Designing GraphQL API

Filip Zachar
PrimeHammer

REST vs GraphQL in Rails

REST

- GET, POST, PUT / PATCH, DELETE
- Path & Query parameters
- Routes & Controllers
- JSON response

```
# Processed by UsersController#index
GET /users?columns=id,name,email
{
  users: [
    {
      id: 15,
      name: "John Wick",
      email: "john.wick@dangerous.com"
    },
    ...
  ]

}
```

REST vs GraphQL in Rails

GraphQL

- `gem 'graphql'`
- Queries & Mutations
- Single endpoint `post "/graphql", to: "graphql#execute"`

Controller

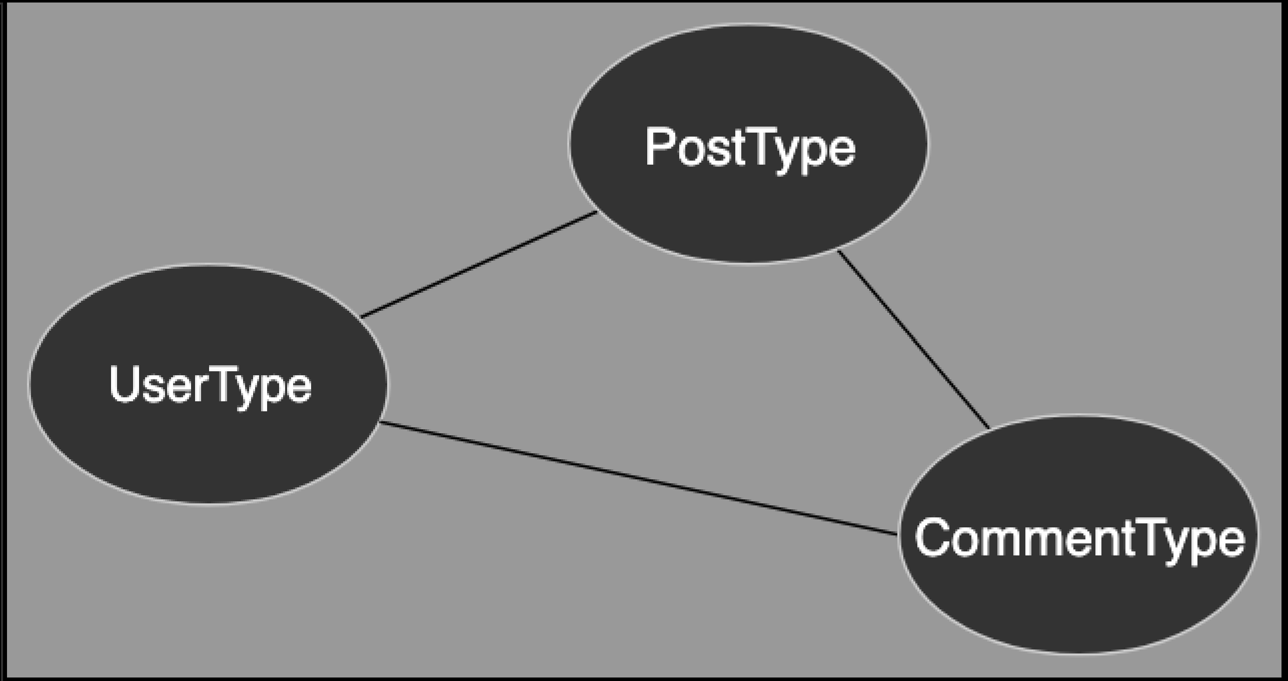
```ruby
class GraphqlController < ApplicationController
  def execute
    result = ExampleAppSchema.execute(
      params[:query],
      variables: params[:variables],
      context: {},
      operation_name: params[:operationName])
    render json: result
  end
  ...
end
```

Types / Queries

Queries / Types

- Each returned object has to be typed
- Types define the shape of the domain graph (the GraphQL Schema)
- Maping of model attributes & methods to type fields

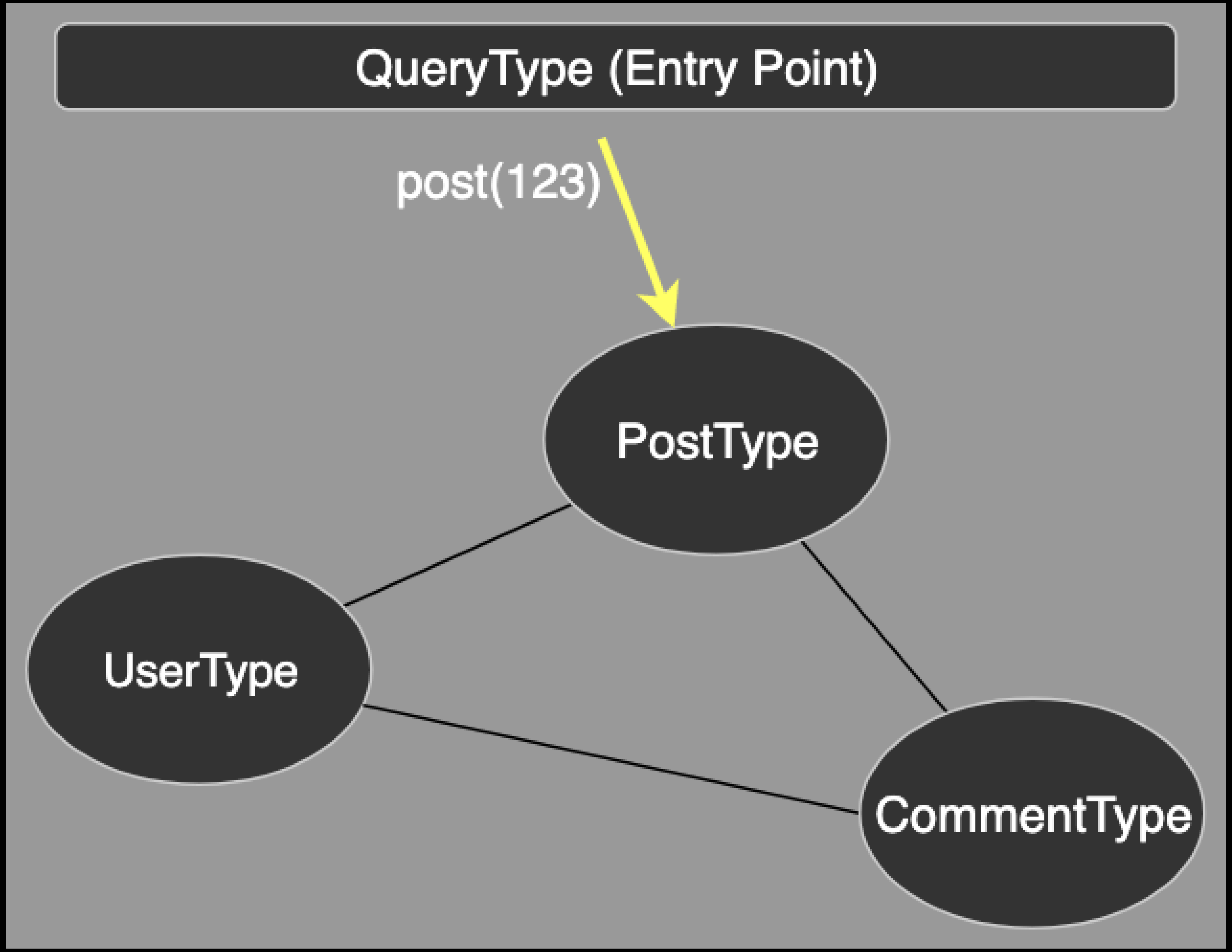
```ruby
module Types
  class UserType
    field :id, ID
    field :name, String
    field :email, String
    field :posts, [Types::PostType]
    field :comments, [Types::CommentType], ...
  end
end
```



Queries / Types

- QueryType is the entry point to the graph
- It defines top level fields

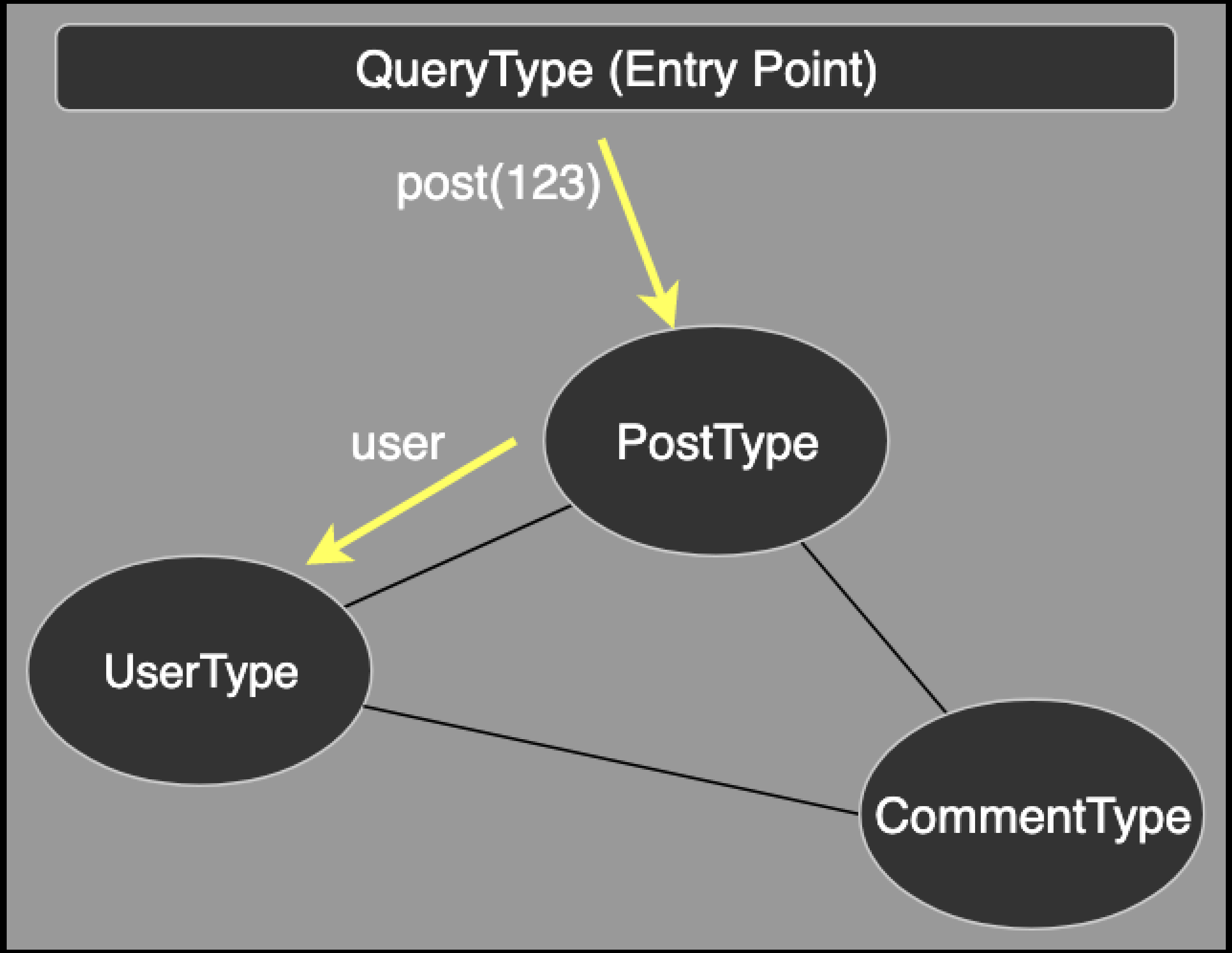
```ruby
module Types
  class QueryType < Types::BaseObject
    field :post, Types::PostType,
      null: true do
      argument :id, ID
    end

    def post(id:)
      Post.find(id)
    end
  end
end
```



Queries / Types

- Traversing the graph from "Entry Point"

```
query {
  post(123) {
    user {
      id
      name
    }
  }
}
```



How to design top level fields?

- Often natural entry point is **current user**
- Calling fields on the **UserType**

```
# query
query {
  currentUser {
    posts(id: 123) {
      name
    }
  }
}
# types/user_type.rb
field :posts, [Types::PostType],
...

# models/user.rb
def posts(id:)
  posts.where(id: id)
end
```

- The **current user** is obtained from the session not query
- We can use graphql context

```
# query
query {
  posts(id: 123) {
    name
  }
}
# graphql_controller.rb
result = GqlSchema.execute(query,
  context: { current_user: user_from_session
}, ...)

# query_type.rb
field :posts, [Types::PostType], ...

def posts(id:)
  context[:current_user]

    .posts.where(id: id)
end
```

Mutations

Queries vs Mutations

- Technically the same
- Different intention
- Both accept arguments
- Both return typed response

Queries

- Obtain data from the server

Mutations

- Manipulate the state on the server
- Obtain the result (of some type)

```
query {
  posts(id: 15) {
    id
    title
    content
  }
}

mutation {
  createPost(
    title: "Ruby Stories",
    content: "..."
  ) {
    id
    title
    content
  }
}
```

Mutations

- **MutationType** fields are top level entry points
- Namespace mutation classes

```ruby
# mutation_type.rb
class Types::MutationType < Types::BaseObject
  field :create_post, mutation: Mutations::Posts::Create
end
# mutations/posts/create.rb
class Mutations::Posts::Create < BaseMutation
  argument :name, String
  field :post, Types::PostType
  field :success, Boolean

  def resolve(name:)
    post = Post.new(name: name)
    result = post.save
    { post: post, success: result }
  end
end
```

Mutations

- Consider having 2 create mutations for 2 different models with similar structure

```
# mutation_type.rb
field :create_post, mutation: Mutations::Posts::Create
field :create_user, mutation: Mutations::Users::Create
# => Duplicate type definition found for name 'CreatePayload'
```

- Each field has a return type
- Mutation return type names are derived from the class name

```
# mutations/posts/create.rb
class Mutations::Posts::Create < BaseMutation
  graphql_name "CreateUserPayload"
  ...
end
```

InputTypes

- (not only) **Mutations** usually accept arguments (e.g. new attributes for model)

```
 # mutations/comments/add_to_post.rb
class Mutations::Comments::AddToPost
  argument :user_id, ID
  argument :content, String
  argument :parent_id, ID
  argument :send_notifications, Boolean
  ...

  def resolve(
    user_id:,
    content:,
    parent_id:,
    send_notifications:)
    # ...
  end
end
```

InputTypes

- Group the attributes using when appropriate
  - Attributes of the same model
  - Differentiate data from control arguments

```ruby
 # mutations/comments/add_to_post.rb
class Mutations::Comments::AddToPost
argument :comment, Input::CommentType
argument :send_notifications, Boolean
...

def resolve(comment:, send_notifications:)
  # ...
end
end

# inputs/comment_type.rb
class Inputs::Comment < GraphQL::Schema::InputObject
argument :user_id, ID
argument :content, String
argument :parent_id, ID
end
```

InputTypes

- Simplifies query definitions in your client

```
 # addCommentToPost.gql
mutation addCommentToPost(
  $userId: ID,
  $content: String,
  $parentId: ID,
  $sendNotifications, Boolean
) {
  addCommentToPost(
    userId: $userId,
    content: $content,
    parentId: $parentId,
    sendNotifications: $sendNotifications,
  ) { ... }
}

# Using input type
  mutation addCommentToPost(

    $comment: CommentInputType
    $sendNotifications, Boolean
  ) {
    addCommentToPost(
      comment: $comment,
      sendNotifications: $sendNotifications,
    ) { ... }
  }
```

Summary

- Get **current user** from context
- Namespace your mutations
- Give `graphql_name` to your mutations when necessary
- Use input types for complex lists of arguments

Resources

- https://graphql-ruby.org
- https://github.com/graphql/graphiql

Questions?