



API TESTING SUMMIT 2021

Automated Testing of GraphQL API

Unmesh Gundecha

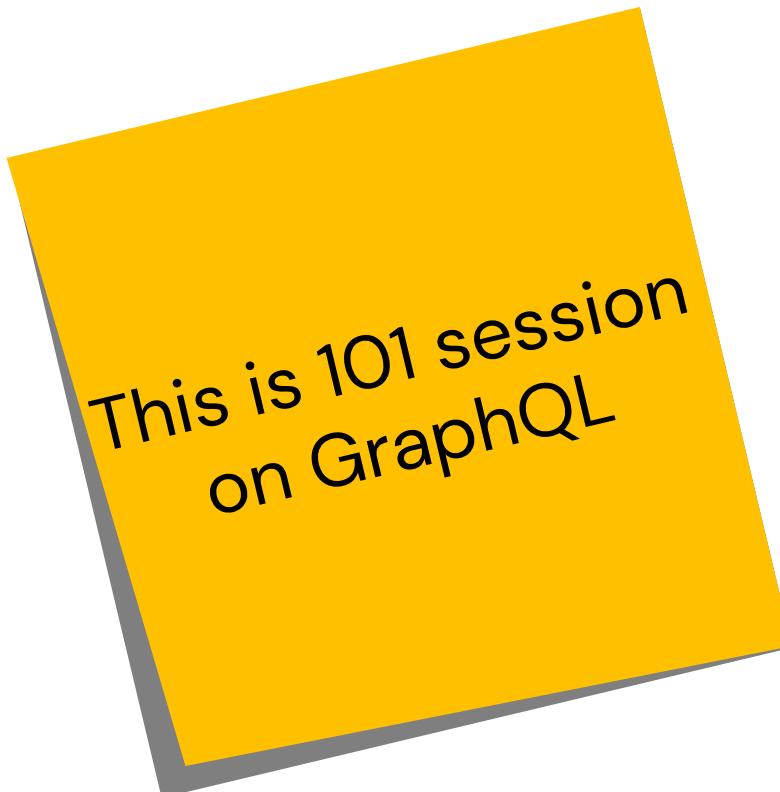
Senior Architect, Test Engineering DveOps and
Developer Experience (DX)

Find me on LinkedIn - <https://www.linkedin.com/in/upgundecha/> | Twitter [@upgundecha](https://twitter.com/upgundecha)

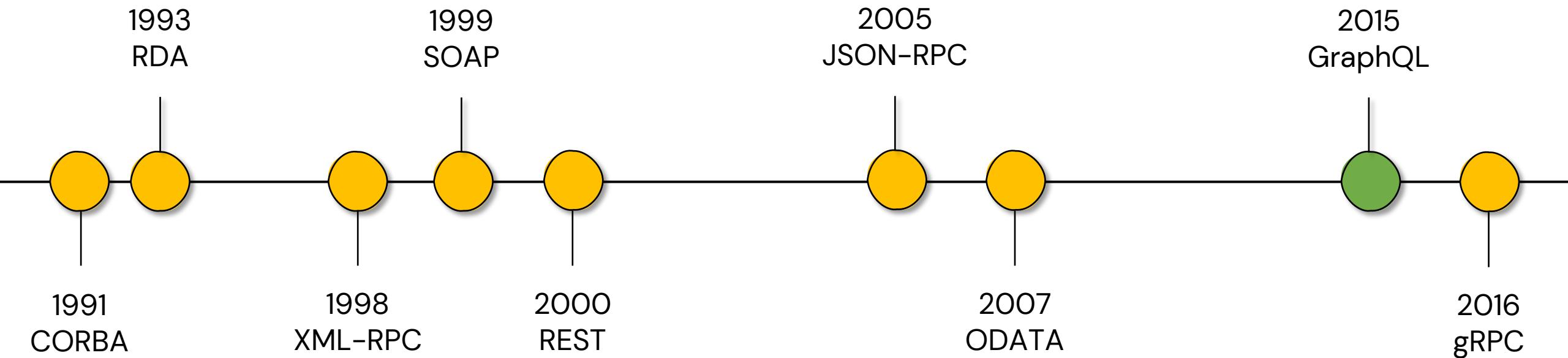


Contents

- A brief history of API
- What is GraphQL?
- REST vs GraphQL
- Benefits of GraphQL
- How GraphQL Works?
- Testing Tools
- Schema and Types
- Queries and Mutations
- Demo (Playground, Postman & Supertest)
- Resources



A brief history of API

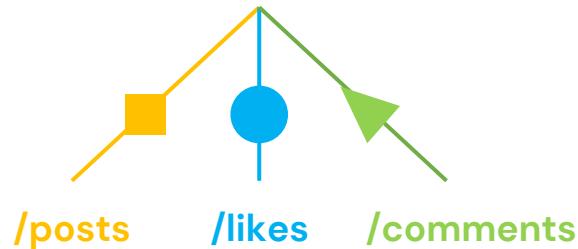


What is GraphQL?

A query language for your API

Built at Facebook in 2012 and open sourced in 2015

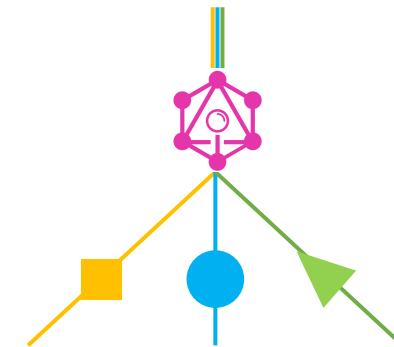
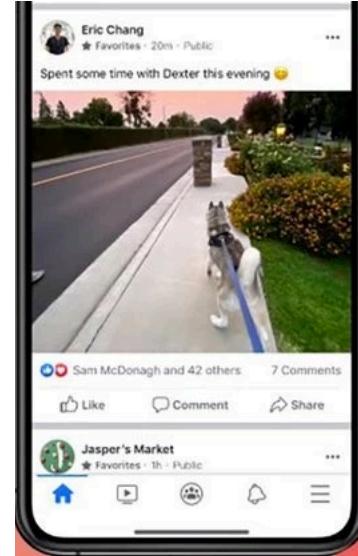
REST



Fetching data with multiple API calls

Performance degradation with over fetching

GraphQL



Posts Likes Comments

Fetching data with a single API call

What is GraphQL?

GraphQL is a **query language for APIs** and a runtime for fulfilling those queries with your existing data. GraphQL provides a complete and understandable description of the data in your API, gives clients the power to ask for **exactly what they need** and nothing more, makes it easier to evolve APIs over time, and enables powerful developer tools.

Describe your data

```
type Project {  
  name: String  
  tagline: String  
  contributors: [User]  
}
```

Ask for what you want

```
{  
  project(name: "GraphQL") {  
    tagline  
  }  
}
```

Get predictable results

```
{  
  "project": {  
    "tagline": "A query language for APIs"  
  }  
}
```

REST vs GraphQL

- 1 Filter down the data
- 2 Aggregate the data
- 3 Waterfall requests for data

from this

```
/users/id  
/users/id/posts  
/users/id/likes  
/users/id/comments
```

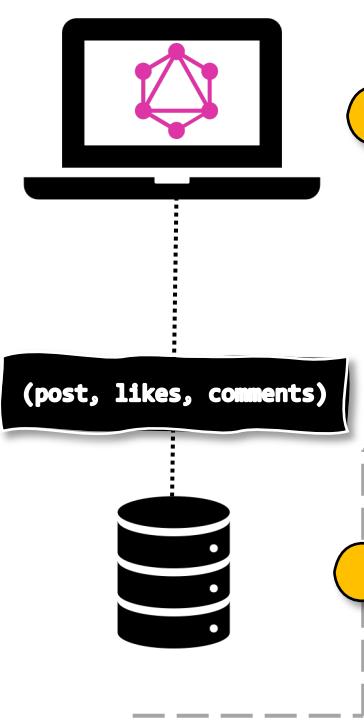
- 1 Receive exactly what you ask for
- 2 No filtering or aggregation
- 3 Single call

to this

```
query {  
  User(id: 1) {  
    name  
    posts {  
      title  
      likes  
    }  
    comments(last: 3) {  
      name  
      text  
    }  
  }  
}
```

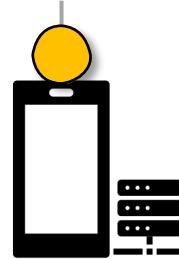
REST vs GraphQL

	REST	GraphQL
Architecture	Server-driven	Client-driven
Organized in terms of	endpoints	Schema and types
Format	XML, JSON, HTML, Plain Text	JSON
Self documenting	No	Yes
Learning Curve	Easy	Medium
Community	Large	Growing
Versioning	Yes	Not Required (depreciation of types)
Use Cases	Public APIs, resource driven apps	Mobile APIs, complex systems, micro-services



UNIQUE ENDPOINT

With single call gets all data and sub data

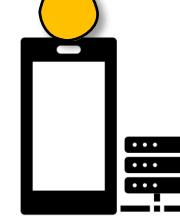


Request only the data that user needs, making response faster and efficient



MULTIPLE ENDPOINTS

Several calls to get information that is needed



Contains all the data, you cannot ask for what is needed, and this affects performance

Server driven,
Doesn't require versioning (types can be deprecated)
Growing community and adoption
Self documenting
Use cases - Mobile APIs, complex systems, micro-services

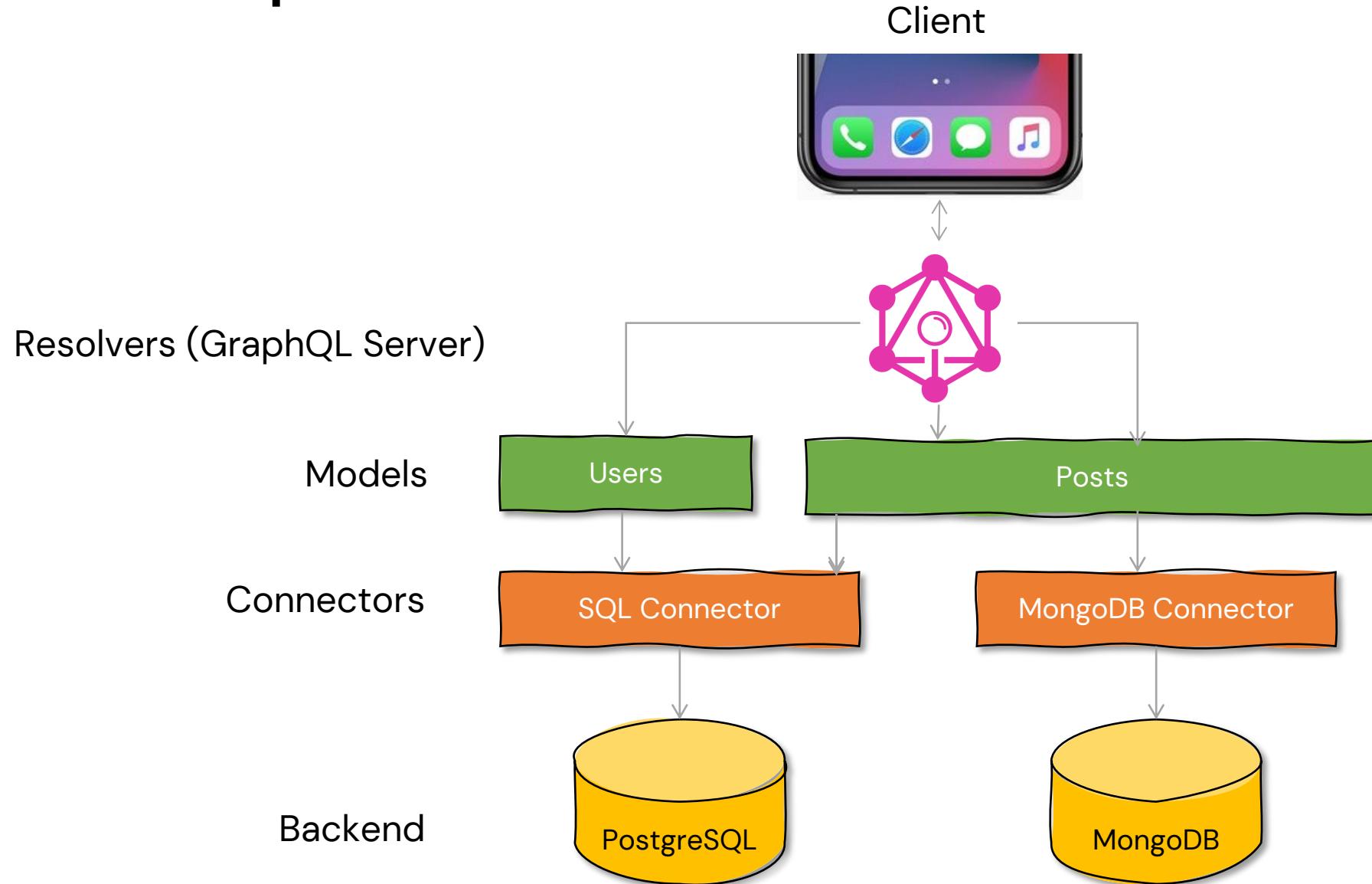
Client driven
Versioning adds complexity
Widely used and large user community
Lack self documentation
Use cases - Public APIs, resource driven apps

Benefits of GraphQL



- Good fit for complex systems and microservices
- Fetching data with a single API call
- No over- and under-fetching problems
- Tailoring requests to your needs
- Validation and type checking out of the box
- Autogenerating API documentation
- API evolution without versioning
- Code-sharing

How GraphQL Works?



Our demo app

amazon.com/Unmesh-Gundecha/e/B00ATKDJOA

Amazon Associates SiteStripe Get Link: Native Shopping Ads (New) Share: Try Mobile GetLink, a tool to create associate links on the go. Learn more Earnings Help

amazon Deliver to Apurva Houston 77058 Kindle Store Search Hello, Unmesh Account & Lists Returns & Orders Cart

All Buy Again Livestreams Shopper Toolkit Health & Household Amazon Basics Coupons Beauty & Personal Care Amazon Launchpad Watch The Tomorrow War now

Buy a Kindle Kindle eBooks Kindle Unlimited Prime Reading Best Sellers & More Kindle Book Deals Kindle Singles Newsstand Manage content and devices Advanced Search

The screenshot shows an Amazon author profile page for 'Unmesh Gundecha'. At the top left is a circular profile picture of a man with glasses. Below it is a yellow button labeled '+ Follow'. To the right of the profile picture is the author's name, 'Unmesh Gundecha'. Below the author's name are five book covers displayed in a row. From left to right: 1. 'Selenium Testing Tools Cookbook Second Edition' by Unmesh Gundecha, Kindle Edition, \$22.39. 2. 'Learning Selenium Testing Tools with Python' by Unmesh Gundecha, Kindle Edition, \$16.54. 3. 'Selenium Testing Tools Cookbook' by Unmesh Gundecha and Palash Aiyar, Kindle Edition, \$16.54. 4. 'Selenium WebDriver 3 Practical Guide' by Unmesh Gundecha and Palash Aiyar, Kindle Edition, \$31.99. 5. 'INSTANT Selenium Testing Tools Starter' by Unmesh Gundecha, Paperback, \$22.99. Below the books are sections for 'About Unmesh Gundecha' (a technology-focused IT professional with over 18 years of experience in Software Engineering, Agile Software Development, Test), 'Author Updates' (with a thumbnail image of the author's head and shoulders), and three blog post links: 'Cypress custom command for', 'Awesome online software testing', and 'More Selenium4 Good'.

Unmesh Gundecha

+ Follow

Follow to get new release updates and improved recommendations

About Unmesh Gundecha

Unmesh is a technology-focused IT professional with over 18 years of experience in Software Engineering, Agile Software Development, Test

Author Updates

Blog post Cypress custom command for

Blog post Awesome online software testing

Blog post More Selenium4 Good

Schema and Types

- Schema is contract between client and server
- Each object is of specific type
- Each type defines a number of fields
- Each field has a type (String, Int, Float, Boolean)
- The exclamation mark denotes that the field is mandatory or non-nullable

```
const { buildSchema } = require("graphql");

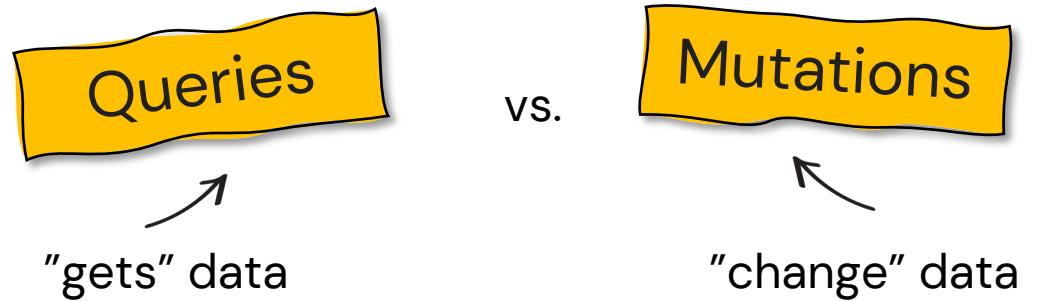
const schema = buildSchema(`
  type Query {
    authors: [Author!]!,
    author(id: Int!): Author!
  }

  type Author {
    id: ID!
    name: String!
    books: [Book!]
  }

  type Book {
    id: ID!
    title: String!
    published: Boolean!
    year: Int!
    link: String
    rating: Float!
    author: Author!
  }
`);

module.exports = schema;
```

Queries and Mutations



Queries are used to request the data from the server. Unlike REST APIs where there's a clearly defined structure of information returned from each endpoint, GraphQL always exposes only one endpoint, allowing the client to decide what data it really needs from a predefined pattern

Mutations are used to modify data on the server. The mutations are equivalent to how you'd use the CUD operations using HTTP Verbs such as POST, PUT, PATCH and DELETE from REST APIs. In GraphQL it will always be a POST call. It is a write followed by fetch operation.

Queries

At its simplest, GraphQL is about asking for specific fields on objects. Let's start by looking at a very simple query and the result we get when we run it:

```
query {  
  authors {  
    name  
  }  
}
```

"gets"

```
{  
  "data": {  
    "authors": [  
      {  
        "name": "Unmesh Gundecha"  
      },  
      {  
        "name": "Dima Kovalenko"  
      },  
      {  
        "name": "James Bond"  
      }  
    ]  
  }  
}
```

Queries

```
query {  
  author(id: 1) {  
    name  
    books {  
      title  
    }  
  }  
}
```

"gets"

```
{  
  "data": {  
    "author": {  
      "name": "Unmesh Gundecha",  
      "books": [  
        {  
          "title": "Selenium Testing  
Tools Cookbook - Second Edition 2nd  
Edition"  
        },  
        {  
          "title": "Selenium  
WebDriver 3 Practical Guide: End-to-  
end automation testing for web and  
mobile browsers with Selenium  
WebDriver, 2nd Edition"  
        }  
      ]  
    }  
  }  
}
```

Mutation

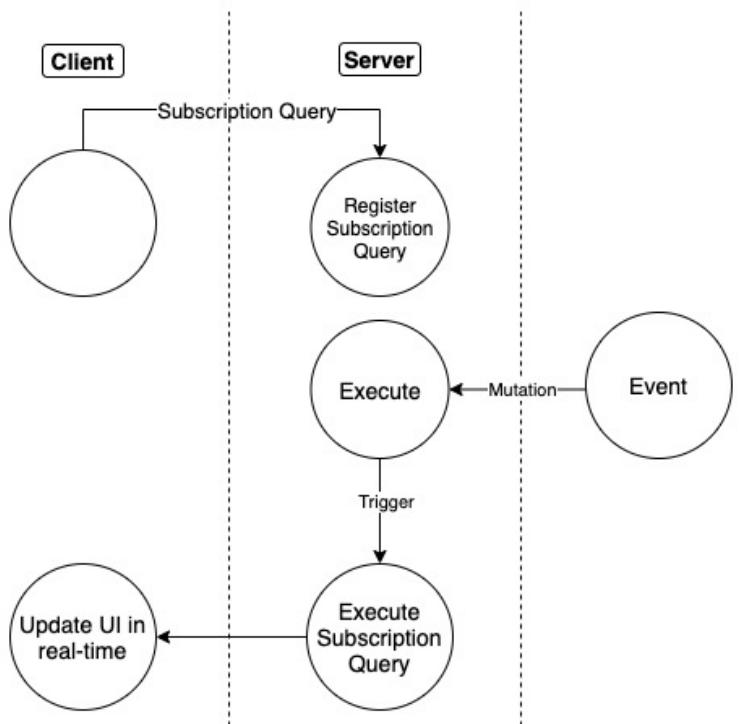
```
mutation {  
  createAuthor(name: "Foo Bar") {  
    id  
    name  
  }  
}
```

"adds"

```
{  
  "data": {  
    "createAuthor": {  
      "id": "6",  
      "name": "Foo Bar"  
    }  
  }  
}
```

Subscriptions

Subscriptions allow clients to listen to real-time messages from the server. The client connects to the server with a bi-directional communication channel using the WebSocket protocol and sends a subscription query that specifies which event it is interested in. When an event is triggered, the server executes the stored GraphQL query, and the result is sent back to the client using the same communication channel.



```
type Todo @withSubscription {  
  id: ID!  
  title: String!  
  description: String!  
  completed: Boolean!  
}
```

Tools for testing GraphQL



cURL



Postman



RestAssured



Karate



Supertest



Easygraphql
-tester



RestSharp



graphene
(Python)



graphql-ruby

or combining any graphql client with a testing framework can be used to automate tests...



Demo Time!

Playground

The screenshot shows a GraphQL playground interface running in a web browser. The URL in the address bar is `localhost:4000/playground`. The main area displays a GraphQL query and its results.

Query:

```
1 ▼ query {
2   ▼ author(id: 1) {
3     name
4     books {
5       title
6       rating
7     }
8   }
9 }
```

Result:

```
▼ {
  ▼ "data": {
    ▼ "author": {
      "name": "Unmesh Gundecha",
      "books": [
        {
          "title": "Selenium Testing Tools Cookbook – Second Edition 2nd Edition",
          "rating": 4.6
        },
        {
          "title": "Selenium WebDriver 3 Practical Guide: End-to-end automation testing for web and mobile browsers with Selenium WebDriver, 2nd Edition",
          "rating": 4
        }
      ]
    }
  }
}
```

The interface includes tabs for **PRETTIFY**, **HISTORY**, and **COPY CURL**. On the right side, there are buttons for **DOCS** and **SCHEMA**. At the bottom, there are buttons for **QUERY VARIABLES**, **HTTP HEADERS**, and **TRACING**.

Postman - Query

The screenshot shows the Postman application interface for making GraphQL requests. The top navigation bar includes tabs for "POST Authors", "POST Author", "POST Mutation", and "GraphQL". The "GraphQL" tab is selected.

The main workspace displays a "POST" request to "http://localhost:4000/graphql". The "Body" tab is active, showing the following GraphQL query:

```
1 {  
2   authors {  
3     id  
4     name  
5   }  
6 }
```

To the right of the query, there is a "GRAPHQL VARIABLES" section with a single variable named "1".

Below the query editor, the "Body" tab is selected, showing the raw JSON response from the server. The response is:

```
1 {  
2   "data": {  
3     "authors": [  
4       {  
5         "id": "1",  
6         "name": "Unmesh Gundecha"  
7       },  
8       {  
9     ]  
10  }
```

The status bar at the bottom indicates "Status: 200 OK", "Time: 225 ms", and "Size: 360 B".

Postman – Mutation

The screenshot shows the Postman application interface for making a GraphQL mutation request.

Request Section:

- Method:** POST
- URL:** http://localhost:4000/graphql
- Body Tab:** Selected. Options: none, form-data, x-www-form-urlencoded, raw, binary, GraphQL (selected), No schema.

QUERY:

```
1 mutation {
2   createAuthor(name: "Foo Bar") {
3     id
4   }
5 }
```

GRAPHQL VARIABLES:

```
1
```

Response Section:

- Body Tab:** Selected.
- Status:** 200 OK
- Time:** 69 ms
- Size:** 271 B
- Save Response:** Available.

Response Body (Pretty JSON):

```
1 {
2   "data": {
3     "createAuthor": {
4       "id": "4"
5     }
6   }
7 }
```

Supertest - Query

```
const app = require("../src/server");
const supertest = require("supertest");
const { stopDatabase } = require("../src/database");

const request = supertest(app);

afterAll(async () => {
  await stopDatabase();
});

test("fetch authors", async (done) => {
  request
    .post("/graphql")
    .send({
      query: "{ authors{ id, name} }",
    })
    .set("Accept", "application/json")
    .expect("Content-Type", /json/)
    .expect(200)
    .end(function (err, res) {
      if (err) return done(err);
      expect(res.body).toBeInstanceOf(Object);
      expect(res.body.data.authors.length).toEqual(3);
      names = res.body.data.authors.map((val, index) => val.name)
      expect(names).toEqual(expect.arrayContaining(["Unmesh Gundecha", "Dima Kovalenko", "James Bond"]));
      done();
    });
});
```

Supertest - Mutation

```
const app = require("../src/server");
const supertest = require("supertest");
const { stopDatabase } = require("../src/database");

const request = supertest(app);

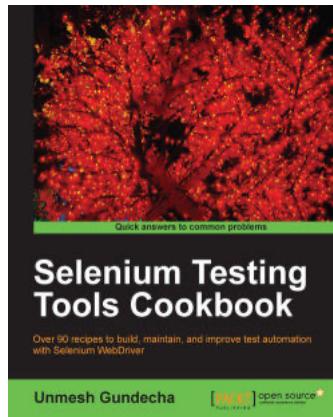
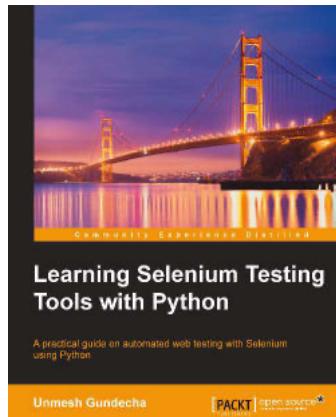
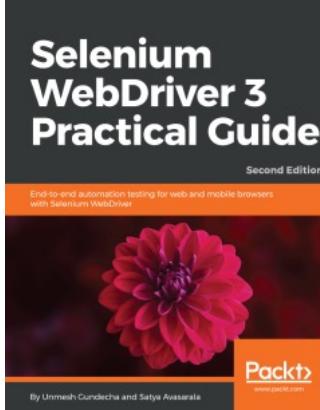
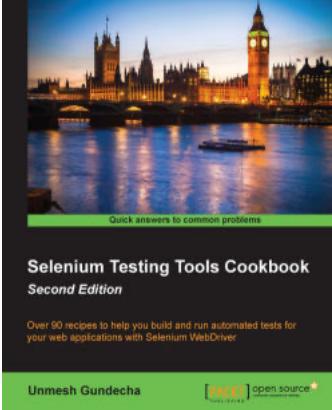
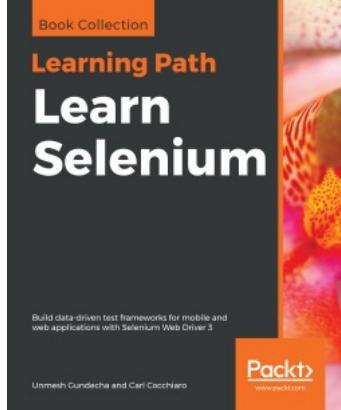
afterAll(async () => {
  await stopDatabase();
});

test("create author", async (done) => {
  request
    .post("/graphql")
    .send({
      query: `mutation {
        createAuthor(name: "Foo Bar") {
          name
        }
      }`
    })
    .set("Accept", "application/json")
    .expect("Content-Type", /json/)
    .expect(200)
    .end(function (err, res) {
      if (err) return done(err);
      expect(res.body).toBeInstanceOf(Object);
      expect(res.body.data.createAuthor.name).toEqual("Foo Bar");
      done();
    });
});
```

Resources

- Code from this session – <https://github.com/chentsulin/awesome-graphql>
- GraphQL – <https://graphql.org/>
- Awesome GraphQL – <https://github.com/chentsulin/awesome-graphql>

My Books & Blog



<https://www.amazon.com/~e/BOOATKDJOA>



Thank you!