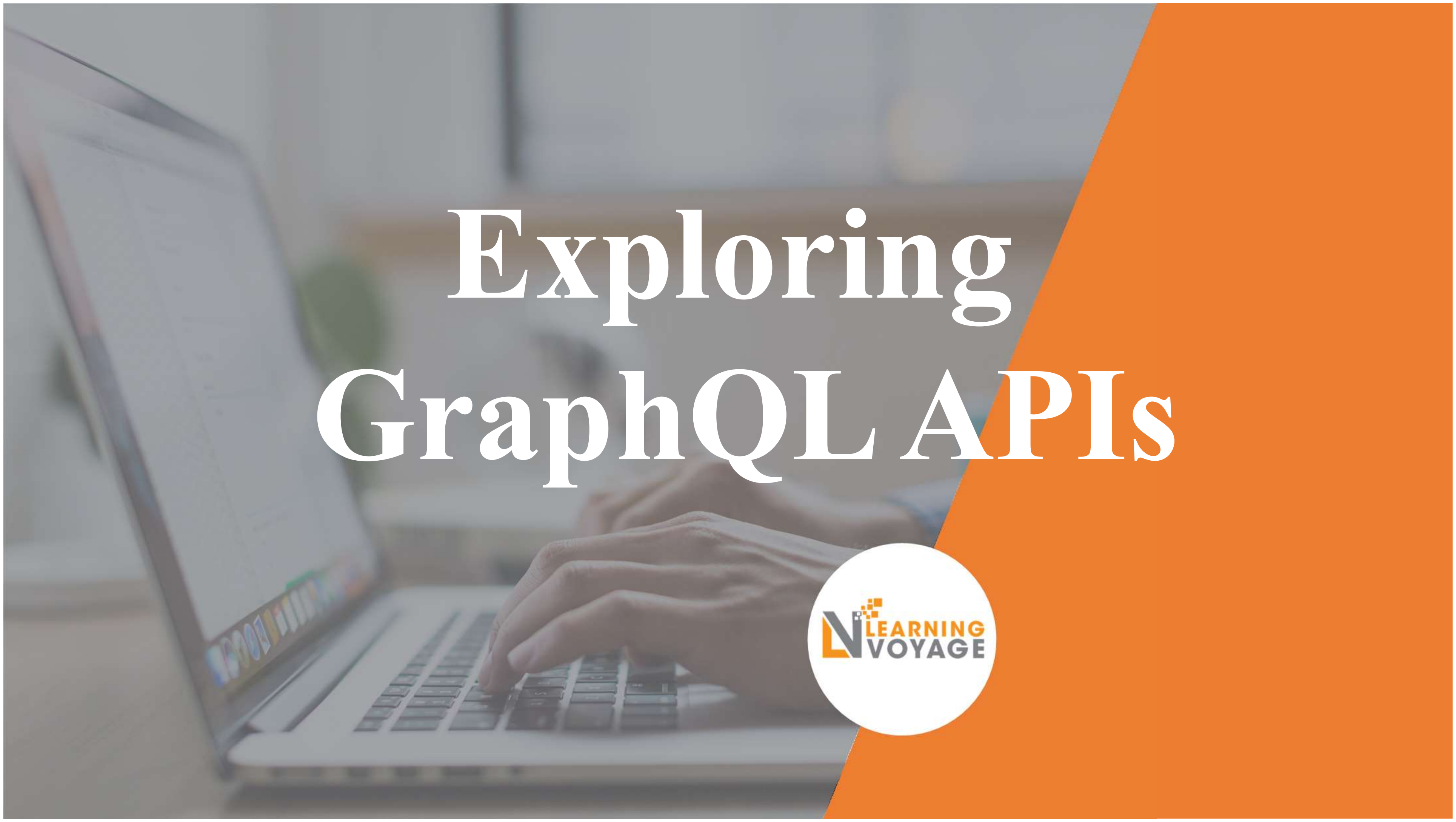


Exploring GraphQL APIs

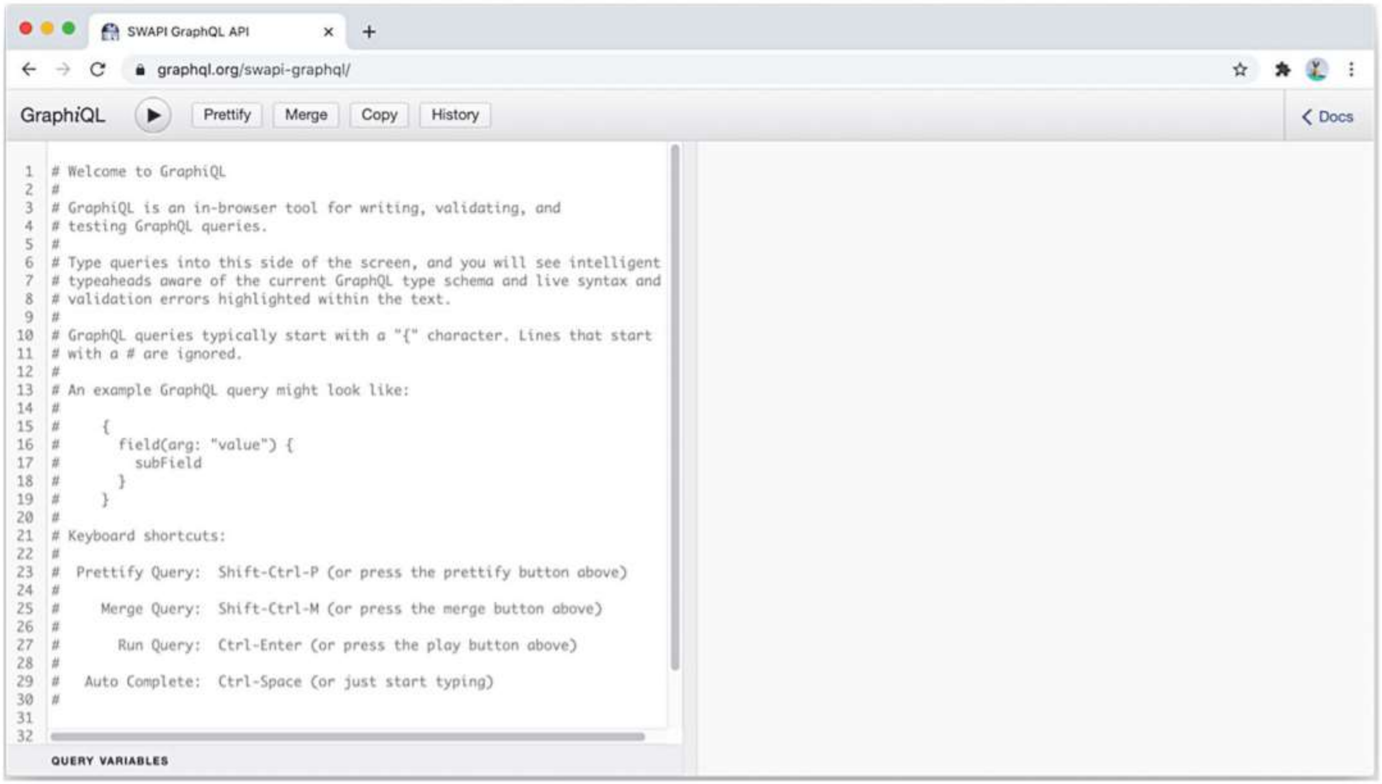
LEARNING VOYAGE

Exploring GraphQL APIs

This lesson covers

- Using GraphQL’s in-browser IDE to test GraphQL requests
- Exploring the fundamentals of sending GraphQL data requests
- Exploring read and write example operations from the GitHub GraphQL API
- Exploring GraphQL’s introspective features

The GraphiQL editor



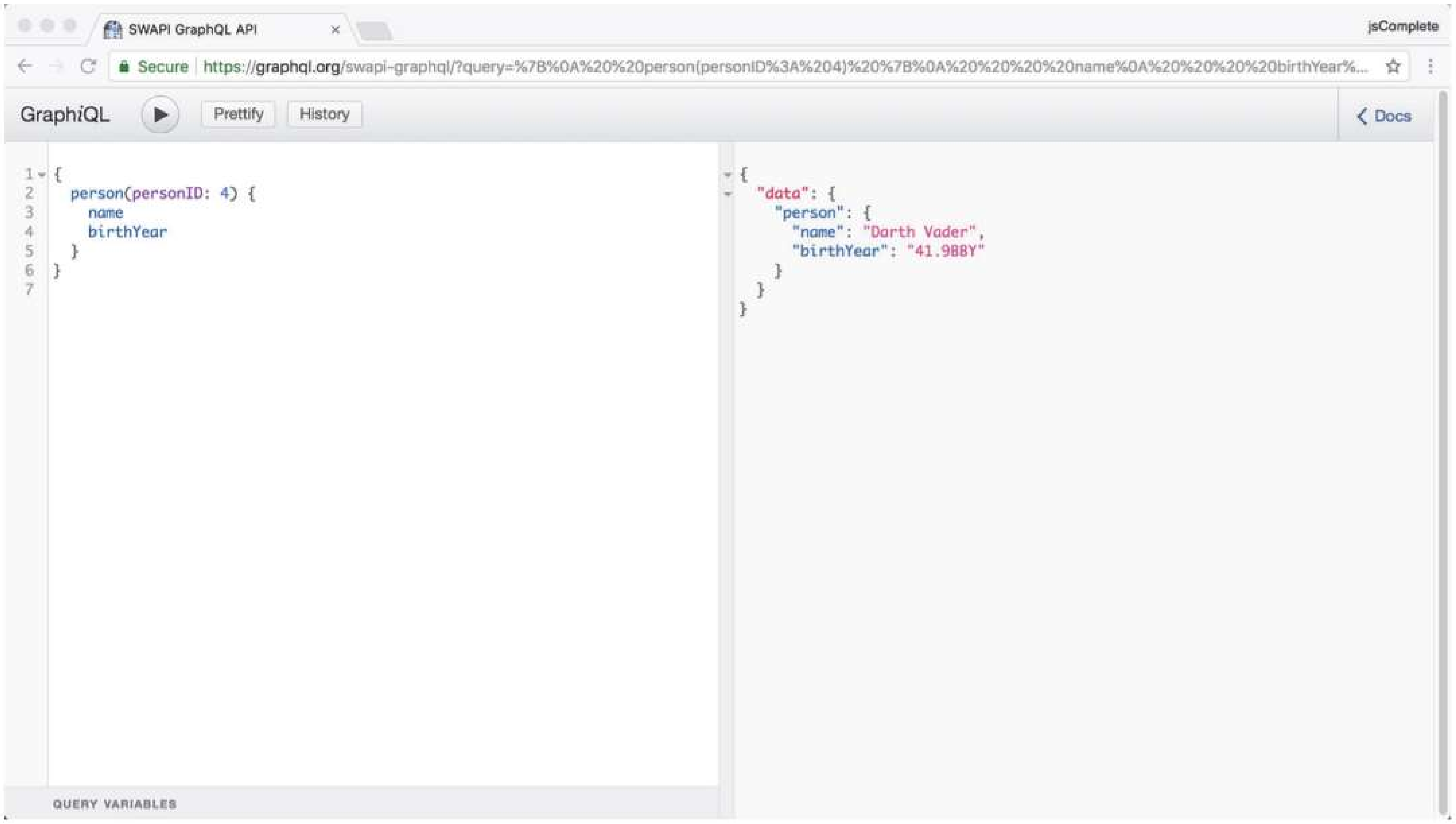
```
# Welcome to GraphiQL
#
# GraphiQL is an in-browser tool for writing, validating, and
# testing GraphQL queries.
#
# Type queries into this side of the screen, and you will see intelligent
# typeaheads aware of the current GraphQL type schema and live syntax and
# validation errors highlighted within the text.
#
# GraphQL queries typically start with a "{" character. Lines that start
# with a # are ignored.
#
# An example GraphQL query might look like:
#
#     {
#       field(arg: "value") {
#         subField
#       }
#     }
#
# Keyboard shortcuts:
#
#  Prettify Query:  Shift-Ctrl-P (or press the prettify button above)
#
#     Merge Query:  Shift-Ctrl-M (or press the merge button above)
#
#       Run Query:  Ctrl-Enter (or press the play button above)
#
#   Auto Complete:  Ctrl-Space (or just start typing)
#
```

The GraphiQL editor

Go ahead and type the following simple GraphQL query in the editor

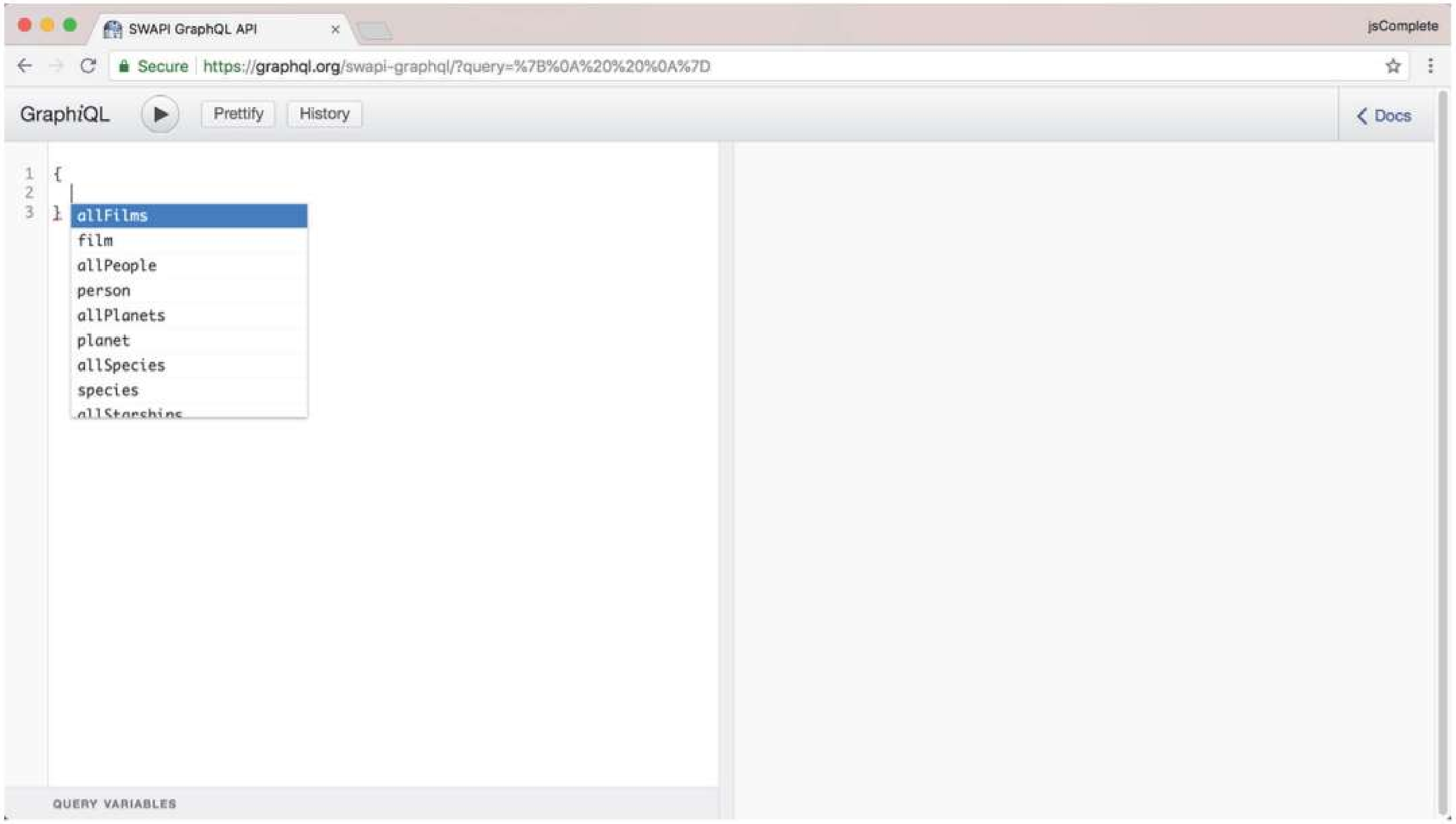
A query for the `person` field

```
{
  person(personID: 4) {
    name
    birthYear
  }
}
```


SWAPI GraphQL API
jsComplete
Secure
https://graphql.org/swapi-graphql/?query=%7B%0A%20%20person(personID%3A%204)%20%7B%0A%20%20%20%20name%0A%20%20%20%20birthYear%...
GraphiQL
Prettify
History
Docs
QUERY VARIABLES

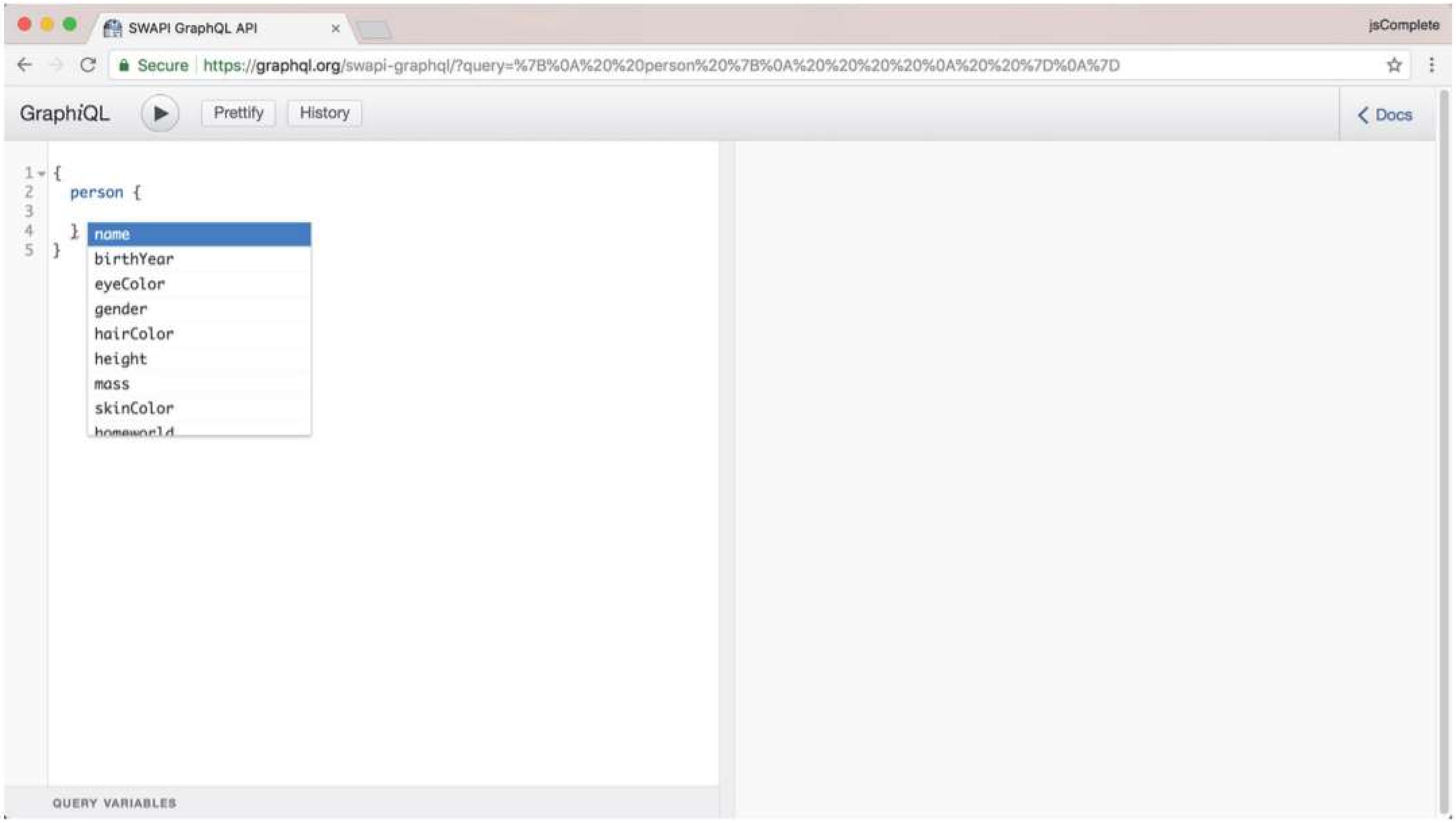
```
{
  person(personID: 4) {
    name
    birthYear
  }
}
```

```
{
  "data": {
    "person": {
      "name": "Darth Vader",
      "birthYear": "41.9BBY"
    }
  }
}
```



```
{

}
```

```
allFilms
film
allPeople
person
allPlanets
planet
allSpecies
species
allStarships
```

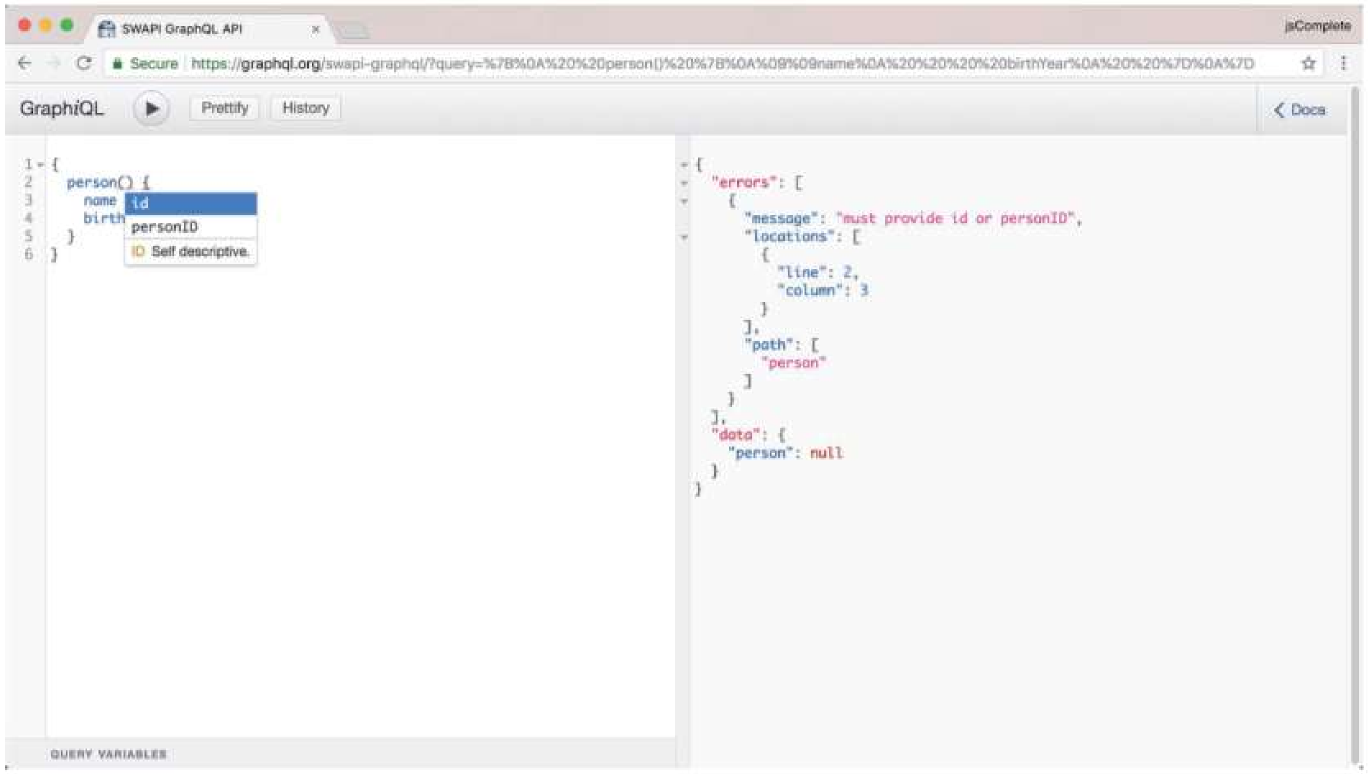

SWAPI GraphQL API
jsComplete
Secure
https://graphql.org/swapi-graphql/?query=%7B%0A%20%20person%20%7B%0A%20%20%20%20%0A%20%20%7D%0A%7D
GraphiQL
Prettify
History
Docs
QUERY VARIABLES

```
{
  person {

  }
}
```

```
name
birthYear
eyeColor
gender
hairColor
height
mass
skinColor
homeworld
```

```
{
  "errors": [
    {
      "message": "must provide id or personID",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
        ],
        "path": [
          "person"
        ]
      }
    ],
    "data": {
      "person": null
    }
  }
```

The GraphiQL editor



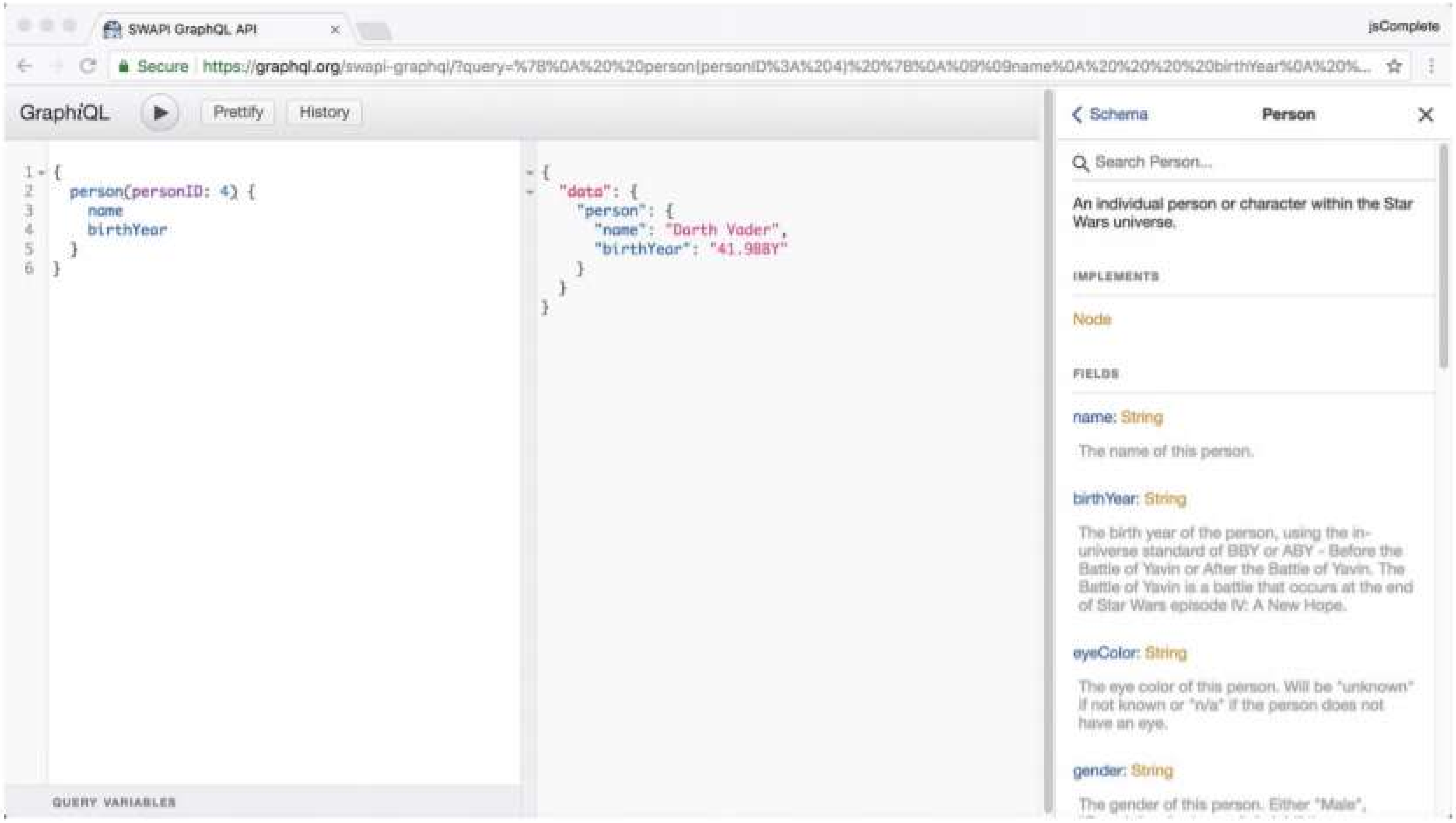
SWAPI GraphQL API

https://graphql.org/swapi-graphql/?query=%7B%0A%20%20person()%20%7B%0A%09%09name%0A%20%20%20%20birthYear%0A%20%20%20%20%7D%0A%7D

```
{
  person() {
    name
    birth
  }
}
```

```
id
personID
ID Self descriptive.
```

```
{
  "errors": [
    {
      "message": "must provide id or personID",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "path": [
        "person"
      ]
    }
  ],
  "data": {
    "person": null
  }
}
```


SWAPI GraphQL API
jsComplete
Secure https://graphql.org/swapi-graphql/?query=%7B%0A%20%20person(personID%3A%204)%20%7B%0A%09%09name%0A%20%20%20%20birthYear%0A%20%...
GraphiQL
Prettify
History
QUERY VARIABLES
Schema
Person

```
{
  person(personID: 4) {
    name
    birthYear
  }
}
```

```
{
  "data": {
    "person": {
      "name": "Darth Vader",
      "birthYear": "41.9BBY"
    }
  }
}
```

Search Person...

An individual person or character within the Star Wars universe.

IMPLEMENTS

Node

FIELDS

name: String

The name of this person.

birthYear: String

The birth year of the person, using the in-universe standard of BBY or ABY - Before the Battle of Yavin or After the Battle of Yavin. The Battle of Yavin is a battle that occurs at the end of Star Wars episode IV: A New Hope.

eyeColor: String

The eye color of this person. Will be "unknown" if not known or "n/a" if the person does not have an eye.

gender: String

The gender of this person. Either "Male",

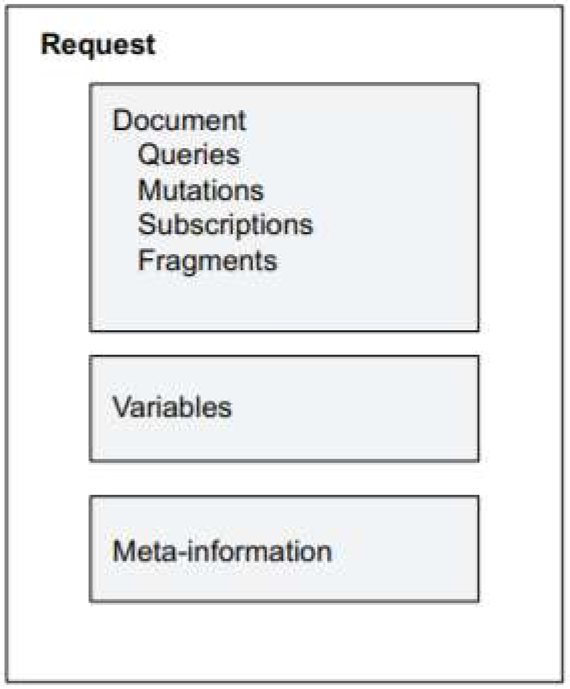
The basics of the GraphQL language

Requests

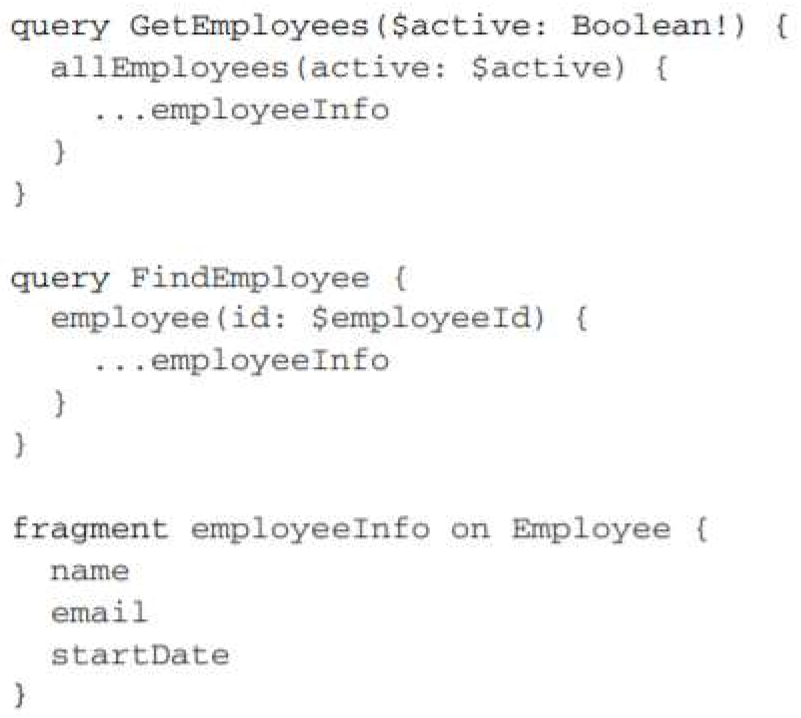
- At the core of a GraphQL communication is a request object, The source text of a GraphQL request is often referred to as a document.
- A document contains text that represents a request through operations like queries, mutations, and subscriptions.
- In addition to the main operations, a GraphQL document text can contain fragments that can be used to compose other operations,

Requests

The structure of a GraphQL request



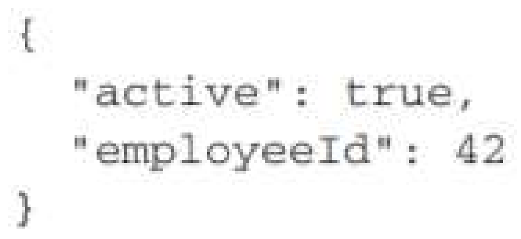
Requests



```
query GetEmployees($active: Boolean!) {
  allEmployees(active: $active) {
    ...employeeInfo
  }
}

query FindEmployee {
  employee(id: $employeeId) {
    ...employeeInfo
  }
}

fragment employeeInfo on Employee {
  name
  email
  startDate
}
```

Requests

- Since this document uses generic variables (the ones starting with the $ sign), we need a JSON object to represent values specific to a request.



```
{
    "active": true,
    "employeeId": 42
}
```

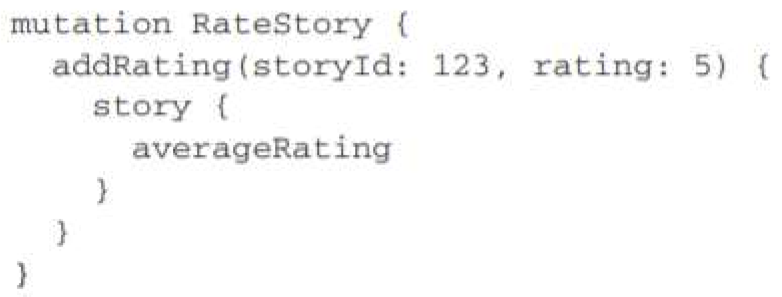
Requests

- Also, since the document contains more than one operation (GetEmployees and FindEmployee), the request needs to provide the desired operation to be executed.

```
operationName="GetEmployees"
```

Requests

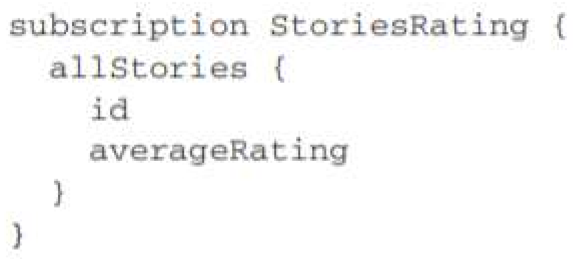
- Here is a hypothetical example of a mutation operation.



```
mutation RateStory {
  addRating(storyId: 123, rating: 5) {
    story {
      averageRating
    }
  }
}
```

Requests

- Here is a hypothetical example of a subscription operation.



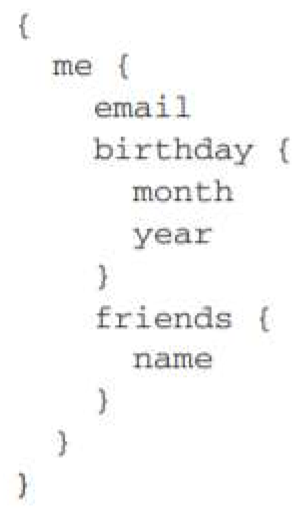
```
subscription StoriesRating {
  allStories {
    id
    averageRating
  }
}
```

Fields

- One of the core elements in the text of a GraphQL operation is the field.
- The simplest way to think about a GraphQL operation is as a way to select fields on objects.
- A field always appears within a selection set (inside a pair of curly brackets), and it describes one discrete piece of information that you can retrieve about an object.

Fields

- Here is an example GraphQL query with different types of fields



```
{
  me {
    email
    birthday {
      month
      year
    }
    friends {
      name
    }
  }
}
```

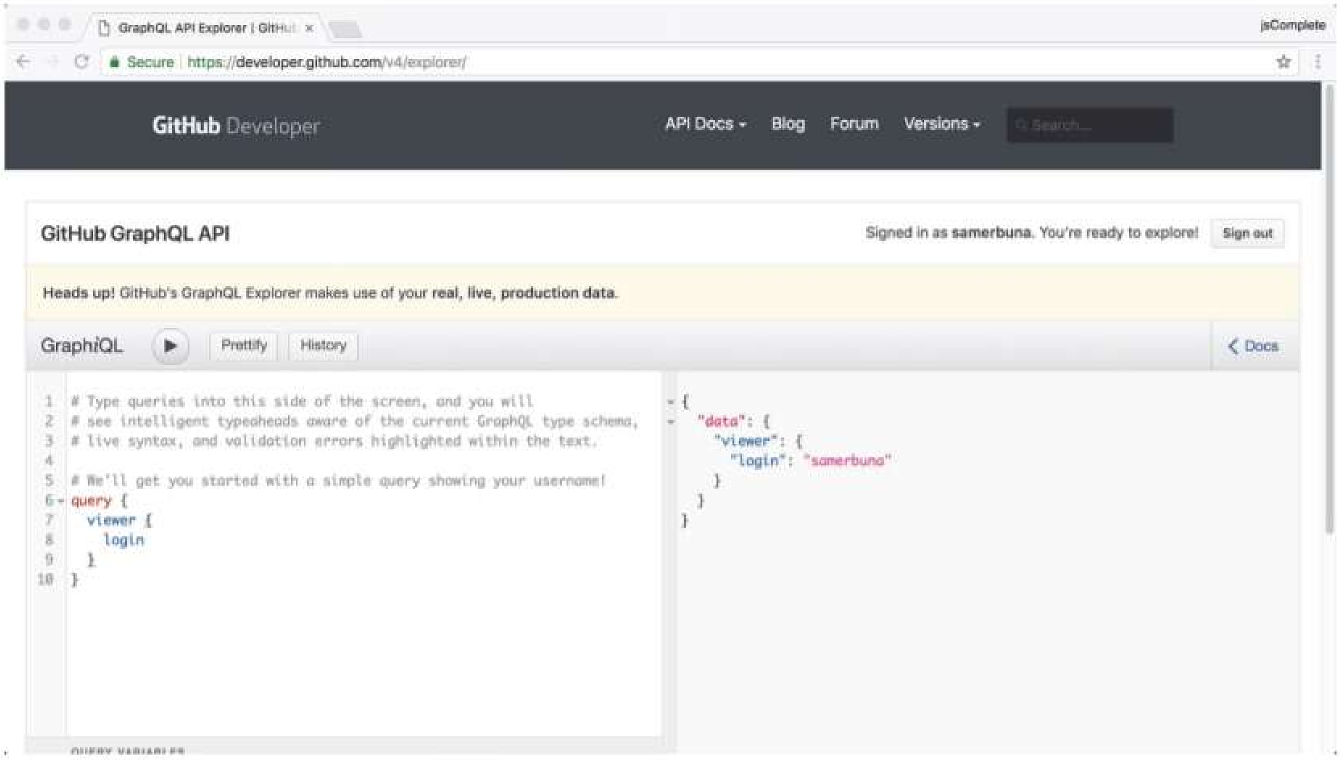
Some typical examples of root fields include references to a currently logged-in user. These fields are often named `viewer` or `me`. For example:

```
{
  me {
    username
    fullName
  }
}
```

Root fields are also generally used to access certain types of data referenced by a unique identifier. For example:

```
# Ask for the user whose ID equal to 42
{
  user(id: 42) {
    fullName
  }
}
```

Examples from the GitHub API



GraphQL API Explorer | GitHub

jsComplete

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

GitHub Developer

API Docs Blog Forum Versions

GitHub GraphQL API

Signed in as **samerbuna**. You're ready to explore! Sign out

Heads up! GitHub's GraphQL Explorer makes use of your **real, live, production data**.

GraphiQL Prettify History Docs

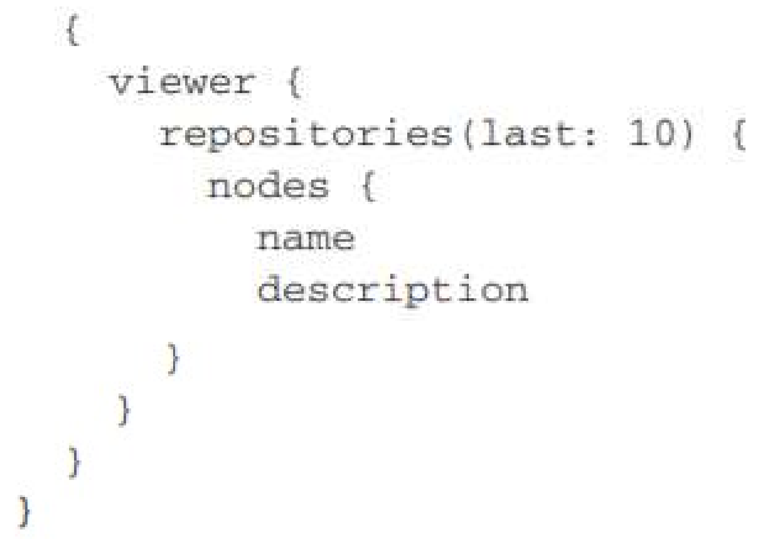
```
# Type queries into this side of the screen, and you will
# see intelligent typeaheads aware of the current GraphQL type schema,
# live syntax, and validation errors highlighted within the text.

# We'll get you started with a simple query showing your username!
query {
  viewer {
    login
  }
}
```

```
{
  "data": {
    "viewer": {
      "login": "samerbuna"
    }
  }
}
```

QUERY VARIABLES

Reading data from GitHub

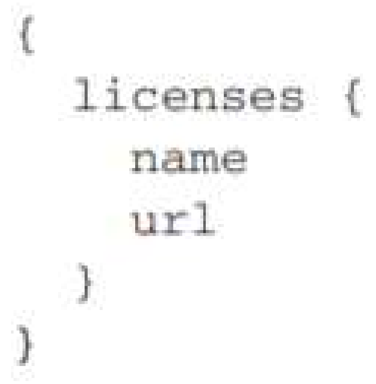
- For example, here is a query to see information about the most recent 10 repositories that you own or contribute to.



```
{
  viewer {
    repositories(last: 10) {
      nodes {
        name
        description

      }
    }
  }
}
```

Reading data from GitHub

- Here is another query to see all the supported licenses in GitHub along with their URLs.

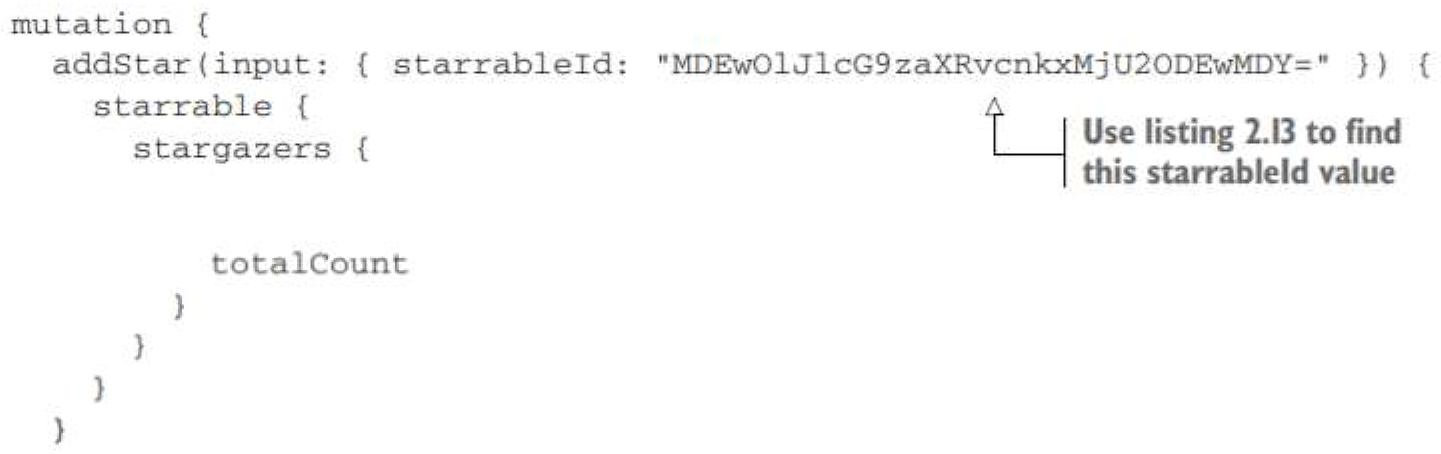


```
{
  licenses {
    name
    url
  }
}
```

Reading data from GitHub

```
{
  repository(owner: "facebook", name: "graphql") {
    issues(first: 10) {
      nodes {
        title
        createdAt
        author {
          login
        }
      }
    }
  }
}
```

Updating data at GitHub



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


          totalCount
        }
      }
    }
  }
```

Use listing 2.13 to find this starrableId value

Updating data at GitHub

- The input for this mutation is a simple object that has a starrableId value, which is the node identifier for the graphql-in-action repository.
- I was able to find that value using this query

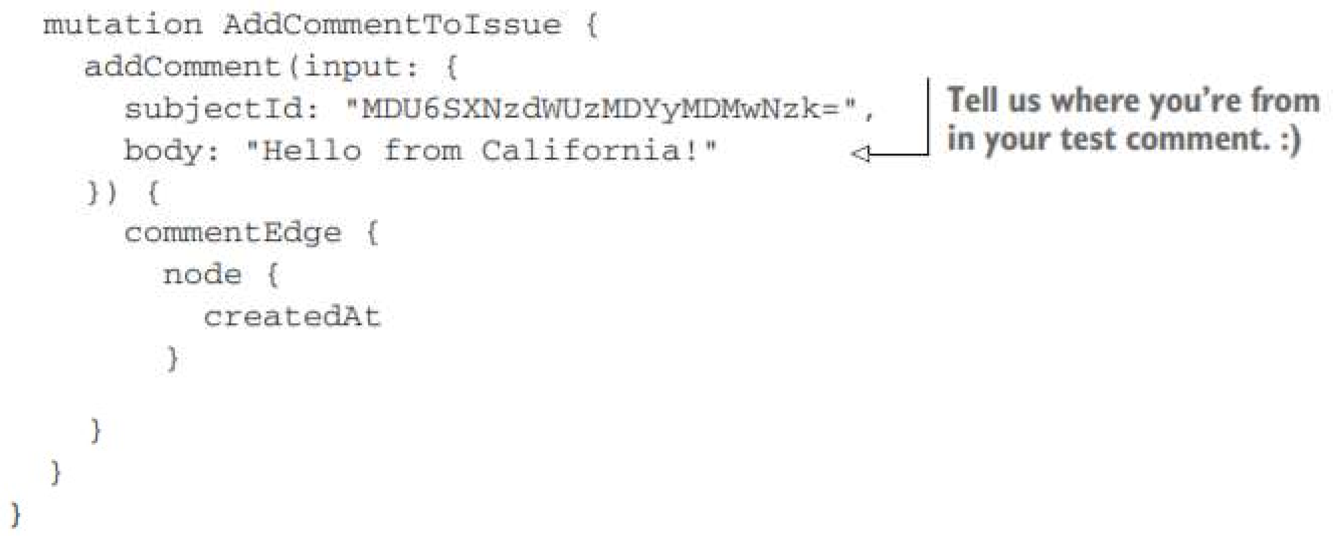
```
{
  repository(name: "graphql-in-action", owner: "jscomplete") {
    id
  }
}
```

Updating data at GitHub

```
query GetIssueInfo {
  repository(owner: "jscomplete", name: "graphql-in-action") {
    issue(number: 1) {
      id
      title
    }
  }
}
```

Updating data at GitHub

- Now execute the following mutation, which uses that id value.



```
mutation AddCommentToIssue {
  addComment(input: {
    subjectId: "MDU6SXNzdWUzMDYyMDMwNzk=",
    body: "Hello from California!"
  }) {
    commentEdge {
      node {
        createdAt
      }

    }
  }
}
```

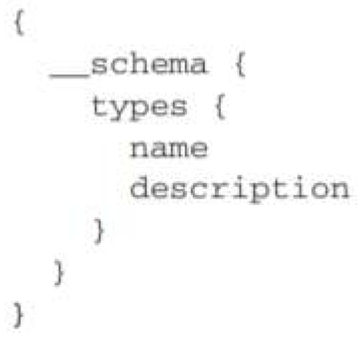
Tell us where you're from in your test comment. :)

Introspective queries

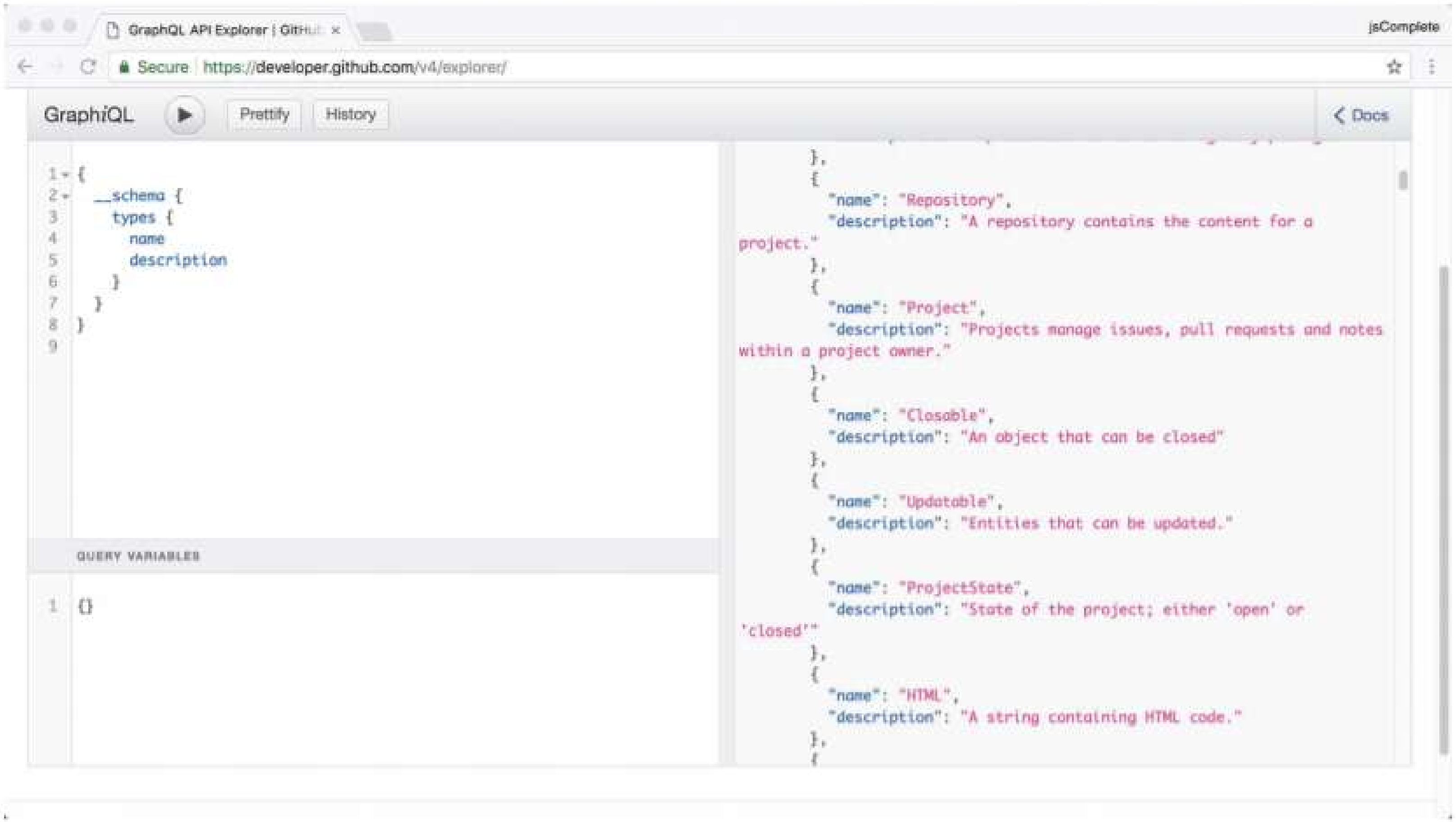
- GraphQL APIs support introspective queries that can be used to answer questions about the API schema.
- This introspection support gives GraphQL tools powerful functionality, and it drives the features we have been using in the GraphiQL editor.
- For example, the awesome type-ahead list in GraphiQL is sourced with an introspective query.

Introspective queries

- Let’s ask the GitHub API schema what types it supports. Here is an introspective query to do that



```
{
  __schema {
    types {
      name
      description
    }
  }
}
```



GraphQL API Explorer | GitHub
jsComplete
Secure https://developer.github.com/v4/explorer/
GraphiQL
Prettify
History
Docs
QUERY VARIABLES

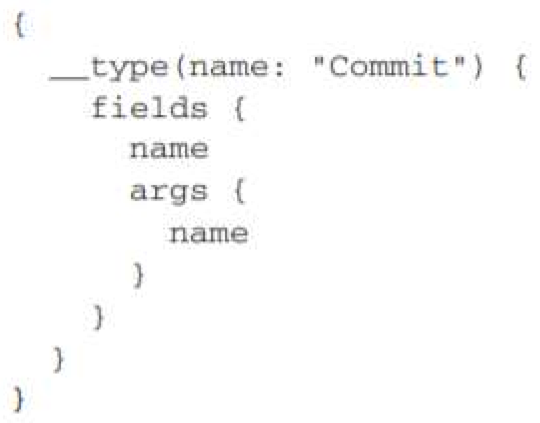
```
{
  __schema {
    types {
      name
      description
    }
  }
}
```

```
{}
```

```
        },
        {
          "name": "Repository",
          "description": "A repository contains the content for a
project."
        },
        {
          "name": "Project",
          "description": "Projects manage issues, pull requests and notes
within a project owner."
        },
        {
          "name": "Closable",
          "description": "An object that can be closed"
        },
        {
          "name": "Updatable",
          "description": "Entities that can be updated."
        },
        {
          "name": "ProjectState",
          "description": "State of the project; either 'open' or
'closed'"
        },
        {
          "name": "HTML",
          "description": "A string containing HTML code."
        },
        {
```

Introspective queries

- For example, here is a query to find all the supported fields under the type Commit along with any arguments they accept.



```
{
  __type(name: "Commit") {
    fields {
      name
      args {
        name
      }
    }
  }
}
```

Summary

- GraphiQL is an in-browser IDE for writing and testing GraphQL requests.
- It offers many great features to write, validate, and inspect GraphQL queries and mutations.
- These features are made possible thanks to GraphQL's introspective nature, which comes with its mandatory schemas.
- A GraphQL request consists of a set of operations, an object for variables, and other meta-information elements as needed.

"Complete Lab"