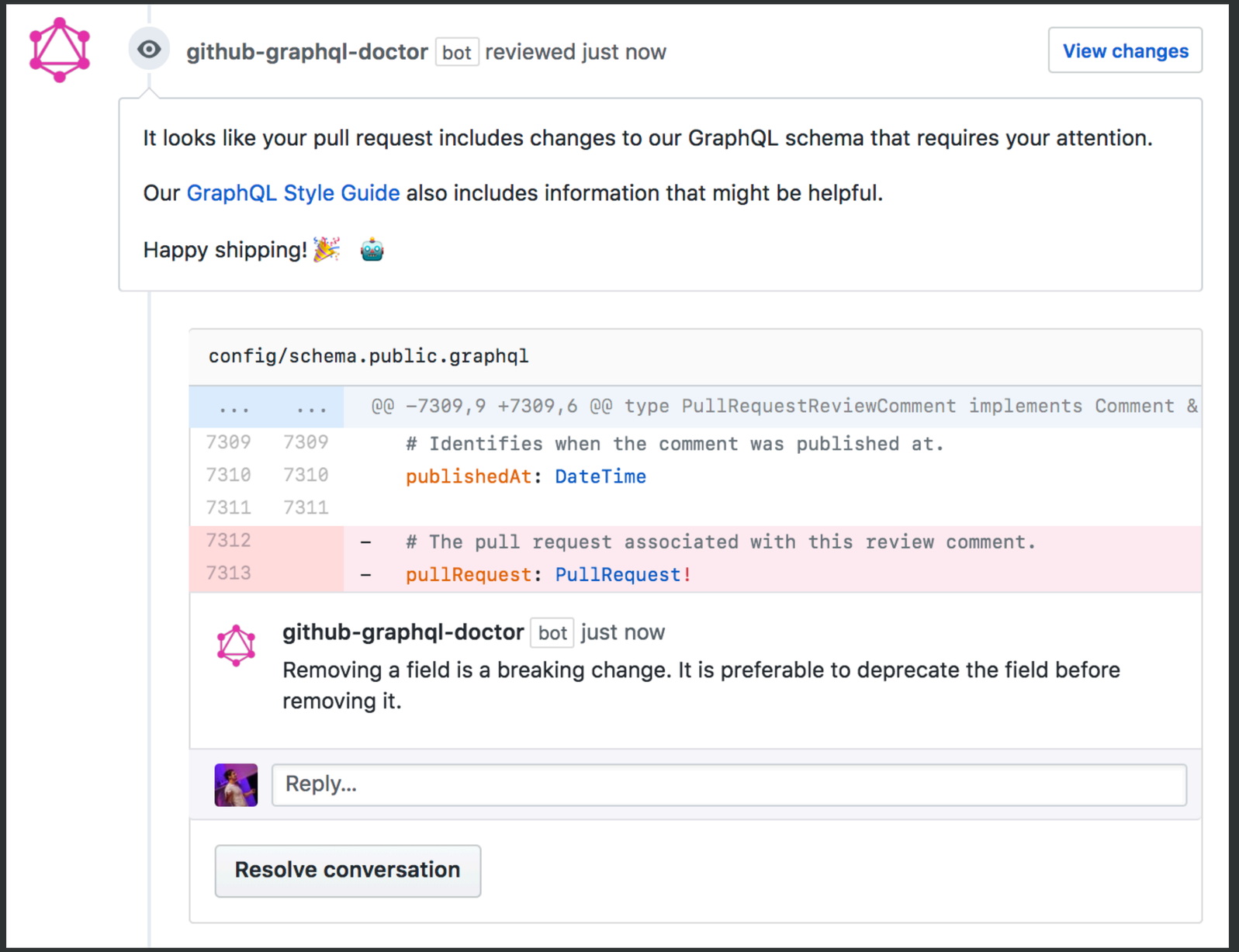

Search or jump to...
Pull requests
Issues
Marketplace
Explore
xuorig / graphql-schema_comparator
Unwatch 6
Unstar 58
Fork 6
Code
Issues 0
Pull requests 0
Projects 0
Wiki
Insights
Settings
Get changes between two GraphQL schemas
Edit
graphql
graphql-ruby
schema-idl
cli
Manage topics
86 commits
1 branch
9 releases
6 contributors
MIT
Branch: master
New pull request
Create new file
Upload files
Find file
Clone or download
xuorig Merge pull request #16 from StanBoyet/patch-1
Latest commit 154286d 12 days ago
bin improved CLI 5 months ago
lib/graphql 0.6.0 2 months ago
test Better test coverage 2 months ago
.gitignore gitignore 5 months ago
.travis.yml Update travis.yml 8 months ago
CHANGELOG.md 0.6.0 2 months ago
CODE_OF_CONDUCT.md First 9 months ago
Gemfile First 9 months ago
LICENSE.txt First 9 months ago
README.md Update Readme.md 19 days ago
Rakefile Add Result tests 9 months ago

COMPARE(SCHEMA_A, SCHEMA_B) => SET<CHANGE>

```
$ bin/testrb test/fast/linting/graphql_breaking_changes_test.rb
Run options: --seed 54244

# Running:

E

Finished in 2.333804s, 0.4285 runs/s, 0.0000 assertions/s.

  1) Error:
GraphQLBreakingChangesTest#test_no_breaking_changes:
RuntimeError:
🚨🚨🚨  *** BREAKING CHANGES DETECTED! *** 🚨🚨🚨
Making a breaking change to our GraphQL schema can potentially impact all of our API integrators.
Our GraphQL schema is public to the world, and it looks like you've made an irreversable change.
Please make sure to get explicit 👍🏽 from @github/ecosystem-api before continuing.
For more information, read https://graphql-docs.githubapp.com/guides/schema-evolution
* Field `pullRequest` was removed from object type `PullRequestReviewComment`
    /Users/xuorig/github/github/test/fast/linting/graphql_breaking_changes_test.rb:169:in
`test_no_breaking_changes'

1 runs, 0 assertions, 0 failures, 1 errors, 0 skips
```

GraphQL Doctor

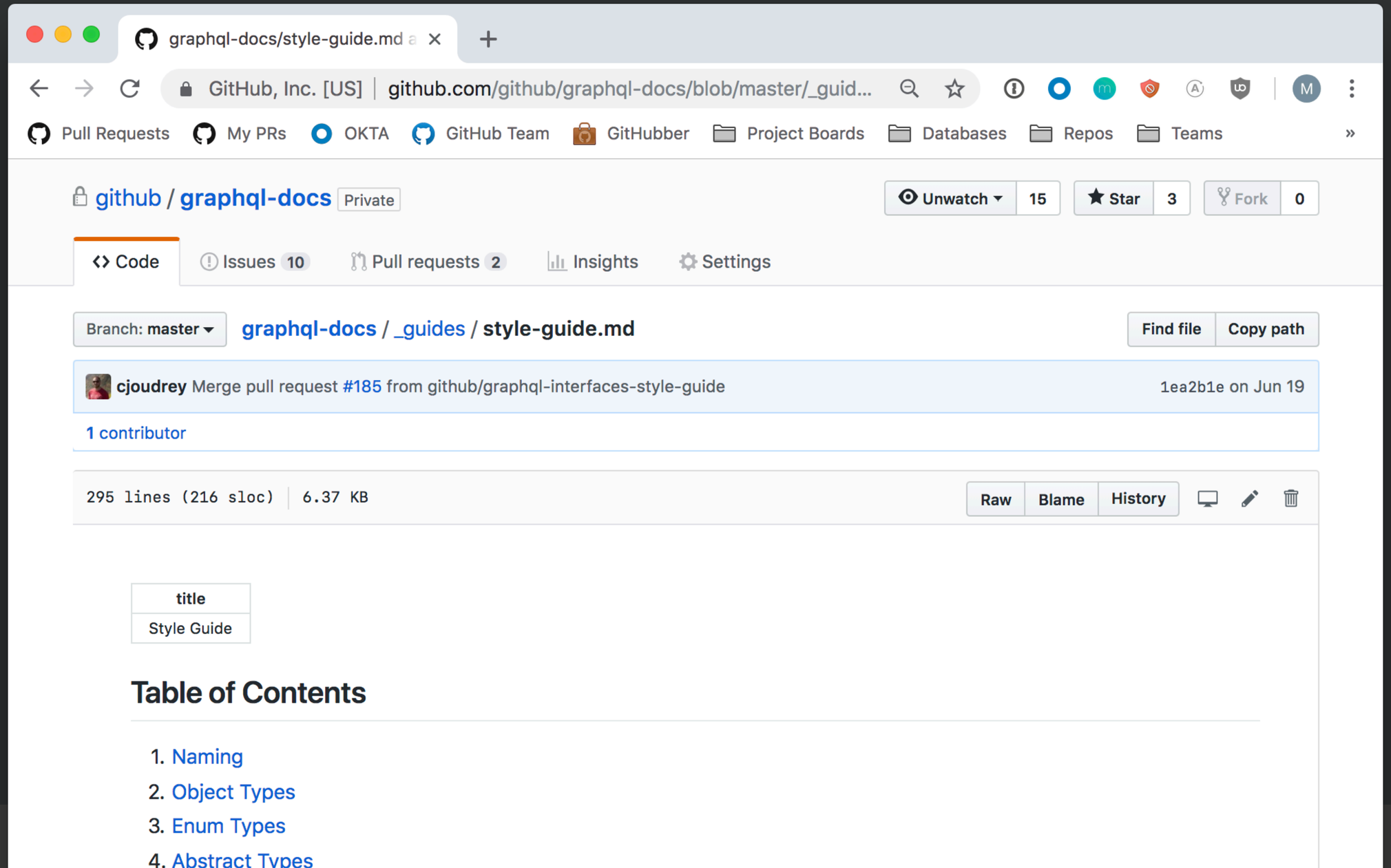


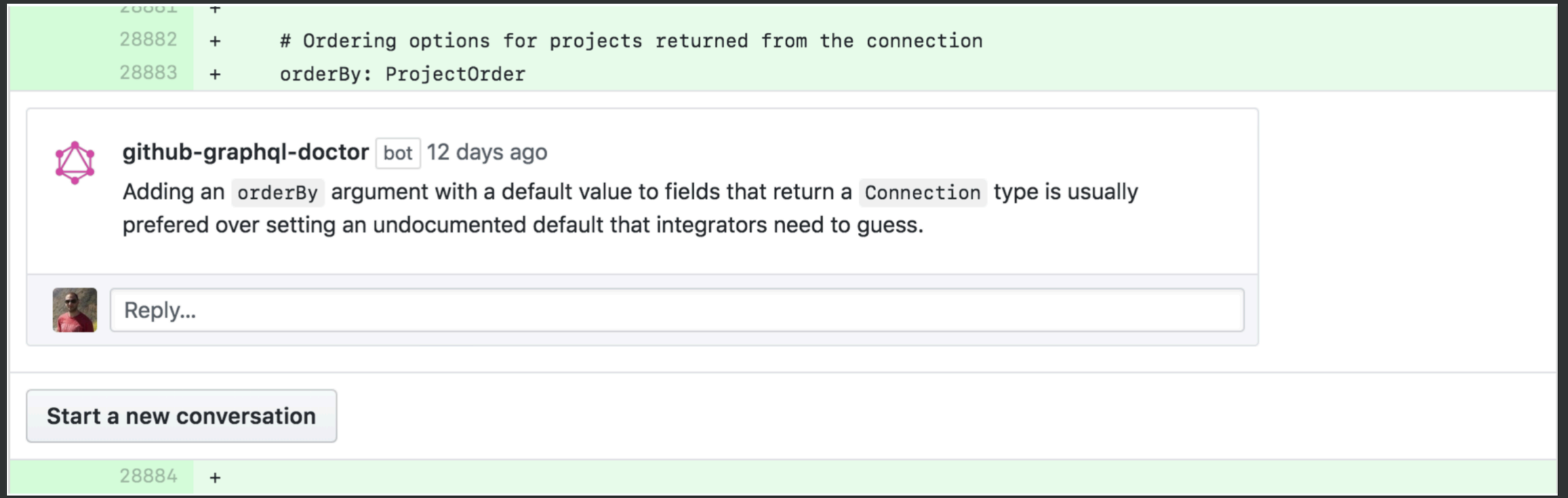

github-graphql-doctor bot reviewed just now
View changes
It looks like your pull request includes changes to our GraphQL schema that requires your attention.
Our GraphQL Style Guide also includes information that might be helpful.
Happy shipping!
config/schema.public.graphql
... ... @@ -7309,9 +7309,6 @@ type PullRequestReviewComment implements Comment &
7309 7309 # Identifies when the comment was published at.
7310 7310 publishedAt: DateTime
7311 7311
7312 - # The pull request associated with this review comment.
7313 - pullRequest: PullRequest!
github-graphql-doctor bot just now
Removing a field is a breaking change. It is preferable to deprecate the field before removing it.
Reply...
Resolve conversation

Have-to Breaking Changes

Prevention First!

Evolvable Schema Design


graphql-docs/style-guide.md
GitHub, Inc. [US] | github.com/github/graphql-docs/blob/master/_guid...
Pull Requests
My PRs
OKTA
GitHub Team
GitHubber
Project Boards
Databases
Repos
Teams
github / graphql-docs
Private
Unwatch 15
Star 3
Fork 0
Code
Issues 10
Pull requests 2
Insights
Settings
Branch: master
graphql-docs / _guides / style-guide.md
Find file
Copy path
cjoudrey Merge pull request #185 from github/graphql-interfaces-style-guide
1ea2b1e on Jun 19
1 contributor
295 lines (216 sloc) | 6.37 KB
Raw
Blame
History
title
Style Guide
Table of Contents
1. Naming
2. Object Types
3. Enum Types
4. Abstract Types


28882 + # Ordering options for projects returned from the connection
28883 + orderBy: ProjectOrder
github-graphql-doctor bot 12 days ago
Adding an orderBy argument with a default value to fields that return a Connection type is usually prefered over setting an undocumented default that integrators need to guess.
Reply...
Start a new conversation
28884 +

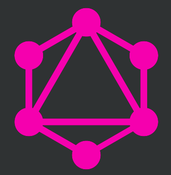
Common Design Issues

Scalar => Object Type

Null => Non-Null 😡

http://bit.ly/evolvable-schema-design

Make it easy for engineers to evolve our schemas

What GraphQL gives us 

3.13.3 @deprecated

```
directive @deprecated(
  reason: String = "No longer supported"
) on FIELD_DEFINITION | ENUM_VALUE
```

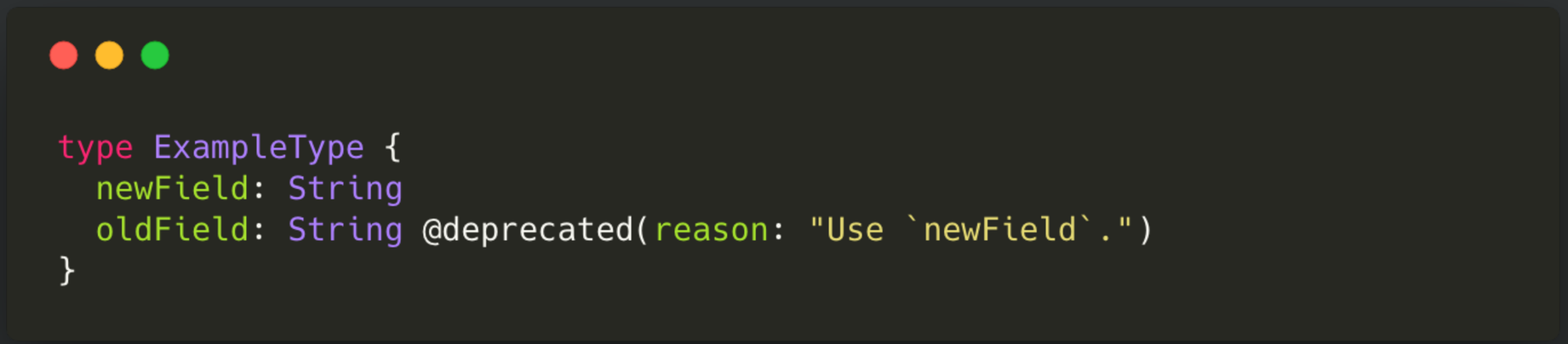
The `@deprecated` directive is used within the type system definition language to indicate deprecated portions of a GraphQL service's schema, such as deprecated fields on a type or deprecated enum values.

Deprecations include a reason for why it is deprecated, which is formatted using Markdown syntax (as specified by CommonMark).

In this example type definition, `oldField` is deprecated in favor of using `newField`.

Example № 85

```
type ExampleType {
  newField: String
  oldField: String @deprecated(reason: "Use `newField`.")
}
```


type ExampleType {
newField: String
oldField: String @deprecated(reason: "Use `newField`.")
}

Problem: free form text, hard to be consistent.

```
type ExampleType {
  newField: String
  oldField: String @deprecated(reason: "Use `newField`.")
}
```

Ruby API

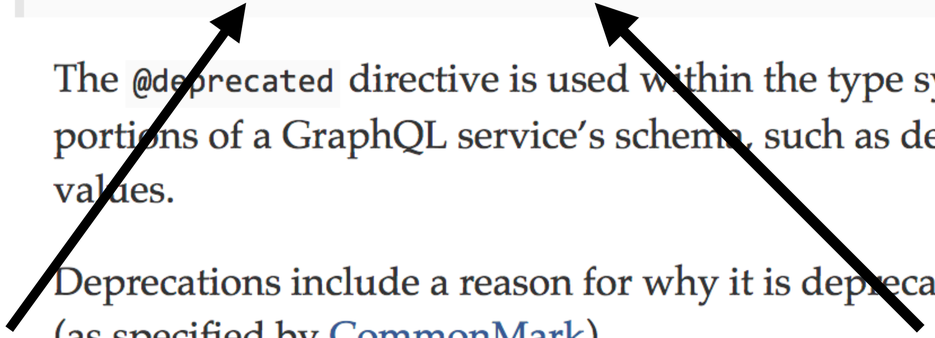
```ruby
deprecated(
  start_date: Date.new(2018, 1, 15),
  reason: "We do not use databaseID anymore.",
  superseded_by: "Use `Node.id` instead."
)
```

Ruby API: Resulting IDL

```
type Repository {
  databaseId: ID! @deprecated(reason: "`databaseID` will be removed. Use `Node.id` instead. Removal on 2017-07-01 UTC.")
}
```

3.13.3 @deprecated

```
directive @deprecated(
  reason: String = "No longer supported"
) on FIELD_DEFINITION | ENUM_VALUE
```



The `@deprecated` directive is used within the type system definition language to indicate deprecated portions of a GraphQL service's schema, such as deprecated fields on a type or deprecated enum values.

Deprecations include a reason for why it is deprecated, which is formatted using Markdown syntax (as specified by CommonMark).

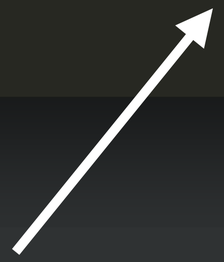
In this example type definition, `oldField` is deprecated in favor of using `newField`.

Example № 85

```
type ExampleType {
  newField: String
  oldField: String @deprecated(reason: "Use `newField`.")
}
```

Another Problem: Only fields and enum values...

```
type Repository {
  issues(states: [IssueState!] @deprecated(reason: "invalid"): [Issue!]!
}
```



Invalid according to spec

Fake It Till You Make It ™

```
type Repository {
  issues(
    # Deprecated: `states` will be removed. Use `newStates` instead. Removal on 2018-07-01 UTC.")
    states: [IssueState!]
  ): [Issue!]!
}
```

Making the change


3 config/schema.public.graphql
View
@@ -7278,9 +7278,6 @@ type PullRequestReviewComment implements Comment & Deletable & Node & Reactable
# Identifies when the comment was published at.
publishedAt: DateTime
- # The pull request associated with this review comment.
- pullRequest: PullRequest!
-
The pull request review associated with this review comment.
pullRequestReview: PullRequestReview

Who is using this field?

Has this field been used in past x days?

```
query {
  viewer {
    pullRequests {
      title
      description
    }
  }
```

GitHub
API


<query>
GitHub
API
kafka

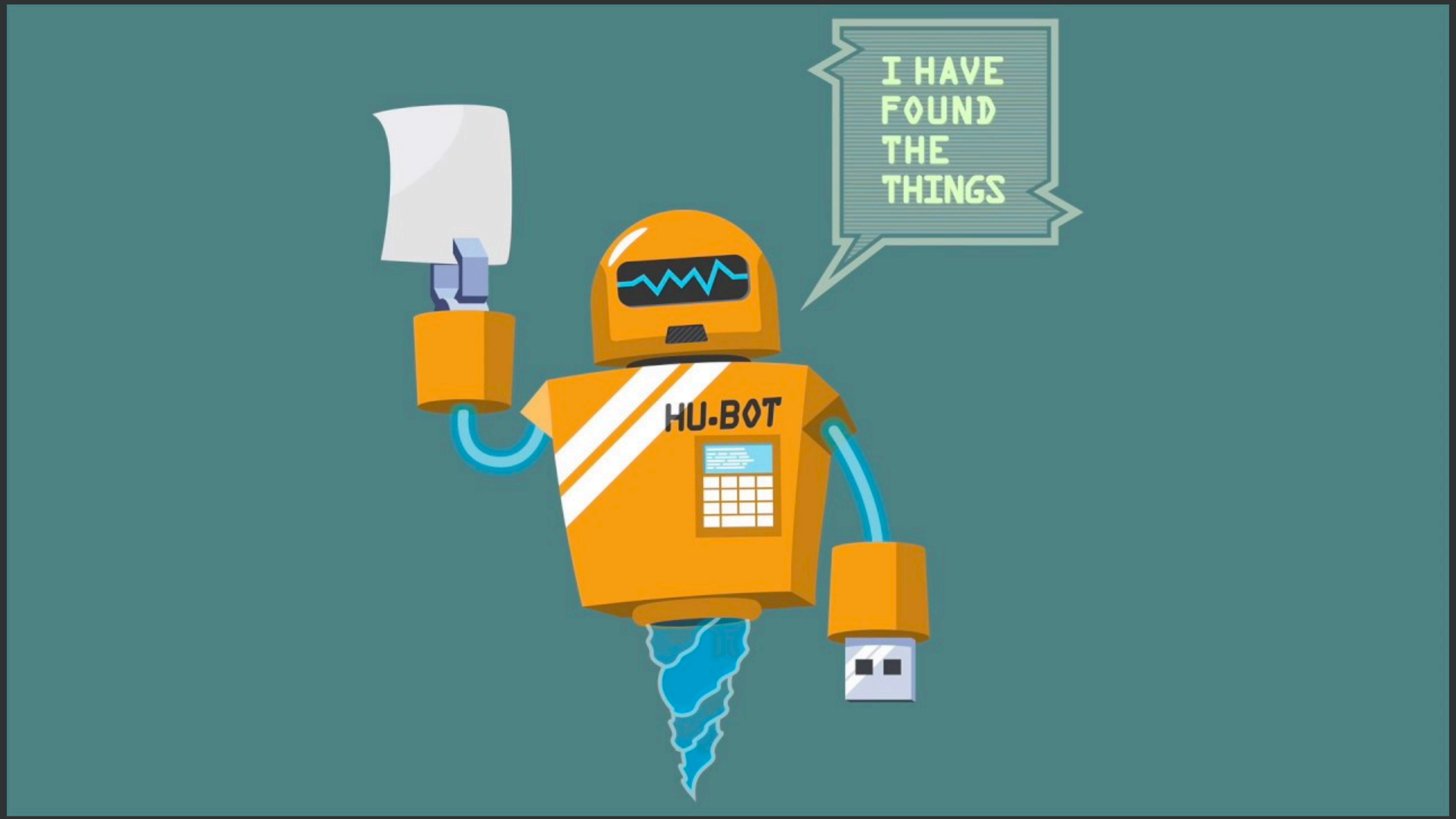
Query
Analytics

```
query {
  viewer {
    pullRequests {
      title
      description
    }
  }
```

Query
Analytics

Query Analytics

```
Type: Query
Field: viewer
App: xxx

Type: User
Field: pullRequests
App: xxx
```


I HAVE FOUND THE THINGS
HU-BOT

xuorig 9:02 PM

.graphql usage Repository.name --schema public --since 5

hubot APP 9:02 PM

One moment while I crunch these numbers for you @xuorig. I'll ping you when the results are ready!

I am running 1 queries for you:

service.github.net/analytics_queries/195

new messages

Your results are ready @xuorig! GraphQL Analytics Query #195, with schema `PUBLIC_SCHEMA` since `2018-09-18 01:02:33 UTC`:

```
+------------+------+---------+
| type       | name | Count   |
+------------+------+---------+
| Repository | name | 123456  |
+------------+------+---------+
```

Message @hubot

graphql-doctor-dev bot reviewed 7 days ago

View changes

It looks like your pull request includes some risky changes to our GraphQL schema.

I've commented on those lines to let you know the impact of those changes. If you have questions, please ping `@github/graphql-api-code-review`.

Our GraphQL Style Guide also includes information that might be helpful.

Happy shipping! 🎉 🤖

schema.graphql

```
...  ...  @@ -14,7 +14,7 @@ type Query {
14   14    type Repository implements Node {
15   15      homepageUri: URI
16   16      id: ID!
17        -  nameWithOwner: String!
```

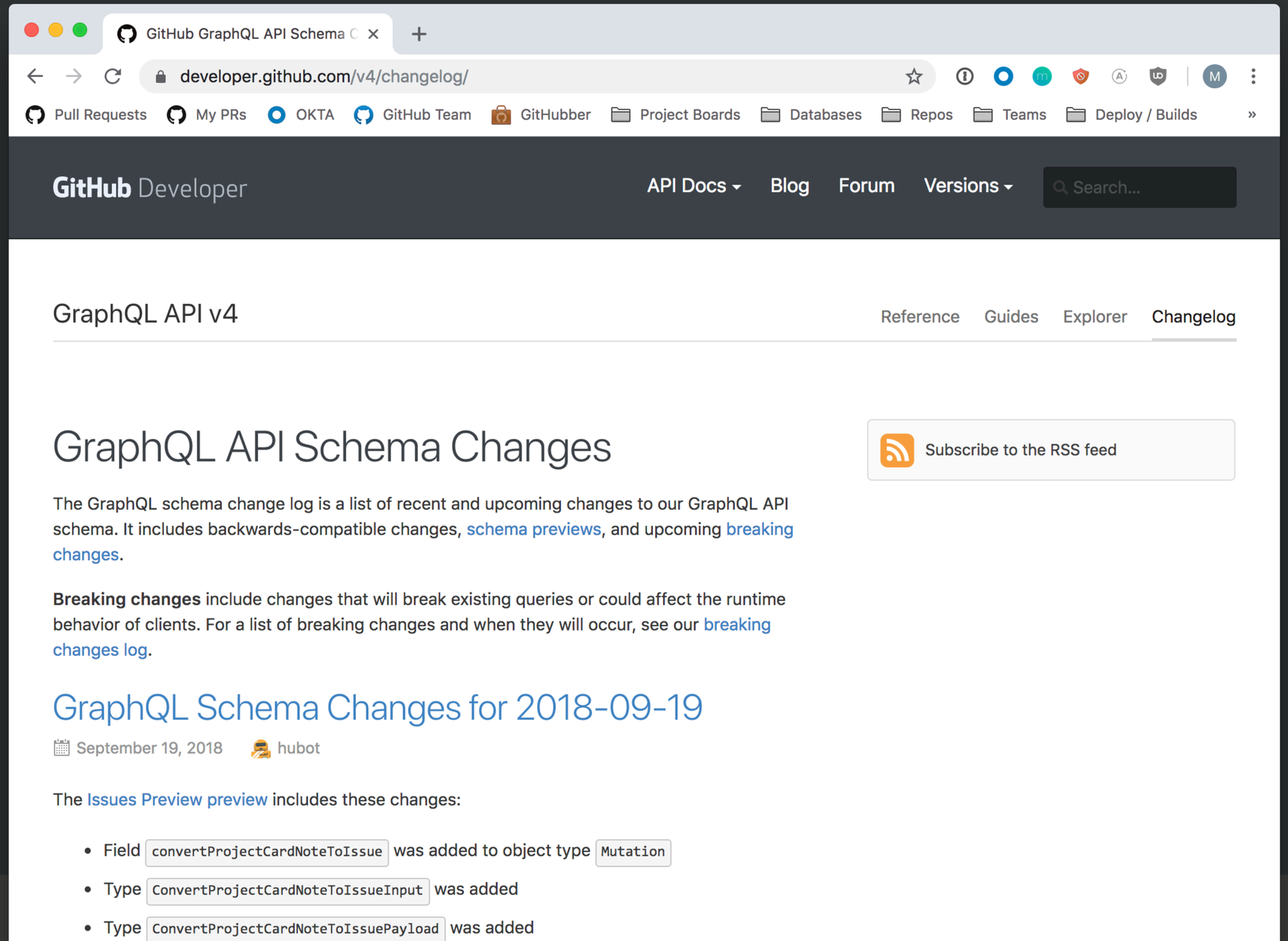
graphql-doctor-dev bot 7 days ago

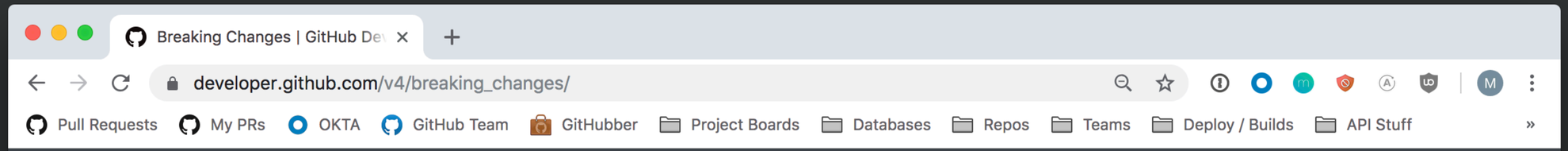
Removing a field is a breaking change. It is preferable to deprecate the field before removing it.

The `Repository.nameWithOwner` field has been used `606` times in the past day.

Reply...

Make it easy for users to evolve their clients.


GitHub GraphQL API Schema C
developer.github.com/v4/changelog/
Pull Requests
My PRs
OKTA
GitHub Team
GitHubber
Project Boards
Databases
Repos
Teams
Deploy / Builds
GitHub Developer
API Docs
Blog
Forum
Versions
Search...
GraphQL API v4
Reference
Guides
Explorer
Changelog
GraphQL API Schema Changes
Subscribe to the RSS feed
The GraphQL schema change log is a list of recent and upcoming changes to our GraphQL API schema. It includes backwards-compatible changes, schema previews, and upcoming breaking changes.
Breaking changes include changes that will break existing queries or could affect the runtime behavior of clients. For a list of breaking changes and when they will occur, see our breaking changes log.
GraphQL Schema Changes for 2018-09-19
September 19, 2018
hubot
The Issues Preview preview includes these changes:
Field convertProjectCardNoteToIssue was added to object type Mutation
Type ConvertProjectCardNoteToIssueInput was added
Type ConvertProjectCardNoteToIssuePayload was added



Breaking Changes

i. Changes scheduled for 2019-01-01

Breaking changes are any changes that might require action from our integrators. We divide these changes into two categories:

- **Breaking:** Changes that will break existing queries to the GraphQL API. For example, removing a field would be a breaking change.
- **Dangerous:** Changes that won't break existing queries but could affect the runtime behavior of clients. Adding an enum value is an example of a dangerous change.

We strive to provide stable APIs for our integrators. When a new feature is still evolving, we release it behind a schema preview.

We'll announce upcoming breaking changes at least three months before making changes to the GraphQL schema, to give integrators time to make the necessary adjustments. Changes go into effect on the first day of a quarter (January 1st, April 1st, July 1st, or October 1st). For example, if we announce a change on January 15th, it will be made on July 1st.

Changes scheduled for 2019-01-01

- **Breaking** A change will be made to AcceptTopicSuggestionPayload.topic.

  Description: Type for `topic` will change from `Topic!` to `Topic`.

  Reason: In preparation for an upcoming change to the way we report mutation errors, non-nullable payload fields are becoming nullable.

- **Breaking** A change will be made to AddCommentPayload.commentEdge.

Schema Previews

i. Deployments
ii. Checks
iii. Team discussions
iv. Branch Protection Rules
v. Protected Branch: Required Signatures
vi. Hovercards
vii. Protected Branch: Multiple Required Approving Reviews
viii. MergeInfoPreview - More detailed information about a pull request's merge state.
ix. Access to a Repositories Dependency Graph
x. Repository Vulnerability Alerts
xi. Temporary Cloning Token for Private Repositories
xii. Resolvable Threads Preview
xiii. Project Event Details
xiv. Issues Preview

Schema previews let you try out new features and changes to our GraphQL schema before they become part of the official GitHub API.

Schema usage data: personalised deprecation warnings!

Learnings & Next Steps 🧠

Deprecating things is hard.

Built-In Deprecations: A double-edged sword ⚔️

How can we make API evolution even better?

Automatic Sunsets?

Smart(er) Clients?

Auto Upgrades?

REST & GraphQL

Migrate? Maintain alongside?

GraphQL-Backed REST API

Referencing REST from GraphQL

Referencing GraphQL from REST

Implementation of GraphQL Resolvers

Your turn: Let's solve some of your design problems!