**Realtime GraphQL Subscriptions**

In this section, you’ll learn how you can bring realtime functionality into your app by implementing GraphQL subscriptions. The goal is to implement two subscriptions to be exposed by your GraphQL server:

* Send realtime updates to subscribed clients when a new Link element is *created*
* Send realtime updates to subscribed clients when an existing Link element is *upvoted*

**What are GraphQL subscriptions?**

Subscriptions are a GraphQL feature that allows a server to send data to its clients when a specific *event* happens. Subscriptions are usually implemented with [WebSockets](https://en.wikipedia.org/wiki/WebSocket" \t "_blank). In that setup, the server maintains a steady connection to its subscribed client. This also breaks the “Request-Response-Cycle” that were used for all previous interactions with the API.

Instead, the client initially opens up a long-lived connection to the server by sending a *subscription query* that specifies which *event* it is interested in. Every time this particular event happens, the server uses the connection to push the event data to the subscribed client(s).

**Implementing GraphQL subscriptions**

We will be using PubSub from the apollo-server library that we have already been using for our GraphQL server to implement subscriptions to the following *events*:

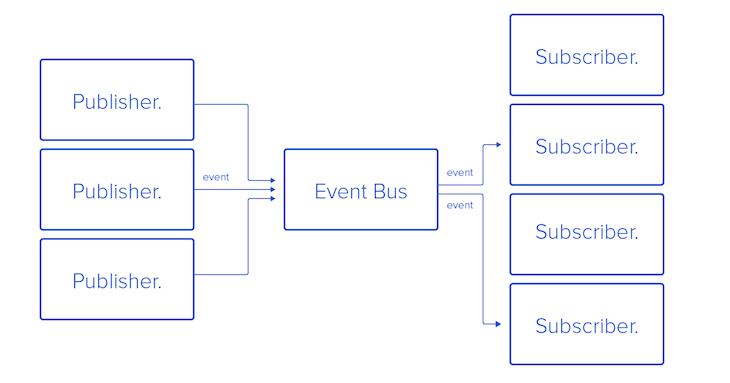
* A new model is **created**
* An existing model **updated**
* An existing model is **deleted**

## How do GraphQL subscriptions work?

Once a GraphQL subscription is executed, a persistent function is created on the server that maps an underlying source stream to a returned response stream, as seen in the diagram below:

GraphQL subscriptions differ from queries in the way that data is delivered to the client. Queries immediately return a single response, while the GraphQL subscriptions return a result every time data is published on a topic to which you have subscribed.

The GraphQL subscription method is facilitated by a [publisher/subscriber mechanism](https://www.apollographql.com/docs/graphql-subscriptions/setup/) that can handle event-driven systems efficiently and at scale. In a publisher/subscriber model, all messages and data flow according to the queue principle (first in, first out) and then to the subscriber:

Diagram of the queue principle in the context of GraphQL subscriptions

*Note: for production, it is recommended to use the*[*pub/sub implementation of Redis*](https://redis.io/topics/pubsub)*.*

There are many packages available on npm that you can use to implement the pub/sub model for GraphQL subscriptions. Below are some of the most commonly used packages:

* [graphql-yoga](https://github.com/prisma-labs/graphql-yoga) : a fully-featured GraphQL server with a focus on easy setup, performance, and a great developer experience
* [graphql-subscriptions](https://github.com/apollographql/graphql-subscriptions): lets you wire GraphQL with a pub/sub system, like Redis, to implement GraphQL subscriptions
* [apollo-server-express](https://github.com/apollographql/apollo-server) : the Express and connect integration of the GraphQL server. Apollo server is a community-maintained, open source GraphQL server that works with many Node.js HTTP server frameworks

We’ll use the graphql-yoga module because it is built on top of the other two packages, and it provides all necessary dependencies and server bindings with Node.js under the hood.

## Building our GraphQL application

We’ll use predefined post data stored inside a JSON file to perform the following operations:

* getPosts : read all posts
* getPost: read a specific post by ID
* updatePost: update a post
* deletePost: delete a post
* createPost: create a post

Then, we’ll add the subscription to the last three operations. Now, it’s time to get our hands dirty with some code! First, make a folder, name it whatever you like, and initialize it with a package.json file as follows:

mkdir graphql-sub

cd graphql-sub

npm init -y

Next, install the required dependencies:

npm i graphql-yoga

Now, we’ll create all of our files:

mkdir src

cd src

touch index.js

postData.json

typeDefs.js

resolver.js

index.js is responsible for the GraphQL server creation with pub/sub, which we’ll see in a minute. postData.json is the JSON file in which we will perform CRUD operations.

create an array of objects for a post with the following schema:

* id:ID!
* title:String!
* subtitle:String!
* body:String!
* published:Boolean!
* author: String!
* upvotes: Int!
* downvotes: Int!
* commentCount: Int!

We use typeDefs.js to create schemas for the operations above. Lastly, resolvers.js has the logic to resolve for all queries, mutations, and subscriptions defined under typeDefs.js.

Inside typeDefs.js, add the following code:

//type definitions and schemas - (operation and data structure)

const typeDefs = `

type Query {

getPosts(query: String):[Post!]!

getPost(query: String):Post!

}

type Post{

id:ID!

title:String!

subtitle:String!

body:String!

published:Boolean!

author: String!

upvotes: Int!

downvotes: Int!

commentCount: Int!

}

type Mutation{

updatePost(

id:ID!

title:String!

subtitle:String!

body:String!

published:Boolean!

author: String!

upvotes: Int!

downvotes: Int!

commentCount: Int!

): Post!

deletePost(id: ID!): Post!

createPost(

id:ID!

title:String!

subtitle:String!

body:String!

published:Boolean!

author: String!

upvotes: Int!

downvotes: Int!

commentCount: Int!

): Post!

}

type Subscription {

post: PostSubscriptionPayload!

}

type PostSubscriptionPayload {

mutation: String!

data: Post!

}

`;

module.exports = typeDefs;

In addition to the normal schema definitions for queries and mutations, we have a type called Subscription that is added on the post object via SubscriptionPayload, a custom type.

Therefore, each time a change is made to a post object, an event will be triggered that returns the name of the mutation performed , either update, delete, or create a post.

Now, let’s code the skeleton of our resolvers.js file for the typeDefs above:

const posts = require('./postData');

//Resolvers - This are the set of the function defined to get the desired output for the given API

const resolvers = {

Query:{

},

Mutation:{

},

Subscription:{

},

}

module.exports = resolvers;

We imported postData, then we added our resolver object, which contains our Query, Mutation, and Subscription objects. We’ll flesh out each object in the sections below.

## Coding our Query object

Inside of our Query object, we’ll define two queries, getPost and getPosts:

Query: {

getPosts() {

return posts;

},

getPost(parent, args) {

return posts.filter((post) => {

const body = post.body.toLowerCase().includes(args.query.toLowerCase());

const title = post.title

.toLowerCase()

.includes(args.query.toLowerCase());

return body || title;

});

},

},

## Coding our mutation object

We’ll define three mutations, createPost, updatePost, and deletePost inside our mutation object.

### createPost

The createPost method checks whether a post for the ID already exists. If it does, we’ll throw an error to the GraphQL server. Otherwise, we’ll create the post from the args object and add it to our posts JSON data:

createPost(parent, args, { pubsub }) {

const id = parseInt(args.id, 10);

const postIndex = posts.findIndex((post)=> post.id === id);

if (postIndex === -1) {

posts.push({

...args

});

pubsub.publish('post', {

post:{

mutation: 'CREATED',

data: {...args}

}

});

return {...args};

};

throw new Error('Post with same id already exist!');

}

Once the post is created, we’ll publish an event called CREATED to all subscribers of the post channel through the socket with the newly created post data in its payload.

### updatePost

In the updatePost method, the post with the specified ID will be updated with the contents of args. If the post ID does not exist, we’ll throw an error:

updatePost(parent, args, { pubsub }) {

const id = parseInt(args.id, 10);

const postIndex = posts.findIndex((post) => post.id === id);

if (postIndex !== -1) {

const post = posts[postIndex];

const updatedPost = {

...post,

...args,

};

});

posts.splice(postIndex, 1, updatedPost);

pubsub.publish("post", {

post: {

mutation: "UPDATED",

data: updatedPost,

}

});

});

### deletePost

The deletePost method deletes a post from the array if the ID exists. Afterwards, a new event called DELETED is published with the deleted post data:

deletePost(parent, args, { pubsub }) {

const id = parseInt(args.id, 10);

const isPostExists = posts.findIndex((post)=> post.id === id);

if (isPostExists === -1) {

throw new Error('Post does not exist!');

}

// splice will return the index of the removed items from the array object

const [post] = posts.splice(isPostExists, 1);

pubsub.publish('post', {

post:{

mutation: 'DELETED',

data: post

}

});

return post;

});

## Coding our subscription object

Lastly, we’ll add our subscription object, which uses a pubsub.asyncIterator function to map the event underlying the source stream to a returned response stream. The asyncIterator takes the channel name through which the event across the app will be mapped out:

post:{

subscribe(parent, args, {pubsub}){

return pubsub.asyncIterator('post');

}

}

Now, the only file left is index.js. Add the following code to it:

const { GraphQLServer, PubSub } = require("graphql-yoga");

const typeDefs = require("./typeDefs");

const resolvers = require("./resolvers");

const pubsub = new PubSub();

const server = new GraphQLServer({

typeDefs,

resolvers,

context: {

pubsub,

},

});

const options = {

port: 3000,

};

server.start(options, ({ port }) => {

console.log(

`Graphql Server started, listening on port ${port} for incoming requests.`

);

});

We created a GraphQL server, passed all our files, then started the server. Finally, we’ll add a script to run our project in package.json:

"scripts": {

"start": "node src/index.js"

},

Open the terminal and run npm start. If everything is working well, you’ll see the following message:

Graphql Server started, listening on port 3000 for incoming requests.

Now, head over to the browser and type [http://localhost:3000](http://localhost:3000/), and you’ll see a GraphQL Playground instance. To confirm that everything is working correctly, run a getPosts query, as shown below:

query {

getPosts {

id,

body,

downvotes,

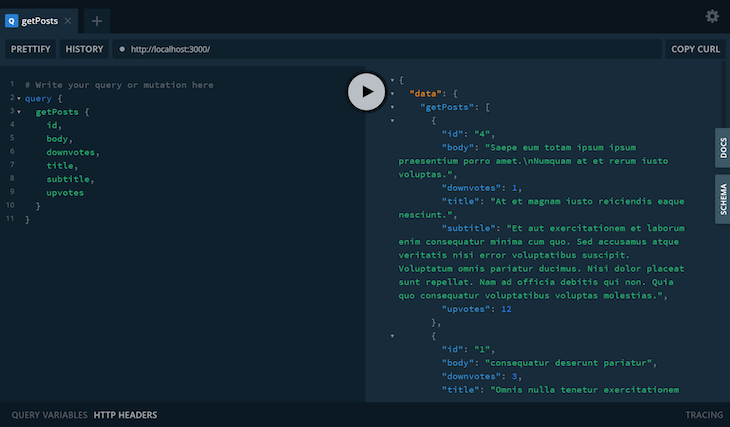
title,

subtitle,

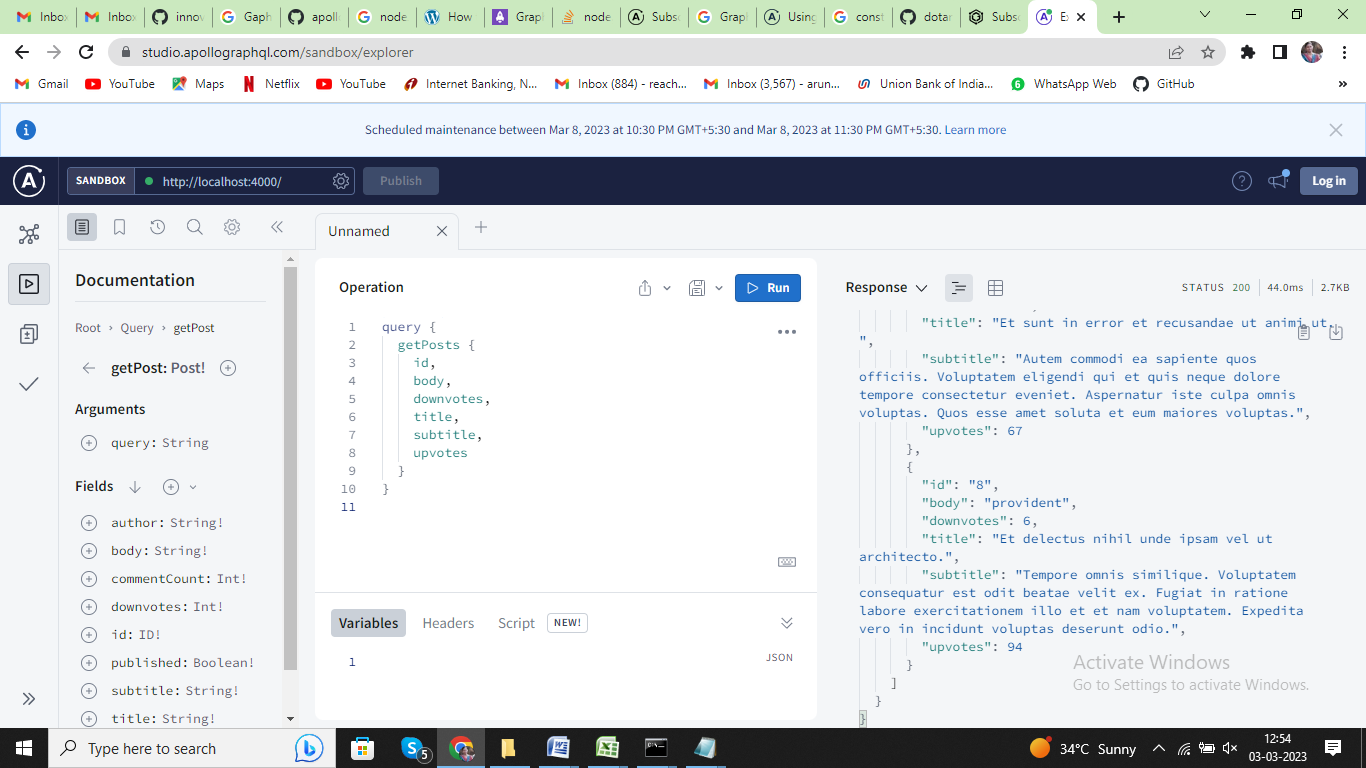
upvotes

}

}



Running a getPosts query in GraphQL



To start our subscription to the post changes, we’ll open up a new tab in GraphQL Playground and run the following code:

subscription{

post{

mutation

data{

id,

title,

subtitle,

body,

published

author,

upvotes,

downvotes,

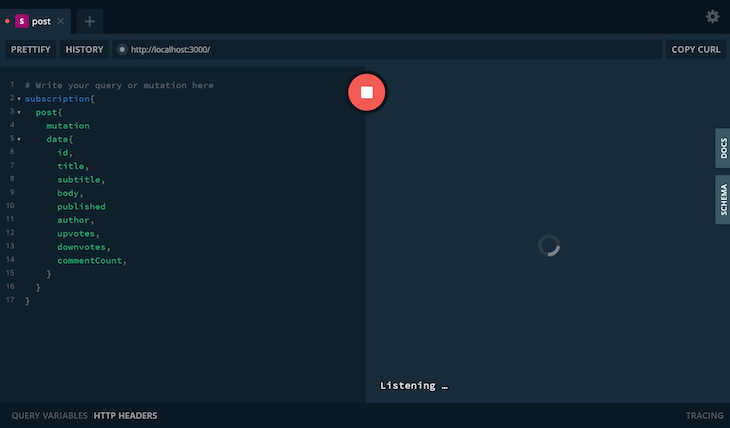
commentCount,

}

}

}

The code above enables us to add a subscription to our channel post and start listening for any event published in the channel:

Add GraphQL subscriptions to channel post

To see our GraphQL subscription in action, just perform any of the mutations in a new tab. For example:

mutation {

updatePost(

id: 8,

downvotes:10,

author: "InnovaptiveUser",

published: true,

subtitle: "testinng subtitle",

body: "testing body",

commentCount: 12,

upvotes: 4,

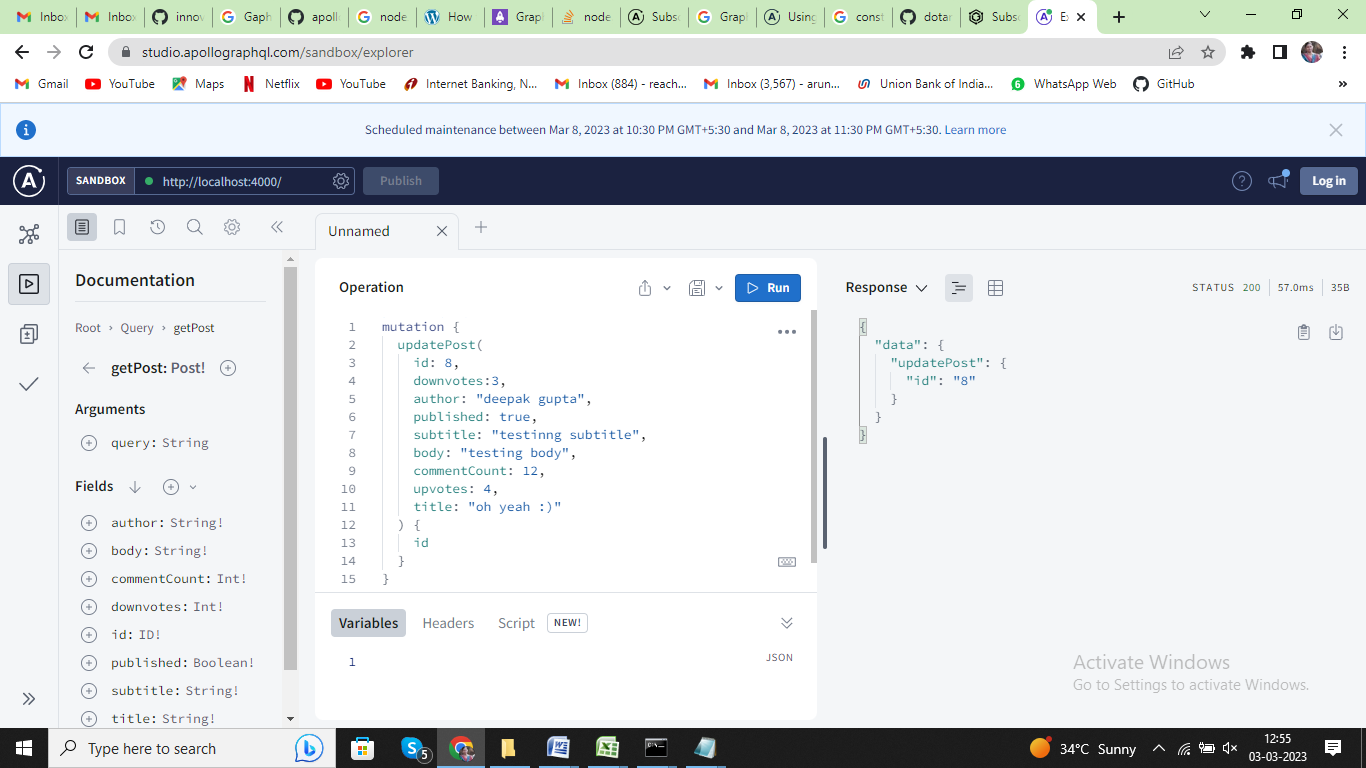
title: "oh yeah :)"

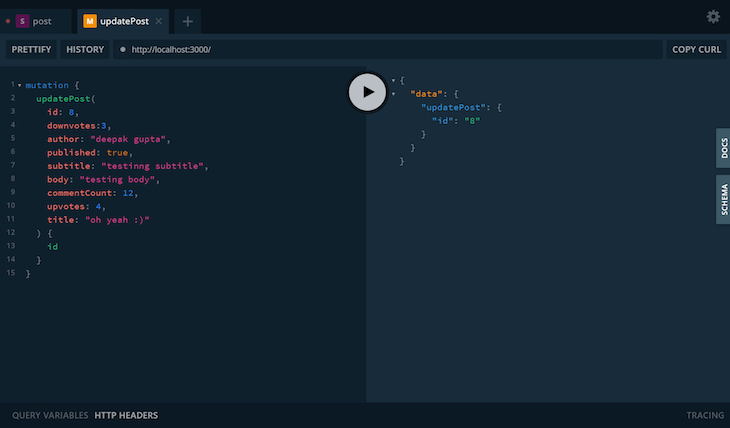
) {

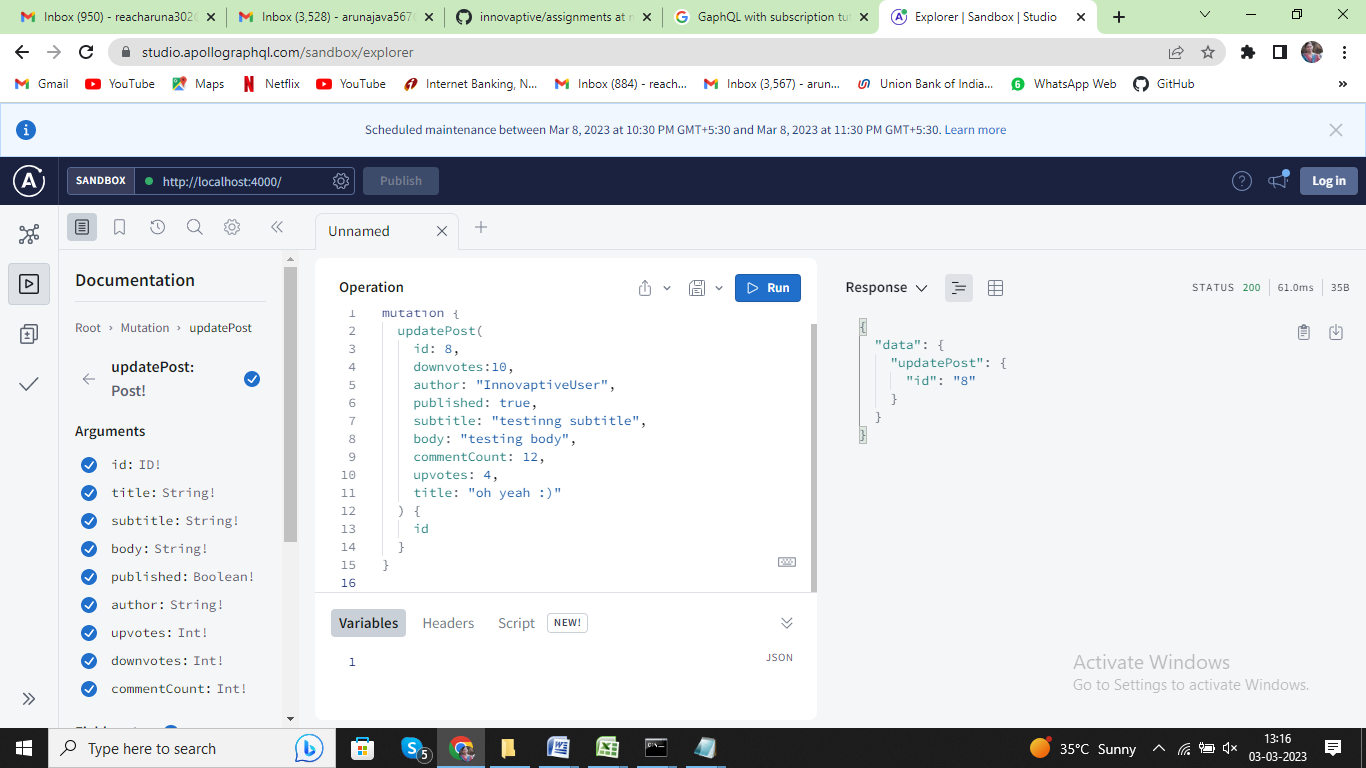
id

}

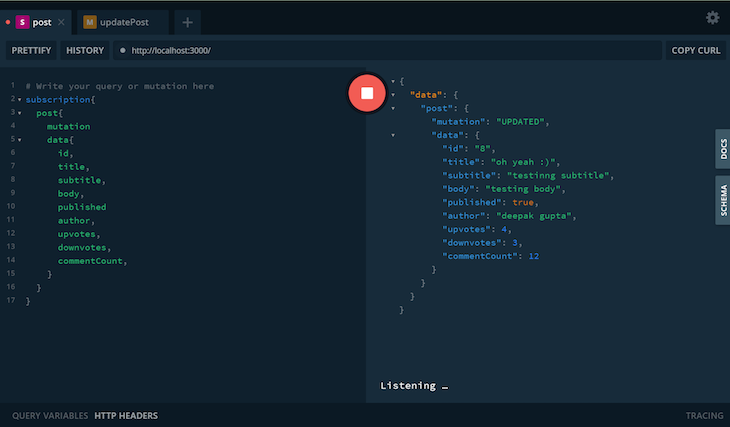
}







Navigate to the tab where the subscription query is running. Notice how the post response stream returned the data for the update event:

Response stream data in a GraphQL subscription

## Recapping the GraphQL subscription process

To wrap up our tutorial, let’s quickly recap the subscription process. The subscription is defined below in typeDefs.js:

type Subscription {

post: PostSubscriptionPayload!

}

type PostSubscriptionPayload {

mutation: String!

data: Post!

}

Use the pub/sub method provided by graphql-yoga to subscribe and publish. The pub/sub method can also facilitate mechanisms like EventEmitter:

const { GraphQLServer, PubSub } = require('graphql-yoga');

const typeDefs = require('./typeDefs');

const resolvers = require('./resolvers');

const pubsub = new PubSub();

const server = new GraphQLServer({

typeDefs,

resolvers,

context: {

pubsub,

},

});

Implement the subscription type resolver to map the event using pubsub.asyncIterator. Once we request a subscription from the GraphQL Playground, pubsub.asyncIteratorwill add our socket to its listening socket list and send back events while we call pubsub.publish:

Subscription: {

post: {

subscribe(parent, args, { pubsub }) {

return pubsub.asyncIterator('post');

},

},

}

Finally, call the pubsub.publish() method from the mutation channel:

pubsub.publish('post', {

post: {

mutation: 'UPDATED',

data: updatedPost,

},

});