GraphQL API Cheatsheet

A comprehensive cheatsheet covering GraphQL syntax, queries, mutations, schema definition, and best practices for designing and implementing GraphQL APIs.

GraphQL Basics

Core Concepts

GraphQL: A query language for your API and a server-side runtime for executing queries by using a type system you define for your data.

Schema: The backbone of any GraphQL API. It defines the structure of the data, including the types, fields, and relationships.

Query: Used to request data from the GraphQL API. Queries specify exactly what data the client needs, and nothing more.

Mutation: Used to modify data on the server. Mutations can create, update, or delete data.

Resolver: A function attached to a field in the GraphQL schema. It fetches the data for that field.

GraphQL vs REST

GraphQL	REST
Single endpoint.	Multiple endpoints.
Client specifies the data required.	Server defines the data returned.
Strongly typed schema.	Loosely defined data structures.
Efficient data fetching (no over-fetching or under-fetching).	Potential for over-fetching and under-fetching.

GraphQL Schema Definition Language (SDL)

Defining Types

Use SDL to define the structure and types of your data.

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

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

Scalars: Basic data types like `Int`, `Float`, `String`, `Boolean`, and `ID`.

Non-Null: Use `!` to indicate a field cannot be null.

Lists: Use `[]` to indicate a field is a list of values.

Queries and Mutations in Schema

Define entry points for querying and mutating data.

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

type Mutation {
  createUser(name: String!, email: String):
User
  updatePost(id: ID!, title: String): Post
}
```

Interfaces and Unions

Interface: Defines a set of fields that concrete types must implement.

```
interface Node {
  id: ID!
}

type User implements Node {
  id: ID!
  name: String!
}
```

Union: Defines a set of possible types a field can return.

```
union SearchResult = User | Post

type Query {
  search(term: String!):
[SearchResult]
}
```

GraphQL Queries

Basic Query Structure

A GraphQL query specifies what data to fetch.

```
query {
  user(id: "123") {
    id
    name
    email
    posts {
      title
    }
  }
}
```

The query selects the `user` with `id: "123"` and requests the `id`, `name`, `email`, and `posts` (including their `title`).

Arguments

Pass arguments to fields to filter or modify the results.

```
query {
  posts(limit: 10, orderBy: "createdAt_DESC")
{
    id
    title
    content
  }
}
```

The query fetches the 10 most recently created posts.

Aliases

Use aliases to rename fields in the response, especially when querying the same field with different arguments.

```
query {
  recentPosts: posts(limit: 5) {
    title
  }
  featuredPosts: posts(orderBy: "likes_DESC",
limit: 3) {
    title
  }
}
```

This query fetches both the 5 most recent posts and the 3 most liked posts, each with their own alias.

Fragments

Use fragments to reuse field selections across multiple queries.

```
fragment PostFields on Post {
  id
  title
  content
}

query {
  recentPosts: posts(limit: 5) {
    ...PostFields
  }
  featuredPosts: posts(orderBy: "likes_DESC",
limit: 3) {
    ...PostFields
  }
}
```

The `PostFields` fragment is used in both `recentPosts` and `featuredPosts` queries.

GraphQL Mutations

Basic Mutation Structure

A GraphQL mutation modifies data on the server.

```
mutation {
  createUser(name: "John Doe", email:
"john.doe@example.com") {
    id
    name
    email
  }
}
```

This mutation creates a new user with the provided name and email, and returns the `id`, `name`, and `email` of the newly created user.

Variables

Use variables to make mutations dynamic.

```
mutation CreateUser($name: String!, $email:
String!) {
  createUser(name: $name, email: $email) {
    id
    name
    email
  }
}
```

Variables:

```
{
  "name": "Jane Smith",
  "email": "jane.smith@example.com"
}
```

This mutation uses variables `name` and `email` to create a new user.

Updating and Deleting Data

Mutations can also be used to update and delete data.

```
mutation UpdatePost($id: ID!, $title: String)
{
  updatePost(id: $id, title: $title) {
    id
    title
    content
  }
}

mutation DeletePost($id: ID!) {
  deletePost(id: $id) {
    id
  }
}
```

These mutations update the title of a post and delete a post, respectively.