



Data Fetching with GraphQL and ActionCable

Robert Mosolgo

MAGIC

The Gathering®

Card



- has a name
- has 0+ colors

Deck

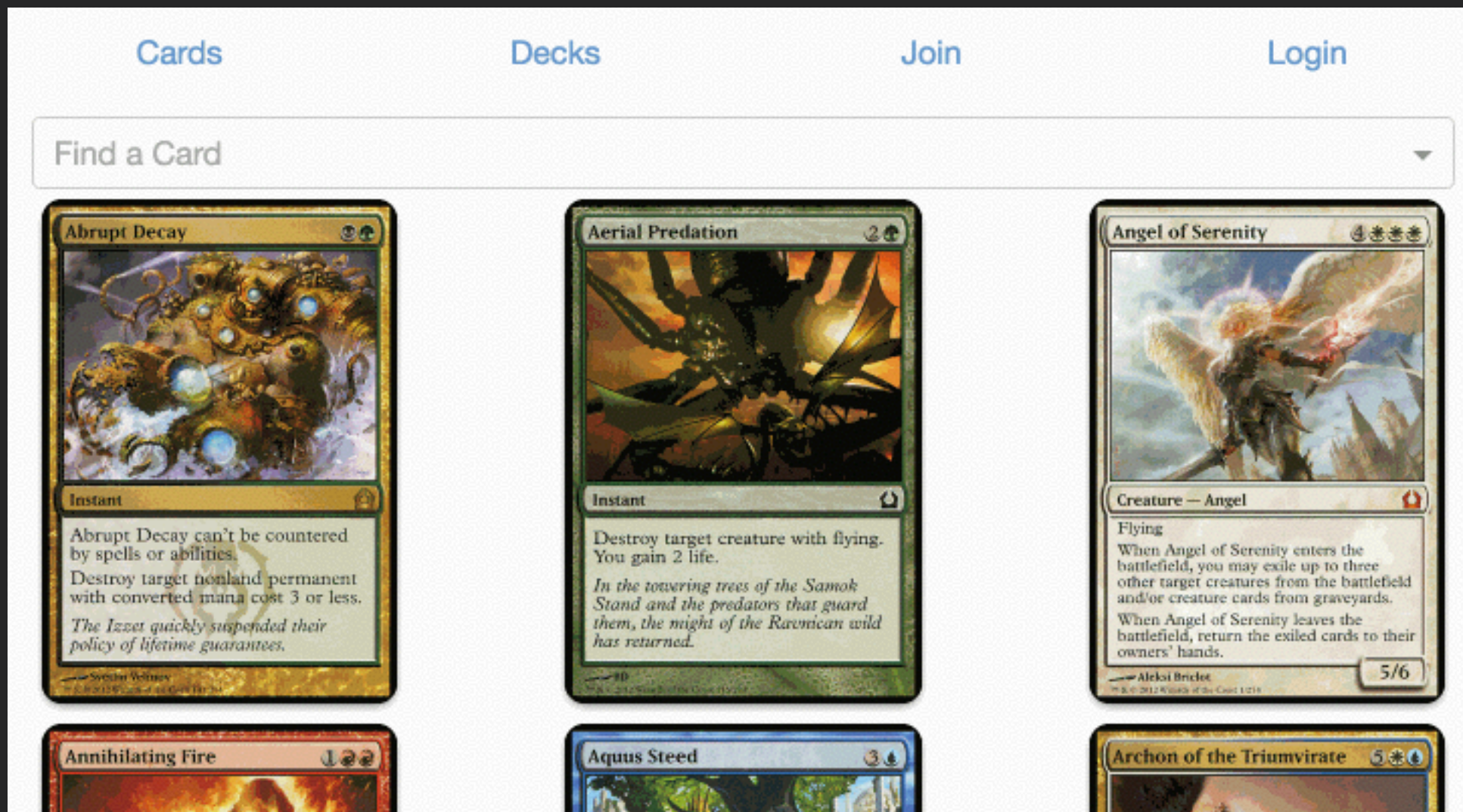


- has many Cards
- has a name

Data-Driven UI

- ➔ Render
- ➔ User input
- ➔ Load data
- ➔ Render again
- ➔ User input again ...

Data-Driven UI



“Canonical Representation”

/api/v1/somethings/1.json

```
def as_json(options = {})
  options[:methods] ||= []
  options[:methods] +=
[:average_rating, :published_at_ago_in_words]

  json = super(options)

  if options[:published_at_format]
    json[:published_at] =
published_at.strftime(options[:published_at_format])
  end

  json
end
```

“Canonical Representation”

`/api/v1/somethings/1.json`

- `#as_json` becomes complex
- Coupled to database
- Overfetching
- Underfetching

GraphQL

POST /graphql

Request

Response

```
{
  deck(id: 1) {
    name
    cards {
      name
      colors
    }
  }
}
```

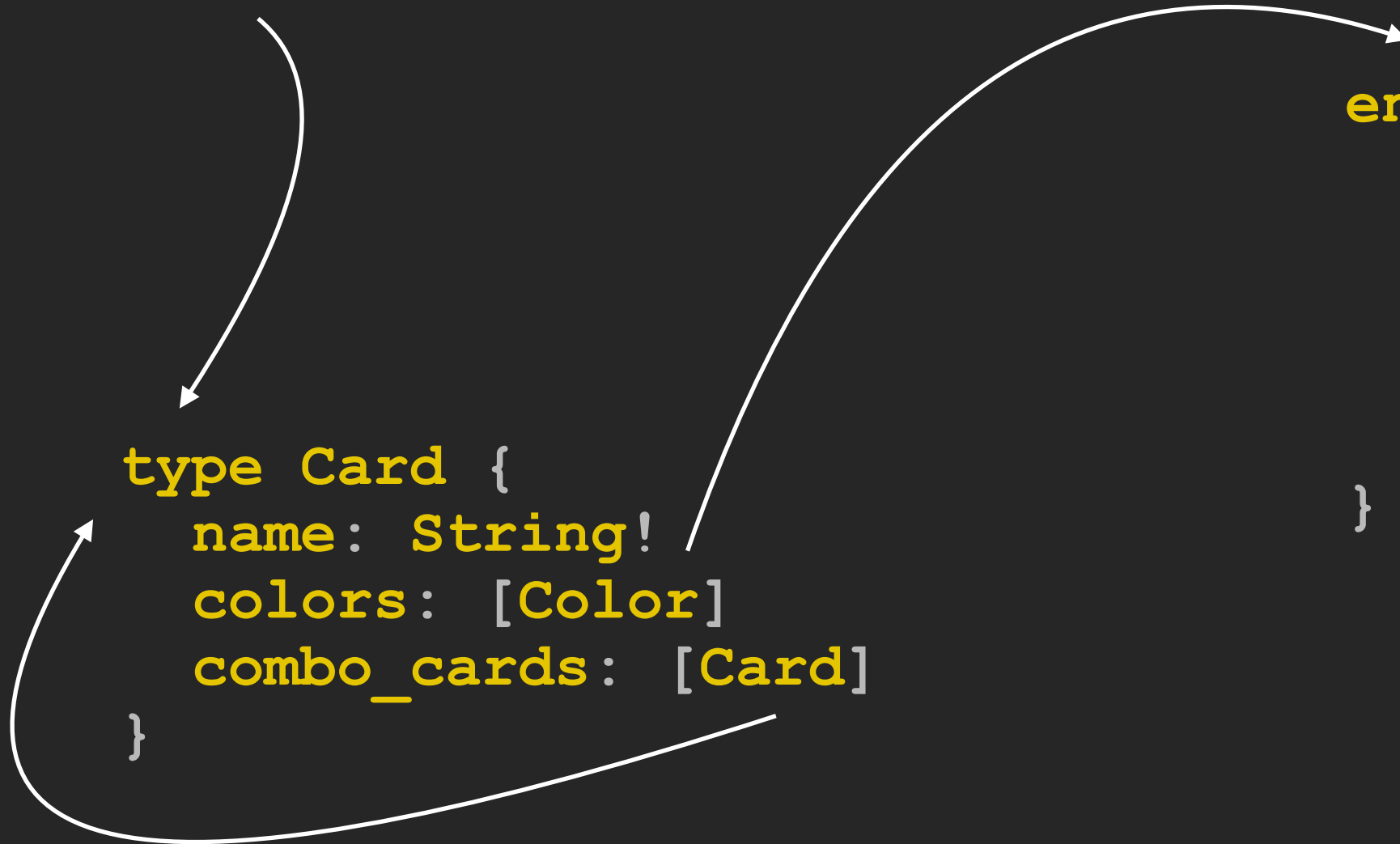
```
{
  "data": {
    "deck": {
      "name": "Turbo-Fog",
      "cards": [
        {
          "name": "Fog",
          "colors": ["GREEN"]
        },
        {
          "name": "Supreme Verdict",
          "colors": ["BLUE", "WHITE"]
        }
      ]
    }
  }
}
```

GraphQL Schema

```
type Deck {  
  name: String!  
  average_rating: Int!  
  cards: [Card]  
}
```

```
type Card {  
  name: String!  
  colors: [Color]  
  combo_cards: [Card]  
}
```

```
enum Color {  
  WHITE  
  BLUE  
  BLACK  
  RED  
  GREEN  
}
```



GraphiQL

The screenshot shows the GraphiQL web interface in a browser. The address bar shows `localhost:3000/graphiql`. The interface is divided into three main sections: a query editor on the left, a JSON response viewer in the middle, and a search results panel on the right.

Query Editor: The query is `query findCards($searchTerm: String!) { cards(search_term: $searchTerm) { name colors } }`. A dropdown menu is open for the `colors` field, showing options: `colors`, `expansions`, `name`, `slug`, `text` (highlighted), and `String The Oracle rules text`. Below the query, a variable is defined: `{ "searchTerm": "Ajani" }`.

JSON Response: The response is a JSON object with a `data` field containing an array of card objects. The visible part of the response is:

```
{
  "data": {
    "cards": [
      {
        "name": "Ajani Goldmane",
        "colors": [
          "WHITE"
        ]
      },
      {
        "name": "Ajani Steadfast",
        "colors": [
          "WHITE"
        ]
      },
      {
        "name": "Ajani Vengeant",
        "colors": [
          "WHITE"
        ]
      }
    ]
  }
}
```

Search Results: The panel shows the title "The colors of Magic, which may apply to decks" and a section titled "VALUES" containing a list of colors: WHITE, BLUE, BLACK, RED, and GREEN.

GraphiQL

```
cards(searchTerm: "Ajani") {  
  name  
  color
```

✖ Cannot query field "color" on type "Card".

```
name  
colors  
... on Nonsense {  
  stuff
```

✖ Unknown type "Nonsense".

```
query find($searchTerm: String!) {  
  cards(searchTerm: "Aja") {
```

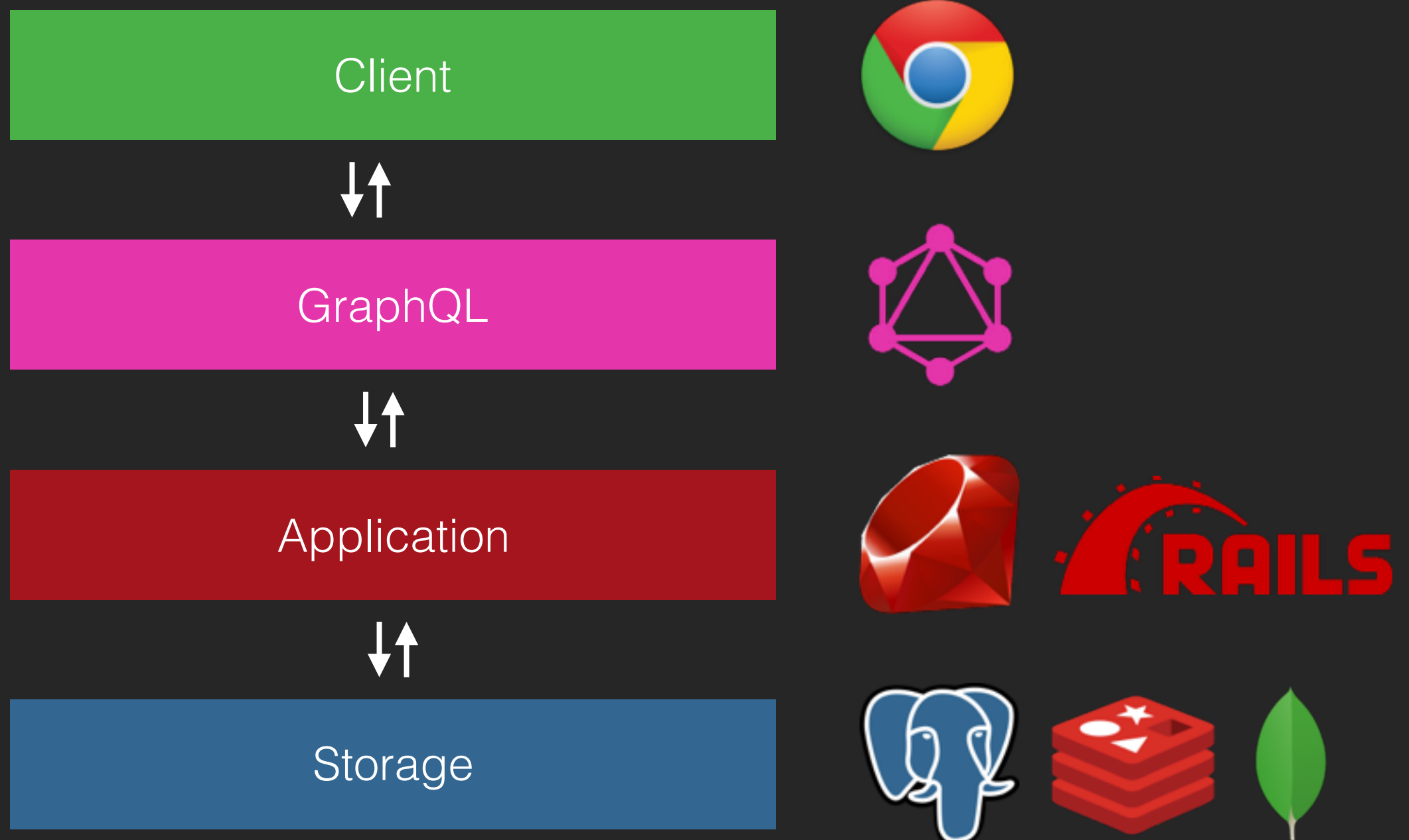
✖ Variable "\$searchTerm" is never used in operation "find".

Introspection

```
1 {  
2   __schema {  
3     types {  
4       name  
5       fields {  
6         name  
7         type {  
8           name  
9         }  
10      }  
11    }  
12  }  
13 }
```

```
{  
  "data": {  
    "__schema": {  
      "types": [  
        {  
          "name": "Query",  
          "fields": [  
            {  
              "name": "cards",  
              "type": {  
                "name": "List"  
              }  
            }  
          ]  
        },  
        {  
          "name": "Card",  
          "fields": [  
            {  
              "name": "colors",  
              "type": {  
                "name": "List"  
              }  
            }  
          ]  
        }  
      ],  
    }  
  }  
}
```

GraphQL “Layer”



GraphQL Recap

- GraphQL query → JSON response
- Schema: typed, self-documenting
- External-facing layer above business logic

GraphQL & Ruby

```
gem "graphql"
```

- ***Types*** expose objects
- ***Fields*** link types & values
- ***Schema*** evaluates queries

Types

```
DeckType = GraphQL::ObjectType.define do
  name "Deck"
  description "A group of magic cards"
  field :name, !types.String
  field :average_rating, !types.Float
  field :cards, types[CardType]
end
```

```
ColorEnum = GraphQL::EnumType.define do
  name "Color"
  description "Colors of Magic"
  value "WHITE"
  value "BLUE"
  value "BLACK"
  value "RED"
  value "GREEN"
end
```

Other types: Interface, Union, Input, Scalar

Fields

Return types

```
# Calls the `name` method  
field :name, types.String
```

```
# This value cannot be `nil`:  
field :name, !types.String
```

```
# This is an array of strings:  
field :previous_names, types[types.String]
```

```
# Returns a Card object  
field :commander, CardType
```

Fields

Custom resolve

```
# Custom `resolve` behavior
field :top_card, !CardType do
  description "Most popular card in the deck"
  resolve -> (obj, args, ctx) {
    obj # => #<Deck>
      .cards
      .order("popularity DESC")
      .first
  }
end
```

Fields

Arguments

```
# `argument` definitions
field :cards, types[CardType] do
  argument :min_rating, types.Int
  resolve -> (obj, args, ctx) {
    min_rating = args[:min_rating] || 0
    # ...
  }
end
```

Schema

Entry points

```
# "root" type -- entry point to the graph
QueryType = GraphQL::ObjectType.define do
  name "Query"

  # {
  #   deck(id: 1) {
  #     # ...
  #   }
  # }
  field :deck, DeckType do
    argument :id, !types.Int
    resolve -> (obj, args, ctx) {
      Deck.find(args[:id])
    }
  end
end
```

Schema

```
Schema = GraphQL::Schema.define do
  query QueryType

  max_complexity 100
  rescue_from(RecordNotFound) { |err| ... }
  # ...
end
```

Schema

Executing queries

```
class QueriesController < ApplicationController
  def create
    query_string = params[:query]
    response = Schema.execute(query_string)
    render json: response
  end
end
```

```
$.post(
  "/queries",
  {query: queryString},
  function(response) { /* ... */ }
)
```

```
gem "graphql-rails"
```

GraphQL & Ruby Recap

- Defining types & fields
- Defining schema
- Executing queries

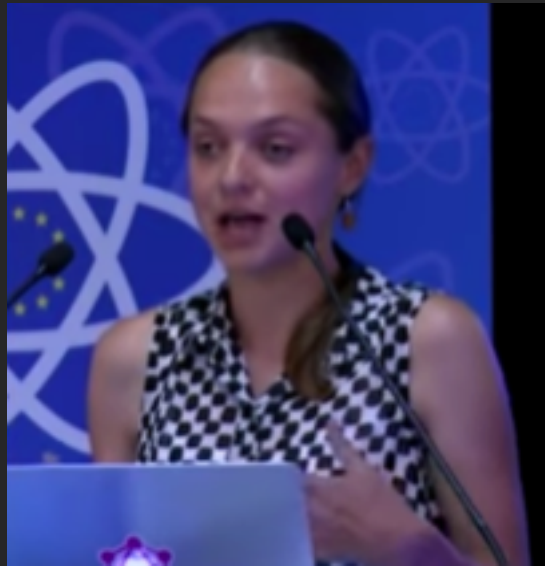
Catching on?



stripe

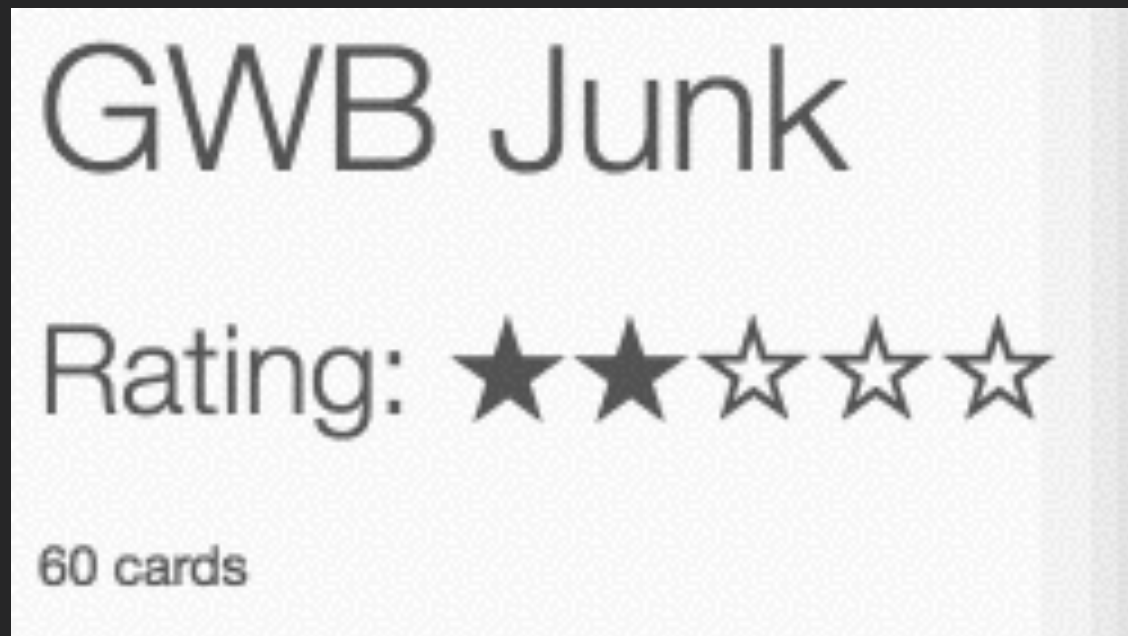


Ideas from Facebook



gem "graphql-streaming"

Live Updates



Subscriptions

```
subscription {  
  new_rating(deckId: 1) {  
    deck {  
      average_rating  
    }  
  }  
}
```

Initial Response:

```
{  
  "data" => {  
    "new_rating" => {  
      "deck" => {  
        "average_rating" => 3.1  
      }  
    }  
  }  
}
```

Push:

```
{  
  "data" => {  
    "new_rating" => {  
      "deck" => {  
        "average_rating" => 3.5  
      }  
    }  
  }  
}
```

ActionCable

Pub-Sub

Listen for changes

```
stream_from "new_rating:#{deck.id}"
```

Publish changes

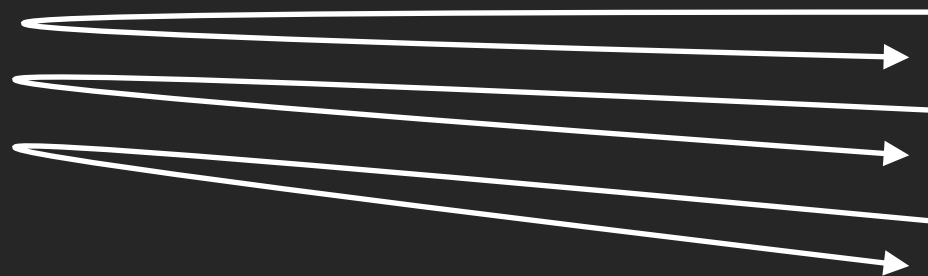
```
ActionCable.server.broadcast("new_rating:#{deck_id}")
```

Channels are updated

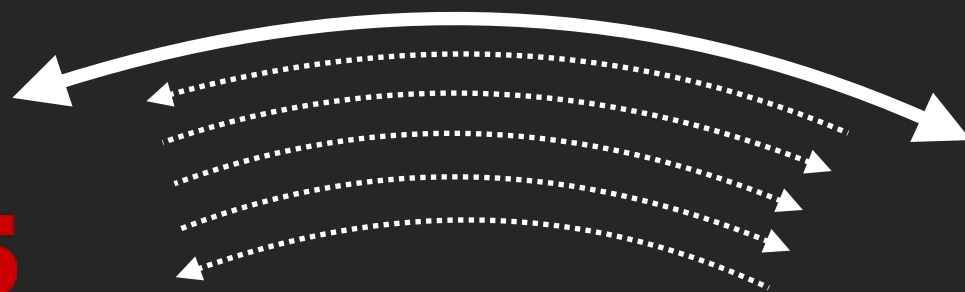
ActionCable

WebSocket Transport

HTTP 1



WebSocket



Long-lived, two-way connection

Subscriptions

```
var queryString = `
    subscription deckRating($deckId: Int!) {
        new_rating(deckId: $deckId) {
            deck {
                average_rating
            }
        }
    }
`

var queryVariables = { deckId: 1 }

var onResponse = function(response) {
    // update your UI with response.data, response.errors
}

App.graphqlChannel.fetch(
    queryString, queryVariables, onResponse
)
```


Live Updates

GWB Junk

Rating: ★★☆☆☆

60 cards

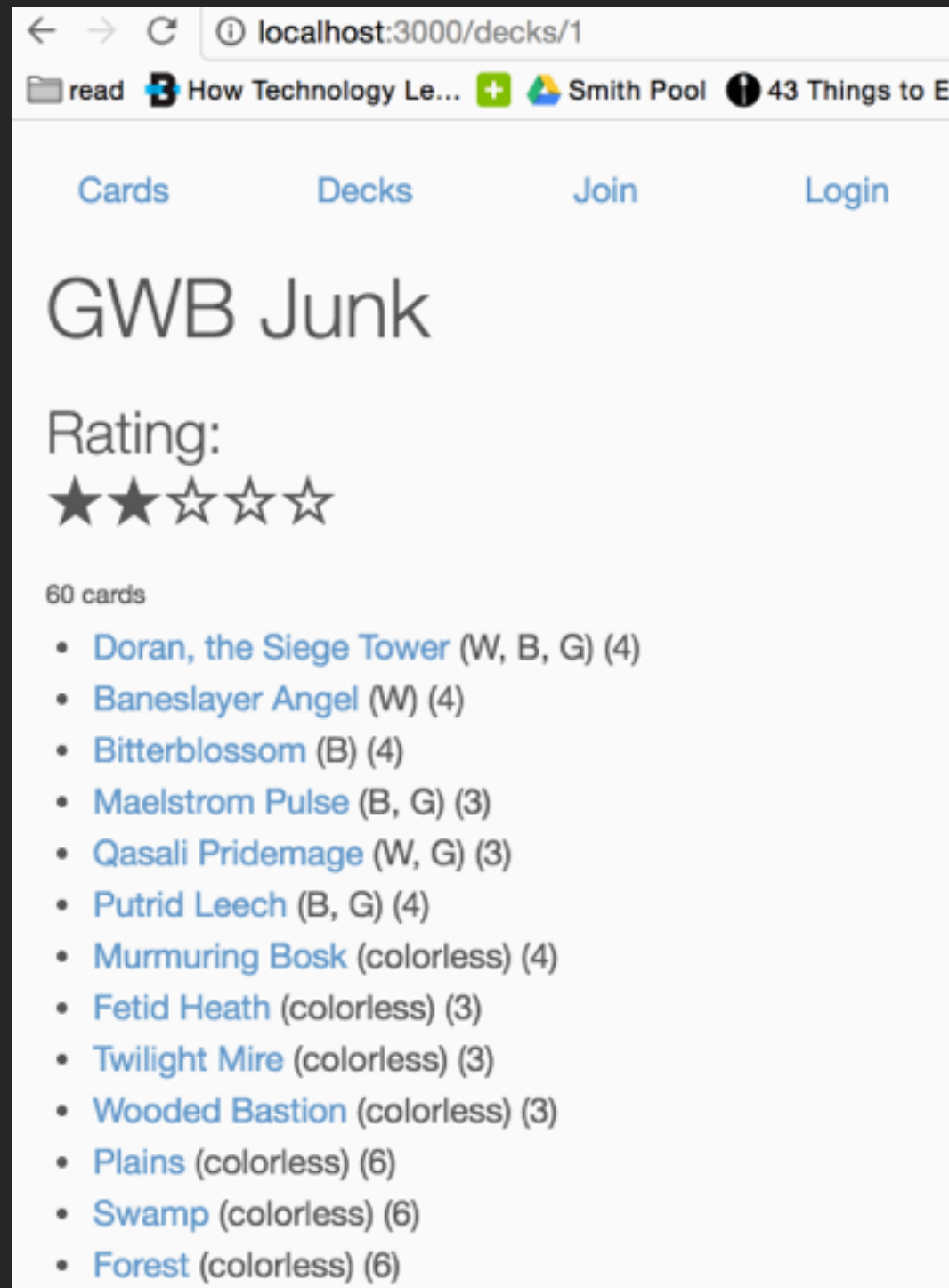
GraphiQL

▶

Prettify

```
1 mutation {  
2   set_rating(rating: 2.3, deck_id: 1)  
3 }
```

Time-to-first-byte



@defer

```
{
  deck(id: 1) {
    name
    win_percentage @defer
    average_rating @defer
  }
}
```

```
  # partial result
  {
    "data" => {
      "deck" => {
        "name" => "Red Deck Wins"
      }
    }
  }

  # patch
  [ "data", "deck" ],
  { "win_percentage" => 0.67 }

  # patch
  [ "data", "deck" ],
  { "average_rating" => 3.1 }
```

@stream

```
{
  deck(id: 1) {
    name
    cards @stream {
      name
    }
  }
}

# partial result
{
  "data" => {
    "deck" => {
      "name" => "Fish"
      "cards" => []
    }
  }
}

# patch
[ "data", "deck", "cards", 0 ]
{ "name" => "Merfolk Sovereign" }

# patch
[ "data", "deck", "cards", 1 ]
{ "name" => "Cursecatcher" }
```

@defer, @stream

```
var queryString = `
  query getDeck($deckId: Int!) {
    deck(id: $deckId) {
      cards @stream { name }
    }
  }
`

var queryVariables = { deckId: 1 }

var onResponse = function(response) {
  // update your UI with:
  // response.data, response.errors
}

App.graphqlChannel.fetch(
  queryString, queryVariables, onResponse
)
```

graphql-streaming to-dos:

- ActionCable Deployment?
- Multiplexing over one channel?
- Complex loading states
- 👍@defer/@stream + Transfer-Encoding=Chunked

Keep Exploring!

- graphql-persisted-queries
- graphql-batch
- view layer
- your idea??



github.com/rmosolgo/graphql-ruby
github.com/rmosolgo/graphql-streaming

@rmosolgo