Customizing and organizing GraphQL operations

Customizing and organizing GraphQL operations

This lesson covers

- Using arguments to customize what a request field returns
- Customizing response property names with aliases
- Describing runtime executions with directives
- Reducing duplicated text with fragments
- Composing queries and separating data requirement responsibilities

Customizing fields with arguments

- The fields in a GraphQL operation are similar to functions, They map input to output.
- A function input is received as a list of argument values.
- Just like functions, we can pass any GraphQL field a list of argument values.
- A GraphQL schema on the backend can access these values and use them to customize the response it returns for that field.

Identifying a single record to return

- Every API request that asks for a single record from a collection needs to specify an identifier for that record.

- This identifier is usually associated with a unique identifier for that record in the server's database, but it can also be anything else that can uniquely identify the record.

Identifying a single record to return

- Here is an example query that asks for information about the user whose email address is jane@doe.name.

```
query UserInfo {
  user(email: "jane@doe.name") {
    firstName
    lastName
    username
  }
}
```

Identifying a single record to return

```
query NodeInfo {
  node(id: "A-GLOBALLY-UNIQUE-ID-HERE") {
    ...on USER {
      firstName
      lastName
      username
      email
    }
  }
}
```

Identifying a single record to return

- Here is an example to read information about the jsComplete organization, which hosts all open source resources for jsComplete.com.

```
query OrgInfo {
  organization(login: "jscomplete") {
    name
    description
    websiteUrl
  }
}
```

Limiting the number of records returned by a list field

GraphiQL ▶ Prettify History Explorer < Docs

```
1  query OrgInfo {
2    organization(login: "jscomplete") {
3      name
4      description
5      websiteUrl
6      repositories {
7        nodes {
8          name
9        }
10     }
11   }
12 }
```

QUERY VARIABLES

```
1  {}
```

```
{
  "data": {
    "organization": null
  },
  "errors": [
    {
      "type": "MISSING_PAGINATION_BOUNDARIES",
      "path": [
        "organization",
        "repositories"
      ],
      "locations": [
        {
          "line": 6,
          "column": 5
        }
      ],
      "message": "You must provide a `first` or `last` value to
properly paginate the `repositories` connection."
    }
  ]
}
```

Limiting the number of records returned by a list field

- Here is the query you can use to retrieve the first 10 repositories under the jsComplete organization

```
query First10Repos {
  organization(login: "jscomplete") {
    name
    description
    websiteUrl
    repositories(first: 10) {
      nodes {
        name
      }
    }
  }
}
```

Ordering records returned by a list field

- Query to retrieve the first 10 repositories when they are ordered alphabetically by name.

```
query orgReposByName {
  organization(login: "jscomplete") {
    repositories(first: 10, orderBy: { field: NAME, direction: ASC }) {
      nodes {
        name
      }
    }
  }
}
```

Ordering records returned by a list field

- Here is one query you can use to do that

```
query OrgPopularRepos {
  organization(login: "jscomplete") {
    repositories(first: 10, orderBy: { field: STARGAZERS, direction: DESC }) {
      nodes {
        name
      }
    }
  }
}
```

Paginating through a list of records

- When you need to retrieve a page of records, in addition to specifying a limit, you need to specify an offset.
- In the GitHub API, you can use the field arguments after and before to offset the results returned by the arguments first and last, respectively.
- To use these arguments, you need to work with node identifiers, which are different than database record identifiers.

Paginating through a list of records

- Here is a query that includes cursor values through the edges field.

```
query OrgRepoConnectionExample {
  organization(login: "jscomplete") {
    repositories(first: 10, orderBy: { field: CREATED_AT, direction: ASC }) {
      edges {
        cursor
        node {
          name
        }
      }
    }
  }
}
```

Paginating through a list of records

```
query OrgRepoConnectionExample2 {
  organization(login: "jscomplete") {
    repositories(
      first: 10,
      after: "Y3Vyc29yOnYyOpK5MjAxNy0wMS0yMVQwODo1NTo0My0wODowMM4Ev4A3",
      orderBy: { field: CREATED_AT, direction: ASC }
    ) {
      edges {
        cursor
        node {
          name
        }
      }
    }
  }
}
```

```
query OrgReposMetaInfoExample {
  organization(login: "jscomplete") {
    repositories(
      first: 10,
      after: "Y3Vyc29yOnYyOpK5MjAxNy0wMS0yMVQwODo1NTo0My0wODowMM4Ev4A3",
      orderBy: { field: STARGAZERS, direction: DESC }
    ) {
      totalCount
      pageInfo {
        hasNextPage
      }
      edges {
        cursor
        node {
          name
        }
      }
    }
  }
}
```

Searching and filtering

- A query that uses a search term within the projects relation to return the Twitter Bootstrap projects that start with v4.1.

```
query SearchExample {
  repository(owner: "twbs", name: "bootstrap") {
    projects(search: "v4.1", first: 10) {

    nodes {
      name
    }
  }
}
}
```

Searching and filtering

- To list only the repositories that you own, you can use the affiliations field argument.

```
query FilterExample {
  viewer {
    repositories(first: 10, affiliations: OWNER) {
      totalCount
      nodes {
        name
      }
    }
  }
}
```

Providing input for mutations

```
mutation StarARepo {
  addStar(input: { starrableId: "MDEwOlJlcG9zaXRvcnkxMjU2ODEwMDY=" }) {
    starrable {
      stargazers {
        totalCount
      }
    }
  }
}
```

Renaming fields with aliases

- Let's say you are developing the profile page in GitHub.
- Here is a query to retrieve partial profile information for a GitHub user

```
query ProfileInfo {
  user(login: "samerbuna") {
    name
    company
    bio
  }
}
```

You get a simple user object in the response

GraphQL API Explorer | GitHub

jsComplete

Secure https://developer.github.com/v4/explorer/

GraphiQL ▶ Prettify History Docs

```
query ProfileInfo {
  user(login: "samerbuna") {
    name
    company
    bio
  }
}
```

```
{
  "data": {
    "user": {
      "name": "Samer Buna",
      "company": "jsComplete.com",
      "bio": "Author for Pluralsight, LinkedIn, Manning, and others - Curator of jsComplete.com"
    }
  }
}
```

QUERY VARIABLES

Renaming fields with aliases

- Luckily, in GraphQL, the awesome alias feature lets us declaratively instruct the API server to return fields using different names.

- All you need to do is specify an alias for that field, which you can do using this syntax:

aliasName: fieldName

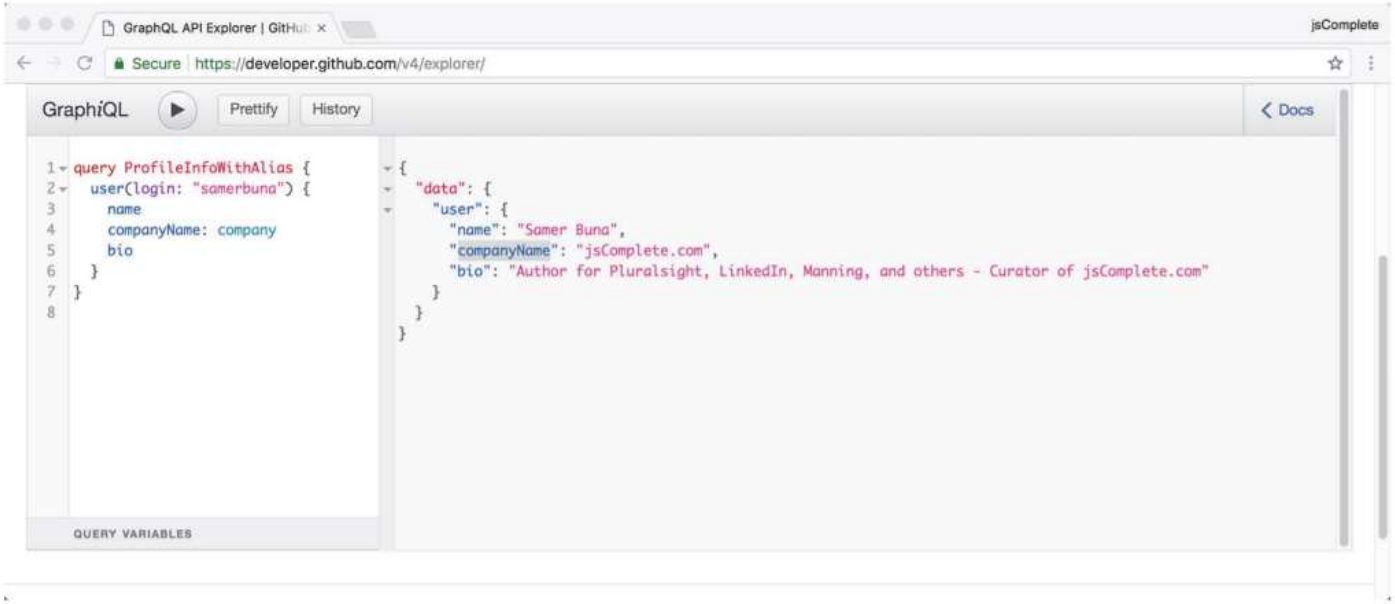
Renaming fields with aliases

- All you need to do is specify a companyName alias.

```
query ProfileInfoWithAlias {
  user(login: "samerbuna") {
    name
    companyName: company
    bio
  }
}
```

Renaming fields with aliases

- Previous code gives a response that is ready for you to plug into the application UI



Customizing responses with directives

- Sometimes, the customization you need on a server response goes beyond the simple renaming of fields.
- You may need to conditionally include (or exclude) branches of data in your responses.
- This is where the directives feature of GraphQL can be helpful.
- A directive in a GraphQL request is a way to provide a GraphQL server with additional information about the execution and type validation behavior of a GraphQL document.

Customizing responses with directives

```
query AllDirectives {
  __schema {
    directives {
      name
      description
      locations
      args {
        name
        description
        defaultValue
      }
    }
  }
}
```

GraphQL API Explorer | GitHub

jsComplete

Secure https://developer.github.com/v4/explorer/

GraphiQL ▶ Prettify History Docs

```
query AllDirectives {
  __schema {
    directives {
      name
      description
      locations
      args {
        name
        description
        defaultValue
      }
    }
  }
}
```

QUERY VARIABLES


```
{
  "data": {
    "__schema": {
      "directives": [
        {
          "name": "include",
          "description": "Directs the executor to include this field or fragment only when the `if` argument is true.",
          "locations": [
            "FIELD",
            "FRAGMENT_SPREAD",
            "INLINE_FRAGMENT"
          ],
          "args": [
            {
              "name": "if",
              "description": "Included when true.",
              "defaultValue": null
            }
          ]
        },
        {
          "name": "skip",
          "description": "Directs the executor to skip this field or fragment when the `if` argument is true.",
          "locations": [
            "FIELD",
            "FRAGMENT_SPREAD",
            "INLINE_FRAGMENT"
          ],
```

Variables and input values

- A variable is simply any name in the GraphQL document that begins with a $ sign: for example, $login or $showRepositories.
- The name after the $ sign can be anything, We use variables to make GraphQL operations generically reusable and avoid having to hardcode values and concatenate strings.
- To use a variable in a GraphQL operation, you first need to define its type, You do that by providing arguments to any named operation.

Variables and input values

GraphiQL ▶ Prettify History Explorer

```
1 query OrgInfo {
2   organization(login: "jscomplete") {
3     name
4     description
5     websiteUrl
6   }
7 }
```

QUERY VARIABLES

```
1 {}
```

```
{
  "data": {
    "organization": {
      "name": "jsComplete",
      "description": "Learn Full-stack JavaScript Development with Node, React, GraphQL, and more.",
      "websiteUrl": "https://jscomplete.com/"
    }
  }
}
```

Schema **organization** ×

Lookup a organization by login.

TYPE

Organization

ARGUMENTS

login: String!

The organization's login.

Variables and input values

- Now we can use the same syntax to define the new variable.
- The type for $orgLogin should match the type of the argument where it is going to be used.
- Here is the OrgInfo query written with this new $orgLogin variable

```
query OrgInfo($orgLogin: String!) {
  organization(login: $orgLogin) {
    name
    description
    websiteUrl
  }
}
```

Variables and input values

- Since we used only one variable, the JSON object for that is

```
{
        "orgLogin": "jscomplete"
}
```

GraphiQL ▶ Prettify History Explorer < Docs

```
1 query OrgInfo($orgLogin: String!) {
2   organization(login: $orgLogin) {
3     name
4     description
5     websiteUrl
6   }
7 }
```

QUERY VARIABLES

```
1 {
2   "orgLogin": "jscomplete"
3 }
```

```
{
  "data": {
    "organization": {
      "name": "jsComplete",
      "description": "Learn Full-stack JavaScript Development with Node, React, GraphQL, and more.",
      "websiteUrl": "https://jscomplete.com/"
    }
  }
}
```

Variables and input values

- For example, the previous query can have the value "jscomplete" as the default value of $orgLogin using this syntax.

```
query OrgInfoWithDefault($orgLogin: String = "jscomplete") {
  organization(login: $orgLogin) {
    name
    description
    websiteUrl
  }
}
```

The @include directive

- The @include directive can be used after fields (or fragments) to provide a condition (using its if argument).
- That condition controls whether the field (or fragment) should be included in the response.

The use of the @include directive looks like this:
fieldName @include(if: $someTest)

The @include directive

- This new $fullDetails variable will be required because we are about to use it with a directive.
- The first line of the OrgInfo query needs to be changed to add the type of $fullDetails:

```
query OrgInfo($orgLogin: String!, $fullDetails: Boolean!)
{
```

The @include directive

- A simple use of the @include directive can do that. The if argument value in this case will be the $fullDetails variable.

Here is the full query.

```
query OrgInfo($orgLogin: String!, $fullDetails: Boolean!) {
  organization(login: $orgLogin) {
    name
    description
    websiteUrl @include(if: $fullDetails)
  }
}
```

GraphiQL ▶ Prettify History Explorer < Docs

```
1 query OrgInfo($orgLogin: String!, $fullDetails: Boolean!) {
2   organization(login: $orgLogin) {
3     name
4     description
5     websiteUrl @include(if: $fullDetails)
6   }
7 }
```

QUERY VARIABLES

```
1 {
2   "orgLogin": "jscomplete",
3   "fullDetails": false
4 }
```

```
{
  "data": {
    "organization": {
      "name": "jsComplete",
      "description": "Learn Full-stack JavaScript Development with
Node, React, GraphQL, and more."
    }
  }
}
```

The @skip directive

- This directive is simply the inverse of the @include directive.
- Just like the @include directive, it can be used after fields (or fragments) to provide a condition (using its if argument).
- The condition controls whether the field (or fragment) should be excluded in the response, The use of the @skip directive looks like this:

fieldName @skip(if: $someTest)

The @skip directive

- Instead of inverting that variable value in the
- JSON values object, we can use the @skip directive to use the $partialDetails value directly.
- The OrgInfo query becomes the following.

```
query OrgInfo($orgLogin: String!, $partialDetails: Boolean!) {
  organization(login: $orgLogin) {
    name
    description
    websiteUrl @skip(if: $partialDetails)
  }
}
```

The @skip directive

- The following query will never include websiteUrl no matter what value you use for $partialDetails.

```
query OrgInfo($orgLogin: String!, $partialDetails: Boolean!) {
  organization(login: $orgLogin) {
    name
    description
    websiteUrl @skip(if: $partialDetails) @include(if: false)
  }
}
```

The @deprecated directive

- When deprecating a field in a GraphQL schema, the @deprecated directive supports a reason argument to provide the reason behind the deprecation.

- The following is the GraphQL's schema language representation of a type that has a deprecated field

```
type User {
  emailAddress: String
  email: String @deprecated(reason: "Use 'emailAddress'.")
}
```

GraphQL fragments

Why fragments?

- To build anything complicated, the truly helpful strategy is to split what needs to be built into smaller parts and then focus on one part at a time.
- Ideally, the smaller parts should be designed in a way that does not couple them with each other.

Defining and using fragments

- For example, let's take the simple GitHub organization information query example:

```
query OrgInfo {
  organization(login: "jscomplete") {
    name
    description
    websiteUrl
  }
}
```

Defining and using fragments

- To make this query use a fragment, you first need to define the fragment.

```
fragment orgFields on Organization {
  name
  description
  websiteUrl
}
```

Defining and using fragments

- To use the fragment, you “spread” its name where the fields were originally used in the query

```
query OrgInfoWithFragment {
  organization(login: "jscomplete") {
    ...orgFields
  }
}
```

Fragments and DRY

```
query MyRepos {
  viewer {
    ownedRepos: repositories(affiliations: OWNER, first: 10) {
      nodes {
        nameWithOwner
        description
        forkCount
      }
    }
    orgsRepos: repositories(affiliations: ORGANIZATION_MEMBER, first: 10) {
      nodes {
        nameWithOwner
        description
        forkCount
      }
    }
  }
}
```

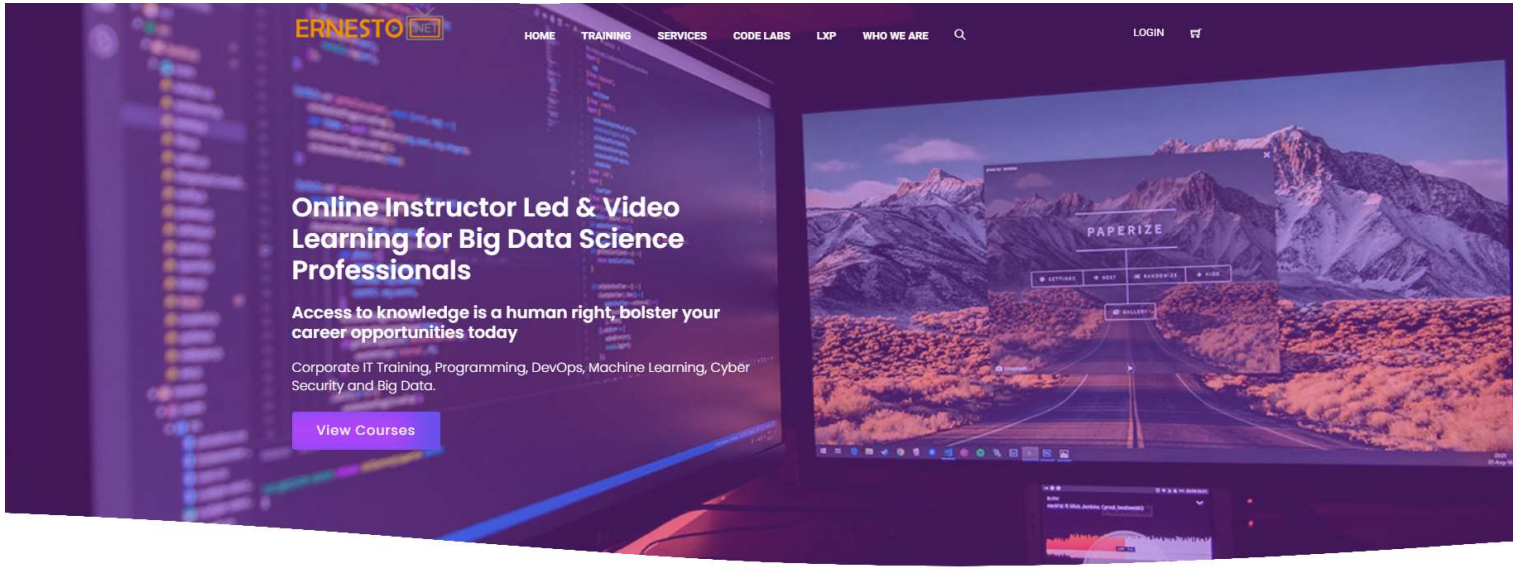
- Here is the same GraphQL operation modified to use a fragment to remove the duplicated parts.

```
query MyRepos {
  viewer {
    ownedRepos: repositories(affiliations: OWNER, first: 10) {
      ...repoInfo
     }
    orgsRepos: repositories(affiliations: ORGANIZATION_MEMBER, first: 10) {
      ...repoInfo
    }
  }
}
fragment repoInfo on RepositoryConnection {
  nodes {
    nameWithOwner
    description
    forkCount
  }
}
```

Fragments and UI components

- The word component can mean different things to different people. In the UI domain, a component can be an abstract input text box or Twitter's full 280-character tweet form with its buttons and counter display.
- You can pick any part of an application and call it a component, Components can be small or big.
- They can be functional on their own, or they can be parts that have to be put together to make something functional.

Fragments and UI components



Do You Have These Concerns with Data Science Courses?

Fragments and UI components

```
const profileData = {
  tweetsCount: ·-·-·,
  profileImageUrl: ·-·-·,
  backgroundImageUrl: ·-·-·,
  name: ·-·-·,
  handle: ·-·-·,
  bio: ·-·-·,
  location: ·-·-·,
  url: ·-·-·,
  createdAt: ·-·-·,
  followingCount: ·-·-·,
  followersCount: ·-·-·,
};
```

- The TweetList component needs a data object that might look like this.

```
const tweetList = [
  { id: ·-·-·,
    name: ·-·-·,
    handle: ·-·-·,
    date: ·-·-·,
    body: ·-·-·,
    repliesCount: ·-·-·,
    tweetsCount: ·-·-·,
    likes: ·-·-·,
  },
  { id: ·-·-·,
    name: ·-·-·,
    handle: ·-·-·,
    date: ·-·-·,
    body: ·-·-·,
    repliesCount: ·-·-·,
    tweetsCount: ·-·-·,
    likesCount: ·-·-·,
  },
  ·-·-·,
];
```

Fragments and UI components

```
fragment headerData on User {
  tweetsCount
  profileImageUrl
  backgroundImageUrl
  name
  handle
  bio
  location
  url
  createdAt
  followingCount
  followersCount
}
```

Fragments and UI components

- The data required by the Sidebar component can be declared using this fragment

```
fragment sidebarData on User {
  SuggestedFollowing {
    profileImageUrl
  }
  media {
    mediaUrl
  }
}
```

Fragments and UI components

- The data required by a single Tweet component can be declared as follows

```
fragment tweetData on Tweet {
  user {
    name
    handle
  }
  createdAt
  body
  repliesCount
  retweetsCount
  likesCount
}
```

Fragments and UI components

- We can use the tweetData fragment here

```
fragment tweetListData on TweetList {
  tweets: {
    ...tweetData
  }
}
```

Fragments and UI components

- To come up with the data required by the entire page, all we need to do is put these fragments together and form one GraphQL query using fragment spreads.

```
query ProfilePageData {
  user(handle: " “Ernesto.net”      {
    ...headerData
    ...sidebarData
    ...tweetListData
  }
}
```

Fragments and UI components

```
fragment tweetData on Tweet {
  user {
    name
    handle
  }
  createdAt
  body
  repliesCount
  retweetsCount
  likesCount
  viewsCount
}
```

- Here is an inline fragment use case from the GitHub API

```
query InlineFragmentExample {
  repository(owner: "facebook", name: "graphql") {
    ref(qualifiedName: "master") {
      target {
        ... on Commit {

        message
      }
    }
  }
}
}
```

Inline fragments for interfaces and unions

- Here is an example from the facebook/graphql repository.

```
query RepoUnionExample {
  repository(owner: "facebook", name: "graphql") {
    issueOrPullRequest(number: 3) {
      __typename
    }
  }
}
```

- Here is a query to pick these different fields based on the type of the issueOrPullRequest whose number is 5

```
query RepoUnionExampleFull {
  repository(owner: "facebook", name: "graphql") {
    issueOrPullRequest(number: 5) {
      ... on PullRequest {
        merged
        mergedAt
      }
      ... on Issue {
        closed
        closedAt
      }
    }
  }
}
```

- For example, a GitHub user search might return a user object or an organization object.
- Here is a query to search GitHub users for the term "graphql".

```
query TestSearch {
  search(first: 100, query: "graphql", type: USER) {
    nodes {
      ... on User {
        name
        bio
      }
      ... on Organization {
        login
        description
      }
    }
  }
}
```

Summary

- You can pass arguments to GraphQL fields when sending requests.
- GraphQL servers can use these arguments to support features like identifying a single record, limiting the number of records returned by a list field, ordering records and paginating through them, searching and filtering, and providing input values for mutations.
- You can give any GraphQL field an alias name, This enables you to customize a server response using the client's request text.

"Complete Lab”