GraphQL

GraphQL is a query language for APIs that was developed by Facebook in 2012 and later open-sourced in 2015. It allows clients to request exactly the data they need, and nothing more, in a single request, rather than making multiple requests to an API with traditional RESTful APIs.

Why use GraphQl instead of restful

- Increased efficiency the client can request all the necessary data in a single query, reducing the number of round-trips to the server and improving performance.
- Flexibility In RESTful APIs, the server determines the structure of the response. With GraphQL, the client specifies the structure of the response, so they can get exactly the data they need and nothing more. This makes it easier to iterate on the client side without changing the API.
- Strong typing GraphQL has a schema that defines the types of data that can be requested, making it easier for developers to understand and work with the API.
- No need for separate API documentation.
- Tooling GraphQL has a rich ecosystem of tools and libraries, such as Apollo and Relay, that make it easy to work with and integrate into existing systems.
- Real-time updates With GraphQL subscriptions, clients can receive real-time updates from the server, making it easier to build real-time applications.

GraphQL also has some cons

- Data Caching: The cache in GraphQL is performed on the client side, but doing so requires adding a lot of extra data or using some external libraries.
- Complexity: The learning curve is more extended than REST because you need to learn a language, types, and how to make requests.
- Performance issues: Sometimes, you can abuse nested attributes on requests, and if your server is not prepared for it, you could encounter some problems.
- File uploading: GraphQL doesn't support this out of the box, so if you want to use images, you will have to use Base64 encoding or a library like Apollo.
- Status Code 200 for everything: You only get a Status 200 (Ok) for all your requests, even if they fail; this is not helpful for the developer experience.

GraphQL key concepts

- Schema
    - GraphQL uses SDL (Schema Definition Language) to define the types of data that can be queried from an API. Types can include objects, enums, interfaces, and unions.
    - GraphQL APIs are organized in terms of types and fields, not endpoints.

```
Unset
type User {
  id: ID!
  name: String!
  email: String!
  posts: [Post!]!
}

type Post {
  id: ID!
  title: String!
  content: String!
  author: User!
}
```

- Queries
    - A query is a request for data from an API using GraphQL. Queries are written in a JSON-like syntax and can be nested to request data from related objects.

```
Unset
type Query {
  users: [User!]!
  user(id: ID!): User
  posts: [Post!]!
  post(id: ID!): Post
}
```

- Mutations
    - Mutations are used to modify data in an API. They are similar to queries, but they use the "mutation" keyword and specify the changes to be made.

Unset

```
type Mutation {
  createUser(name: String!, email: String!): User!
  updateUser(id: ID!, name: String, email: String): User!
  deleteUser(id: ID!): User!
  createPost(title: String!, content: String!, authorId:
ID!): Post!
  updatePost(id: ID!, title: String, content: String): Post!
  deletePost(id: ID!): Post!
}
```

- Subscriptions
  - Subscriptions allow clients to receive real-time updates from an API. They are written like queries, but use the "subscription" keyword and return a stream of data.
- Resolver functions.
  - In GraphQL, resolver functions are responsible for fetching the data for a field in a query or mutation. Each field in a GraphQL schema can have a resolver function that determines how the data is retrieved from the server or another data source.
  - Resolver functions are defined on the server and are called by the GraphQL engine when a query or mutation is executed.
- Directives
  - In GraphQL, directives are a feature that allows you to provide additional instructions to the GraphQL server about how to execute a query or mutation. Directives are used to modify the behavior of a GraphQL query or mutation based on conditions that are not known until runtime. and can be used to include or exclude fields, paginate results, and more.
  - GraphQL defines two built-in directives: @include and @skip.

Unset

```
query GetPosts($published: Boolean!) {
  posts {
    title
    content
    author {
      name
```

```
      email @include(if: $published)
    }
  }
}
```

Integrate GraphQL with rails

- Setup
  - Add gems:
    - bundle add graphql
    - bundle add "graphiql-rails" -> IDE for graphql queries
  - Generate files:
    - rails generate graphql:install
      - This will generate graphql files and add endpoint to the graphl server
- Folder structure

Unset

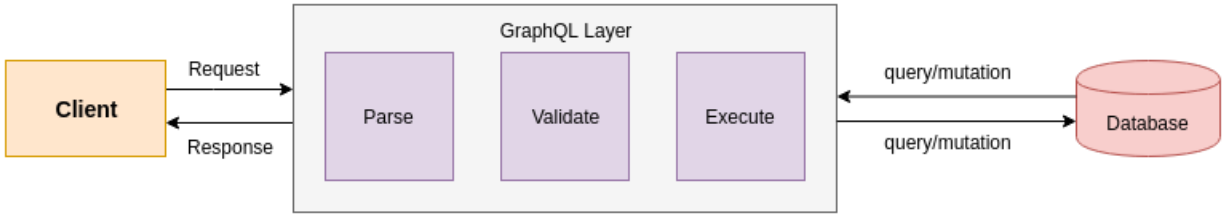
```
app/
├── controllers/
│   ├── graphql_controller.rb
├── graphql/
│   ├── mutations/
│   │   ├── base_mutation.rb
│   │   └── ...
│   ├── queries/
│   │   ├── base_query.rb
│   │   └── ...
│   ├── types/
│   │   ├── base_object.rb
│   │   ├── query_type.rb
│   │   ├── mutation_type.rb
│   │   └── ...
│   ├── resolvers/
│   │   ├── base_resolver.rb
```

```
|   |   ├── user_resolver.rb
|   |   └── ...
|   └── schema.rb
├── models/
├── views/
└── ...
```

    - app/controllers/graphql_controller: The entry point of graphql server contains the execute function and you might add authentication or authorization logic to the execute method to secure your GraphQL API.
    - app/graphql/mutations: This directory contains mutations for your GraphQL schema. Each mutation should inherit from Mutations::BaseMutation and define a resolve method that handles the mutation's behavior.
    - app/graphql/queries: This directory contains queries for your GraphQL schema. Each query should inherit from Queries::BaseQuery and define a resolve method that handles the query's behavior.
    - app/graphql/types: This directory contains the GraphQL types for your schema. Each type should inherit from Types::BaseObject and define the fields that can be queried for that type. You'll also typically define your schema's QueryType and MutationType in this directory.
    - app/graphql/resolvers: This directory contains resolver classes that handle fetching data for your schema's fields. Each resolver should inherit from Resolvers::BaseResolver and define a resolve method that returns the field's data.
    - app/graphql/schema.rb: This file defines your GraphQL schema using the GraphQL::Schema class. It typically imports your schema's QueryType and MutationType from the Types directory and sets them as the schema's query and mutation properties.

- Request response Cycle
    - A client sends a GraphQL query to the server. [**Request processing**]
    - The query is received by the GraphqlController and passed to the execute action.
    - The execute method parses the query string into an AST using GraphQL::Language::Parser. [**Parsing**]
    - The parsed AST is validated against the schema using GraphQL::Schema::Validator. [**Validation**]

- The execute method executes the query against the schema using YourAppSchema.execute, passing in the parsed AST, any variables, and any context.
- The execute method traverses the AST and calls the appropriate resolver functions to retrieve the data for each requested field. [**Execution**]
- The data is serialized into the requested format (such as JSON) and returned as the response by the GraphqlController. [**Serialization**]



GraphQL Query Analysis

GraphQL query analysis is the process of analyzing a GraphQL query to understand its structure, fields, and complexity. This analysis can help you optimize the performance of your GraphQL server by identifying queries that are too complex or fetch too much data.
There are several tools and techniques available for analyzing GraphQL queries, including

- GraphQL query parsers: These are tools that parse GraphQL queries into an abstract syntax tree (AST), which can be analyzed programmatically.
- GraphQL schema introspection: GraphQL schemas can be introspected to provide information about the types, fields, and operations available in the schema.
- Query complexity analysis: Query complexity analysis tools can be used to determine the complexity of a GraphQL query based on the number of fields, depth of nesting, and other factors.
- Execution tracing: Execution tracing can be used to analyze the performance of a GraphQL query by tracing the time taken to execute each resolver function.

https://graphql-ruby.org/queries/complexity_and_depth#prevent-deeply-nested-queries
https://graphql-ruby.org/queries/ast_analysis.html#using-analyzers

Errors Handling

GraphQL always responds with 200 OK, if the action has not caused the server-side error. Two types of errors usually happen user input-related errors (validations) and exceptions.

- Validation errors can appear only in mutations and are included in the data sent back. They are meant to provide useful feedback to the user
- Exceptions could happen in any query and signal that something went wrong with the query: for example, authentication/authorization issues, unprocessable input data, etc.

Good To know

- Generators https://graphql-ruby.org/schema/generators
- Alias https://graphql.org/learn/queries/#aliases
- Fragment https://graphql.org/learn/queries/#fragments
- https://graphql.org/learn/

Points need some investigation

- Authentication
- Frontend Integration
- Testing
- Caching
- File uploads

https://github.com/Khaled-Galal-98/graphql-demo demo app
https://sites.google.com/a/eaglerider.com/wiki/e3/rental-management-migration/graphql Wiki docs
https://sites.google.com/a/eaglerider.com/wiki/e3/rental-management-migration/graphql/learngraphql