Course Introduction

This course is a super fast introduction to setting up a GraphQL backend with Hasura.

In 30 mins, you will setup a Powerful, Scalable Realtime GraphQL Backend complete with Queries, Mutations, and Subscriptions. You will also learn how Hasura helps you integrate custom business logic (in any programming language), both as custom GraphQL APIs that you write yourself, and as Event Triggers that run asynchronously and are triggered by database events.

What will I learn?

This course will help you understand the different Hasura features, when and how to use them, and is an ideal starting point before diving into advanced topics.

1. Hasura Basics: Use Hasura to generate the realtime GraphQL API powered by a Postgres database
2. Postgres power: Use the power of Postgres to make efficient data transformations so that we don't have to do them in code
3. Authorization: Setup authorization so that app users can only run operations on data that they should be allowed to
4. Authentication: Integrate a JWT based auth provider (using Auth0) with Hasura
5. Remote schemas: Add a custom GraphQL resolver to create an API for fetching data that is not in the database
6. Event triggers: Run business logic on the backend when there are database events

What will we be building?

We will be building the backend of a realtime todo app. If you're interested in building the Frontend and are new to GraphQL, head to the GraphQL tutorials for different frontend frameworks.

Try this deployed version of the app, to see what features our backend will need to have: https://learn-hasura-todo-app.netlify.com/

What do I need to take this tutorial?

Just your browser!

However, be assured that everything you do via the browser in this tutorial can be done via code, can be version controlled and can work with any programming language you choose on the server side.

We've kept this course light on developer workflows and environment choices so that you

can focus on the key concepts and go on to set up your favorite tools and workflows.

How long will this tutorial take?

Less than 30 mins.

Deploy Hasura

Let's start by deploying Hasura.

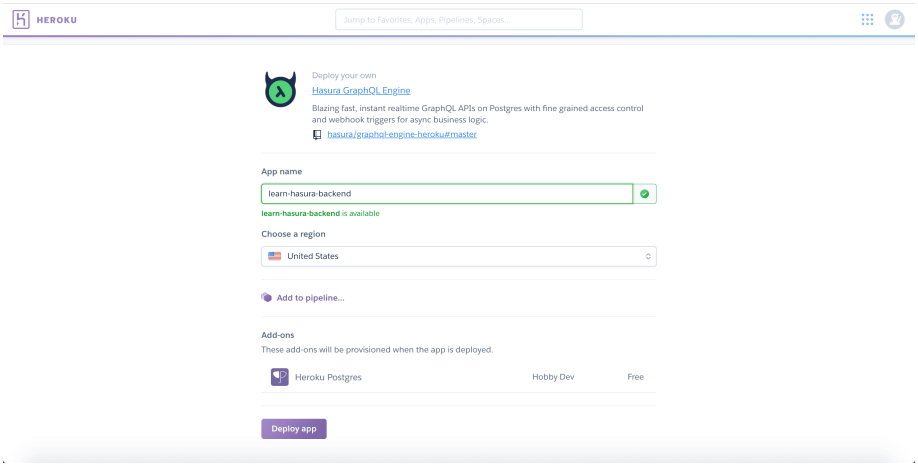
One-click deployment on Heroku

The fastest way to try Hasura out is via Heroku.

- Click on the following button to deploy GraphQL Engine on Heroku with the free Postgres add-on:

  Deploy to Heroku

  This will deploy Hasura GraphQL Engine on Heroku. A PostgreSQL database will be automatically provisioned along with Hasura. If you don't have an account on Heroku, you would be required to sign up. *Note*: It is free to signup and no credit-card is required.

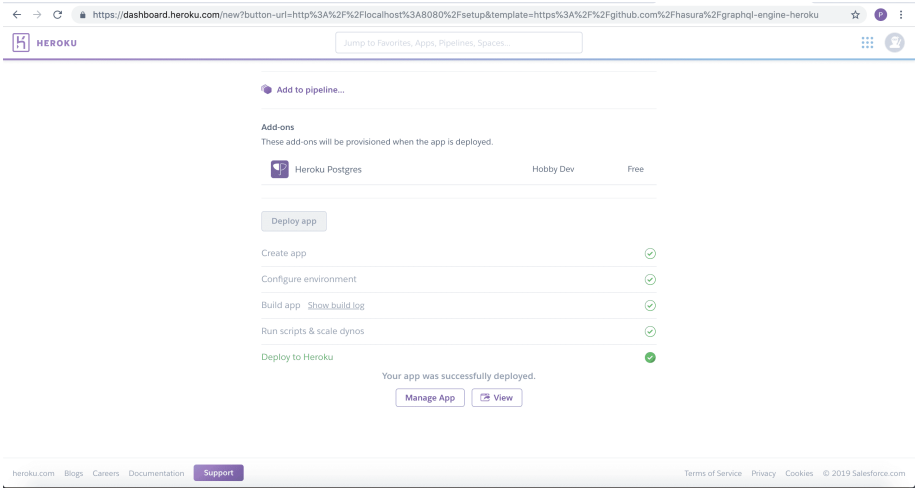


Deploy on Heroku

Type in the app name, select the region of choice and click on Deploy app button.

Hasura Console

Once the app is deployed, you should see the following on your Heroku dashboard.

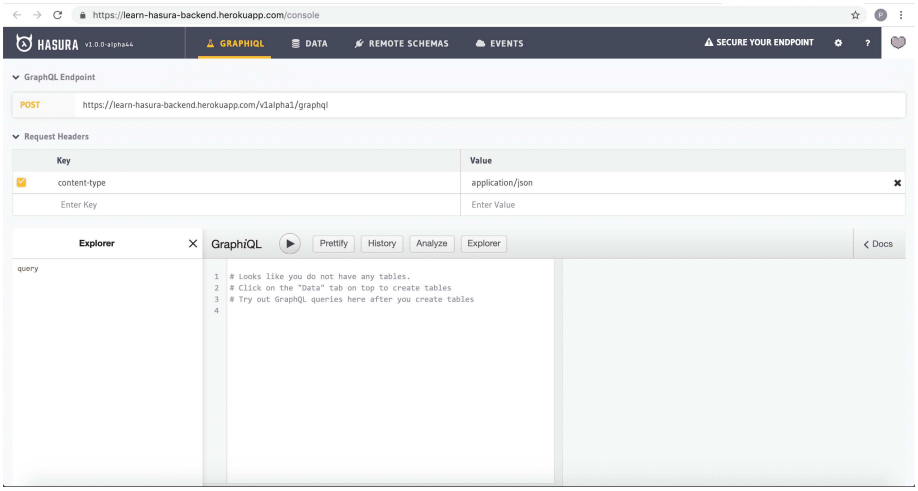


Hasura GraphQL on Heroku

- Open the Hasura console

  Click on the `View` button to open the app. Alternatively you can always visit `https://<app-name>.herokuapp.com` (*replace <app-name> with your app name*) to open the admin console.

It should look something like this:



Hasura Console

Great! You have now deployed Hasura GraphQL Engine and have the admin console ready to get started!

Basic Data Modelling

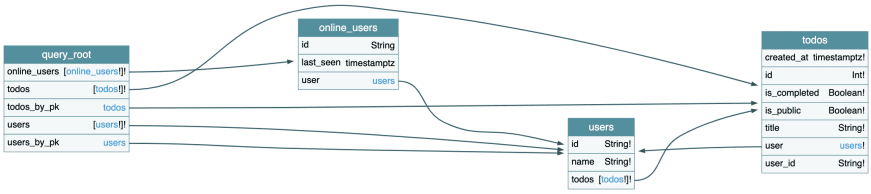
In this part of the course, we will build the data model for a realtime todo app. Our todo app will have the following features:

- Users can maintain personal todos
- Users can view public todos
- A list of current online users using the app
- Send email when a user signs up

Broadly this means that we have two models in this app: `users` and `todos`, each with its own set of properties.

We will go over them in the subsequent steps.

The final model looks like the following:



Schema Todo app

As we create tables using the console or directly on postgres, Hasura GraphQL engine creates GraphQL schema object types and corresponding query/mutation fields with resolvers automatically.

Create table users

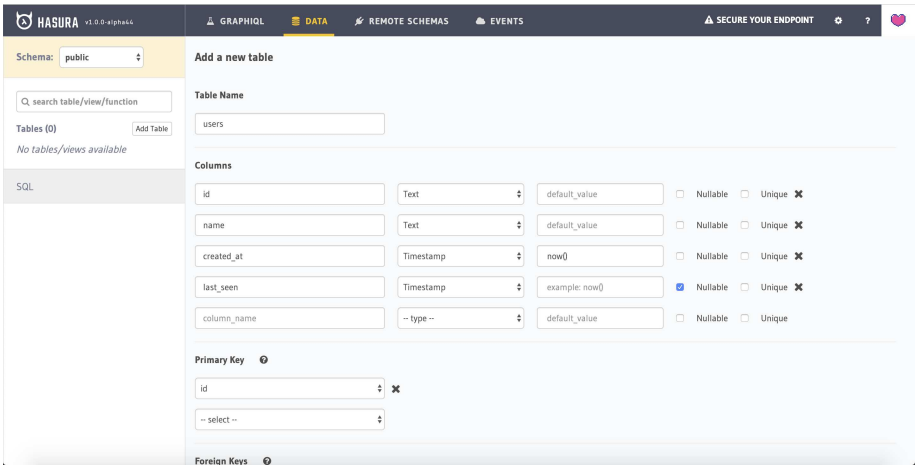
Let's get started by creating the table users.

The `users` table will have the following columns:

- `id` (type text),
- `name` (type text),
- `created_at` (type timestamp and default now())
- `last_seen` (type timestamp and nullable)

The columns are mostly self-explanatory. The `last_seen` column is used to store the latest timestamp of when the user was online.

In the Hasura Console, head over to the `Data` tab section and click on `Create Table`. Enter the values for creating the table as mentioned above.



Create table users

Once you are done, click on `Add Table` button to create the table.

Great! You have created the first table required for the app.

Try out GraphQL APIs

- Mutation
- Query
- Subscription

As you are aware that Hasura gives you Instant GraphQL APIs over Postgres, it can be tested on the table that we just created.

Let's go ahead and start exploring the GraphQL APIs for `users` table.

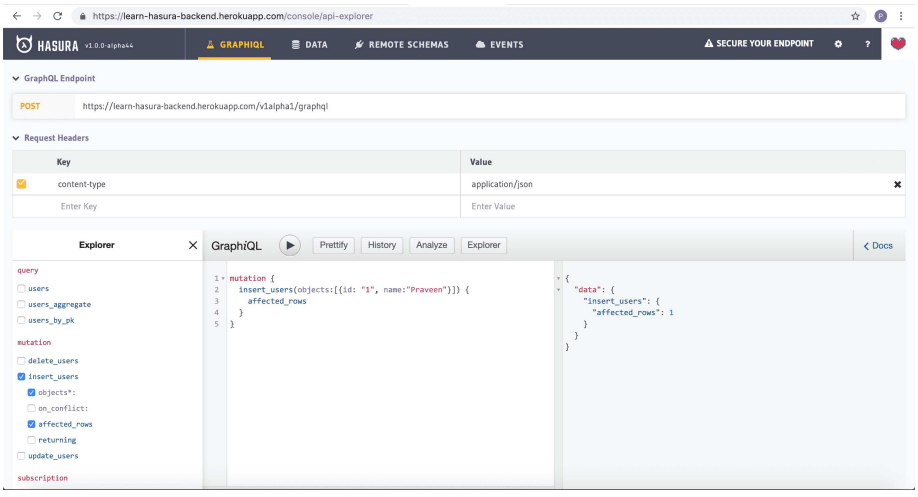
Mutation

Head over to Console -> GRAPHIQL tab and insert a user using GraphQL Mutations.

```
mutation {
  insert_users(objects:[{id: "1", name:"Praveen"}]) {
    affected_rows
  }
}
```

Click on the `Play` button on the GraphiQL interface to execute the query.

You should get a response looking something like this:



User Mutation

Great! You have now consumed the mutation query for the `users` table that you just created. Easy isn't it?

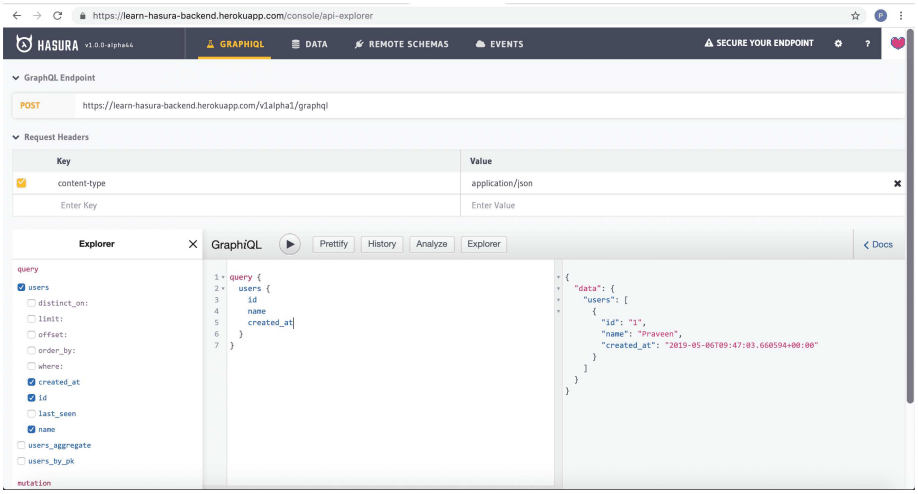
Tip: You can use the `Explorer` on the GraphiQL interface to generate the mutation in a few clicks.

Query

Now let's go ahead and query the data that we just inserted.

```
query {
  users {
    id
    name
    created_at
  }
}
```

You should get a response looking something like this:



User Query

Note that some columns like `created_at` have default values, even though you did not insert them during the mutation.

Subscription

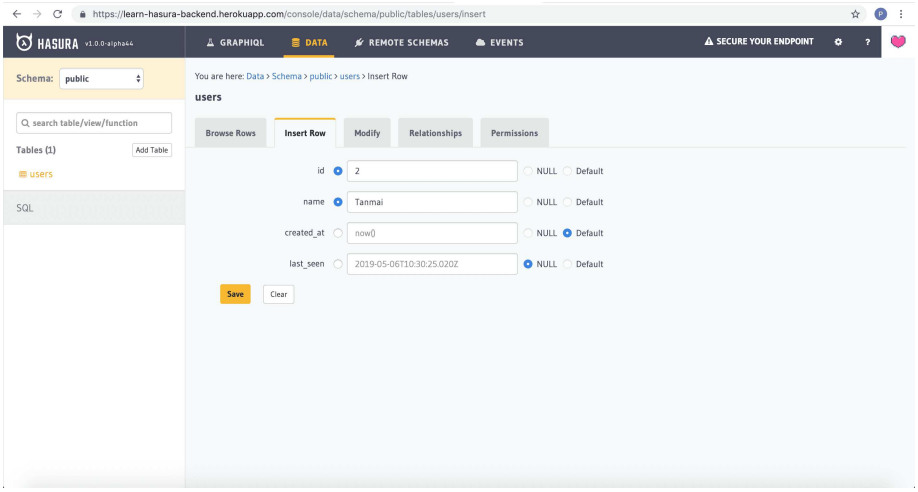
Let's run a simple subscription query over `users` table to watch for changes to the table.

```
subscription {
  users {
    id
    name
    created_at
  }
}
```

Initially the subscription query will return the existing results in the response.

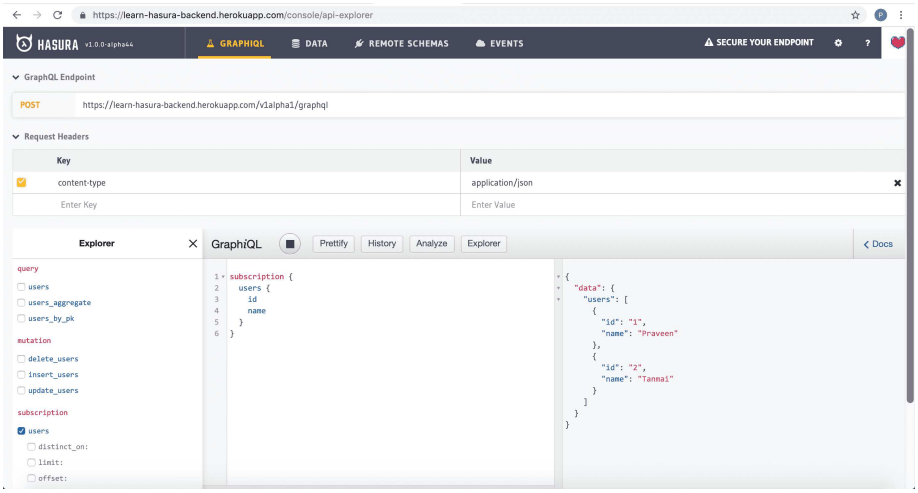
Now let's insert new data into the users table and see the changes appearing in the response.

In a new tab, Head over to Console -> DATA tab -> users -> Insert Row and insert another row.



Insert new user

And switch to the previous `GRAPHIQL` tab and see the subscription response returning 2 results.



User Subscription

An active subscription query will keep returning the latest set of results depending on the query.

Create table todos

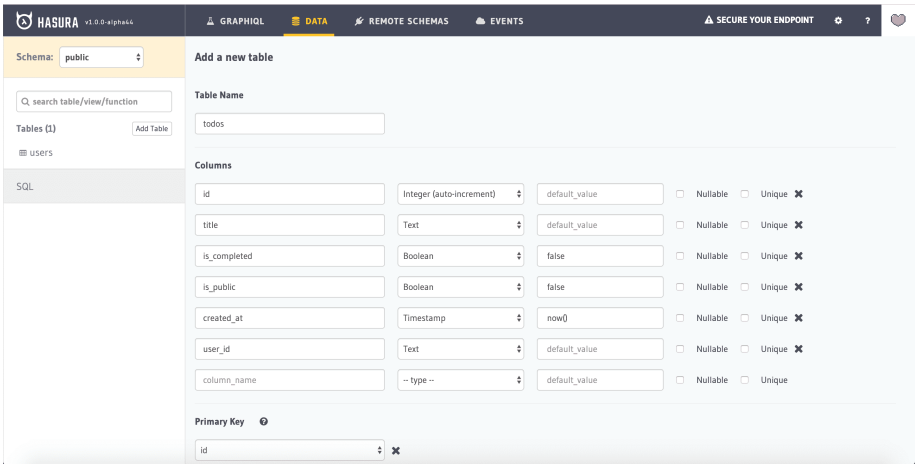
Now let's move on to creating the other model `todos`

The `todos` table will have the following columns:

- `id` (type integer;auto-increment),
- `title` (type text),
- `is_completed` (type boolean and default false)
- `is_public` (type boolean and default false)
- `created_at` (type timestamp and default now())
- `user_id` (type text)

The columns are mostly self-explanatory.

In the Hasura Console, head over to the `Data` tab section and click on `Create Table`. Enter the values for creating the table as mentioned above.



Create table users

Once you are done, click on `Add Table` button to create the table.

Try out GraphQL APIs

- Mutation
- Query
- Subscription

Similar to the `users` table, the `todos` table created in the previous step would have auto-generated GraphQL APIs for us to explore.

Let's go ahead and start exploring the GraphQL APIs for `todos` table.

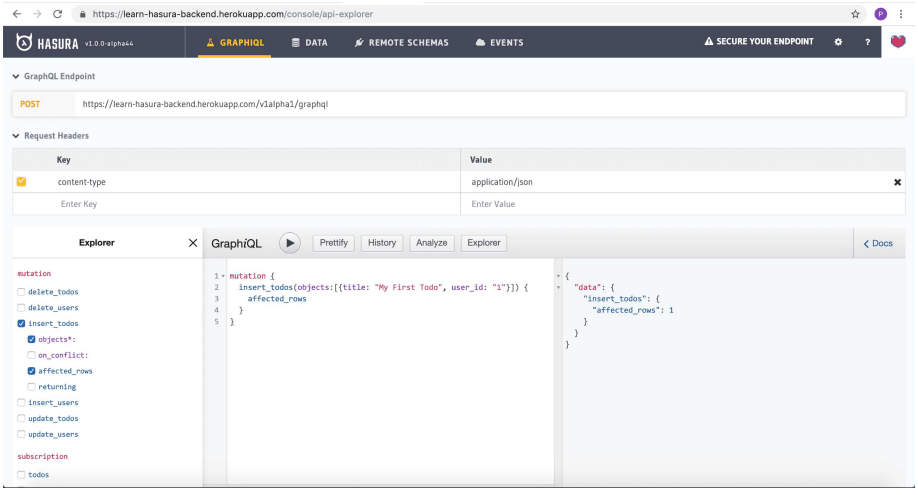
Mutation

Head over to Console -> GRAPHIQL tab and insert a todo using GraphQL Mutations.

```
mutation {
  insert_todos(objects:[{title: "My First Todo", user_id: "1"}]) {
    affected_rows
  }
}
```

Click on the `Play` button on the GraphiQL interface to execute the query.

You should get a response looking something like this:



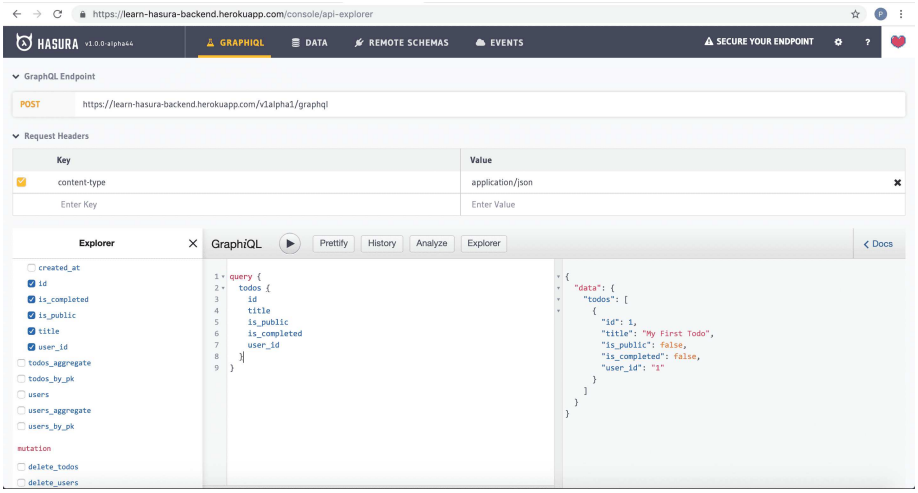
Todo Mutation

Query

Now let's go ahead and query the data that we just inserted.

```
query {
  todos {
    id
    title
    is_public
    is_completed
    user_id
  }
}
```

You should get a response looking something like this:



Todo Query

Note that some columns like `is_public`, `is_completed` have default values, even though you did not insert them during the mutation.

Subscription

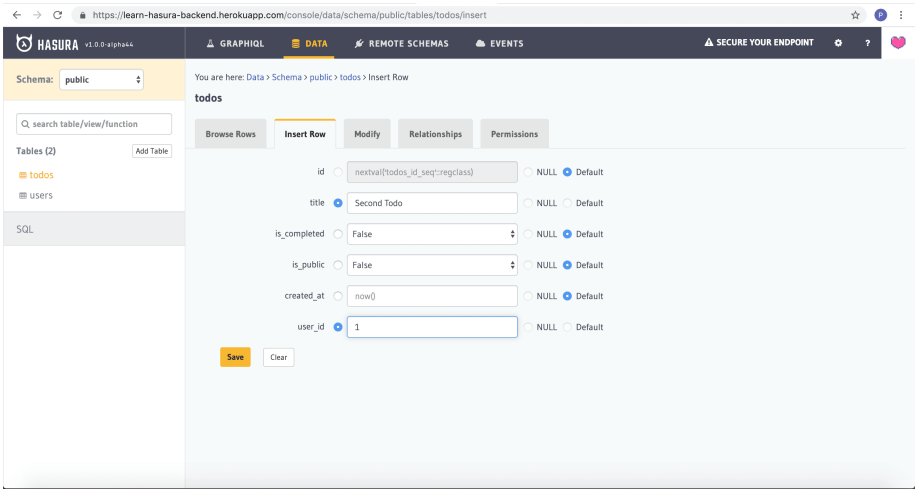
Let's run a simple subscription query over `todos` table to watch for changes to the table. In the above graphql query, replace `query` with `subscription`

```
subscription {
  todos {
    id
    title
    is_public
    is_completed
    user_id
  }
}
```

Initially the subscription query will return the existing results in the response.

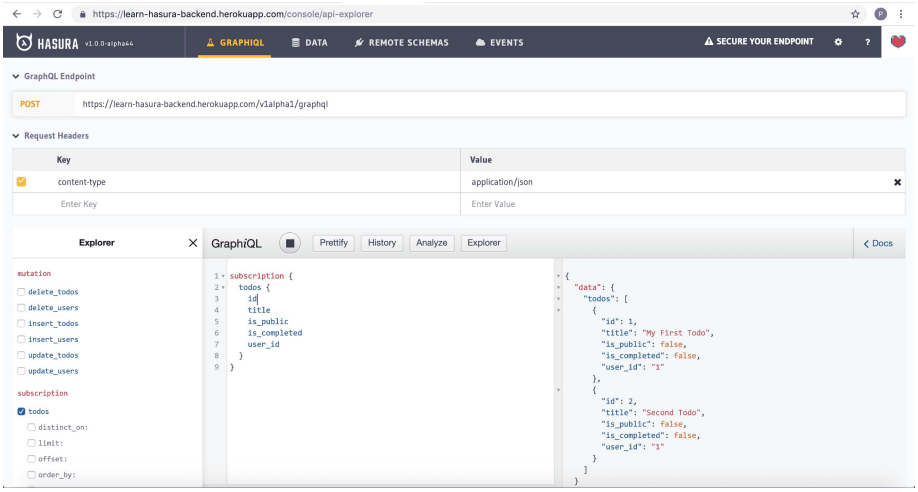
Now let's insert new data into the todos table and see the changes appearing in the response.

In a new tab, Head over to Console -> DATA tab -> todos -> Insert Row and insert another row.



Insert new todo

And switch to the previous GRAPHIQL tab and see the subscription response returning 2 results.



Todo Subscription

Relationships

Relationships enable you to make nested object queries if the tables/views in your database are connected.

GraphQL schema relationships can be either of

- object relationships (one-to-one)
- array relationships (one-to-many)

Object Relationships

Let's say you want to query `todos` and more information about the `user` who created it. This is achievable using nested queries if a relationship exists between the two. This is a one-to-one query and hence called an object relationship.

An example of such a nested query looks like this:

```
query {
  todos {
    id
    title
    user {
      id
      name
    }
  }
}
```

In a single query, you are able to fetch todos and its related user information. This can be very powerful because you can nest to any level.

Array Relationships

Let's look at an example query for array relationships.

```
query {
  users {
    id
    name
    todos {
      id
      title
    }
  }
}
```

In this query, you are able to fetch users and for each user, you are fetching the todos (multiple) written by that user. Since a user can have multiple todos, this would be an array relationship.

Relationships can be captured by foreign key constraints. Foreign key constraints ensure that there are no dangling data. Hasura Console automatically suggests relationships based on these constraints.

Though the constraints are optional, it is recommended to enforce these constraints for data consistency.

The above queries won't work yet because we haven't defined the relationships yet. But this gives an idea of how nested queries work.

Create Foreign Key

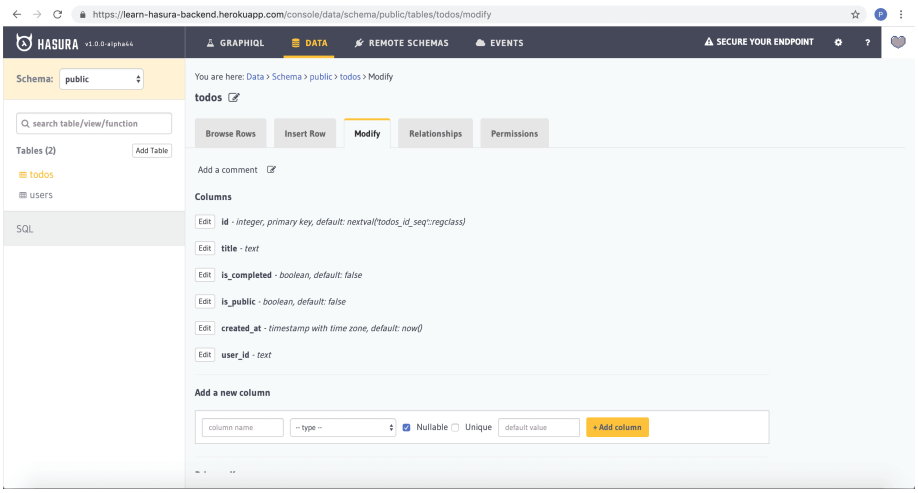
In the `todos` table, the value of `user_id` column must be ideally present in the `id` column of `users` table. Otherwise it would result in inconsistent data.

Postgres allows you to define foreign key constraint to enforce this condition.

Let's define one for the `user_id` column in `todos` table.

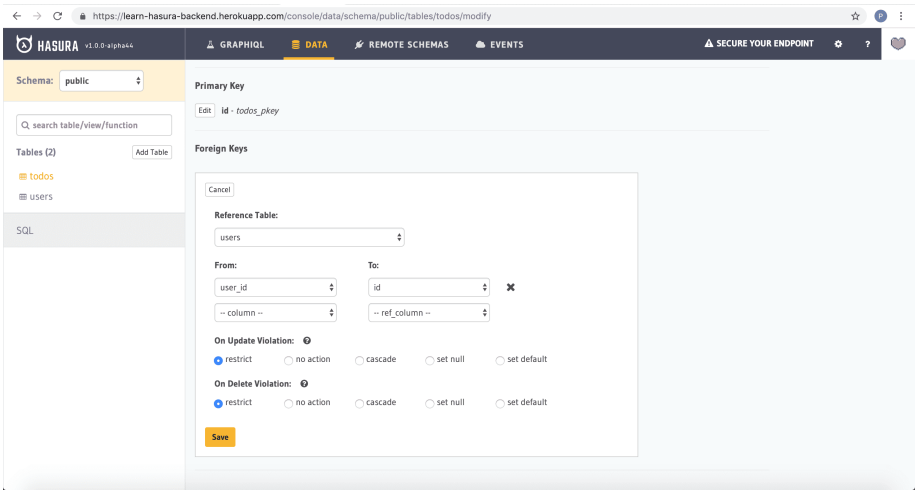
Head over to Console -> Data -> todos -> Modify page.

It should look something like this:



Todos Modify Page

Scroll down to `Foreign Keys` section at the bottom and click on `Add`.



user_id foreign key

- Select the Reference table as `users`
- Choose the From column as `user_id` and To column as `id`

We are enforcing that the user_id column of todos table must be one of the values of id in users table.

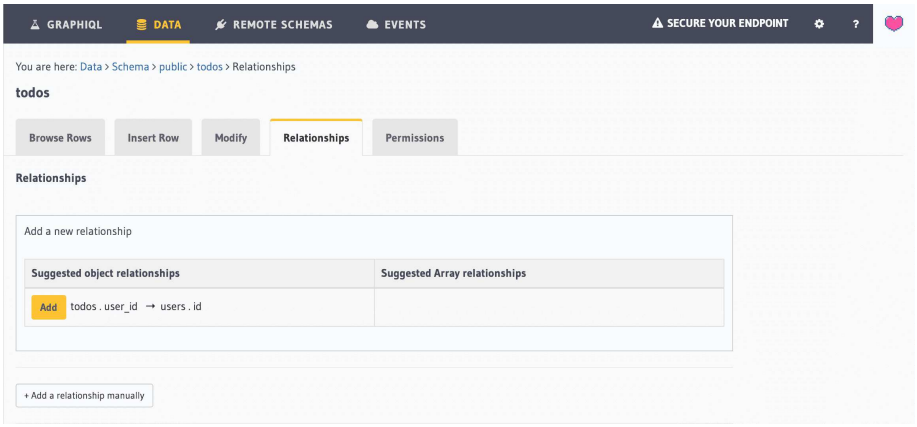
Click on `Save` to create the foreign key.

Great! Now you have ensured data consistency.

Create Relationship

Now that the foreign key constraint is created, Hasura Console automatically suggests relationships based on that.

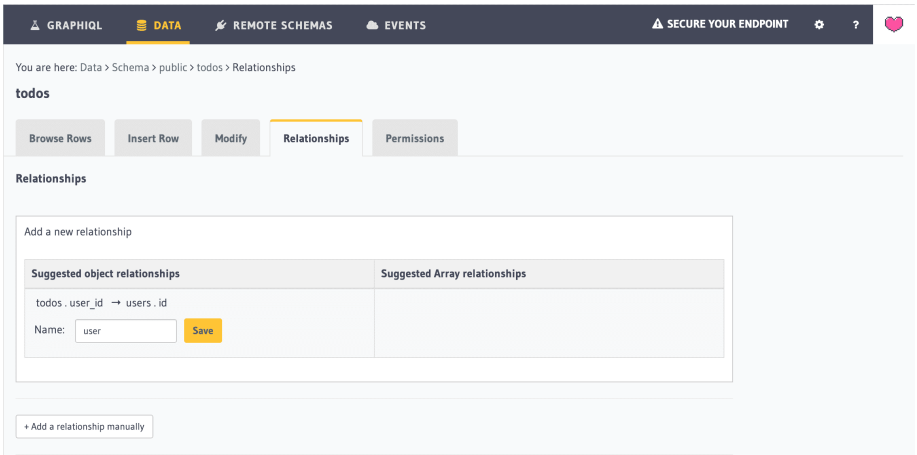
Head over to `Relationships` tab under `todos` table and you should see a suggested relationship like below:



Todos Relationships Page

Click on `Add` in the suggested object relationship.

Enter the relationship name as `user` (already pre-filled) and click on `Save`.



User Object Relationship

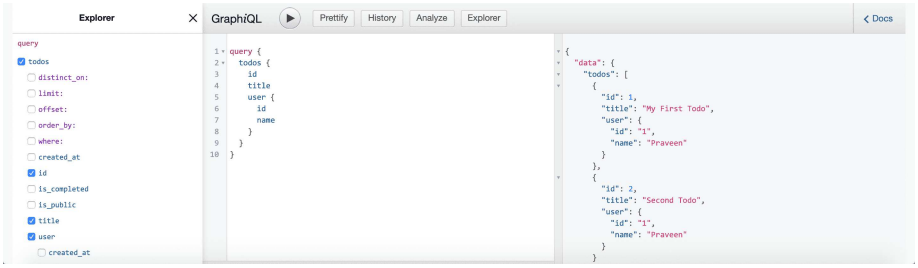
A relationship has now been established between todos and users table.

Try out Relationship Queries

Let’s explore the GraphQL APIs for the relationship created.

```
query {
  todos {
    id
    title
    user {
      id
      name
    }
  }
}
```

You can see the response in the following format:



relationship query

As you can see, in the same response, you are getting the results for the user’s information, exactly like you queried. This is a simple example of a one-to-one query/object relationship.

Data Transformations

One of the realtime features of the todo app is to display the list of online users. We need a way to fetch this information based on the value of `last_seen` which tells when the user was last online.

So far we were building tables and relationships. Postgres allows you to perform data transformations using: - Views - SQL Functions

In this example, we are going to make use of `Views`. This view is required by the app to find the users who have logged in and are online in the last 30 seconds.

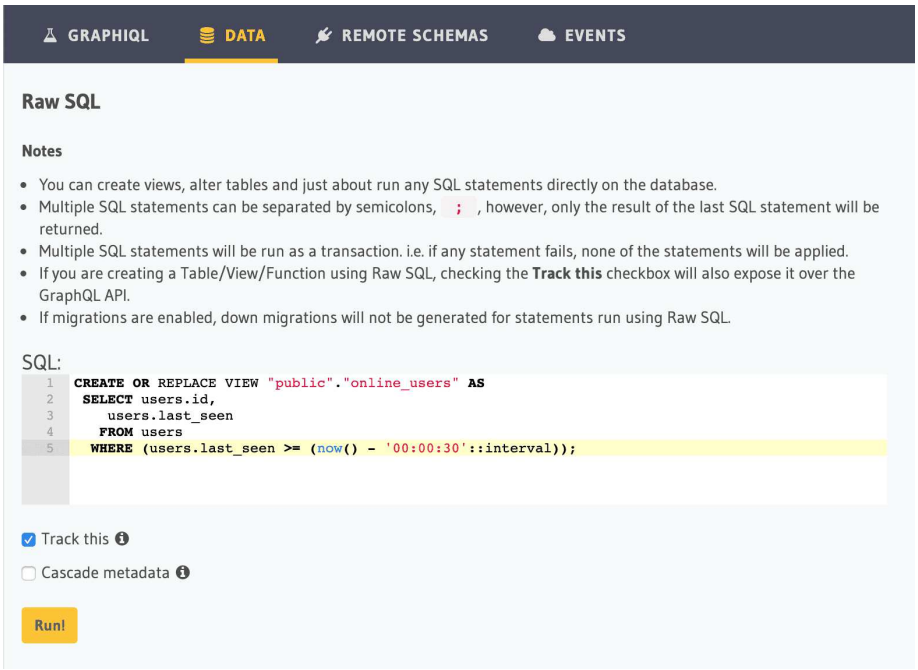
Create View

The SQL statement for creating this view looks like this:

```
CREATE OR REPLACE VIEW "public"."online_users" AS
 SELECT users.id,
    users.last_seen
   FROM users
  WHERE (users.last_seen >= (now() - '00:00:30'::interval));
```

Let's add this view and track the view with Hasura to be able to query it.

Head to Console -> Data -> SQL page.



Create view online_users

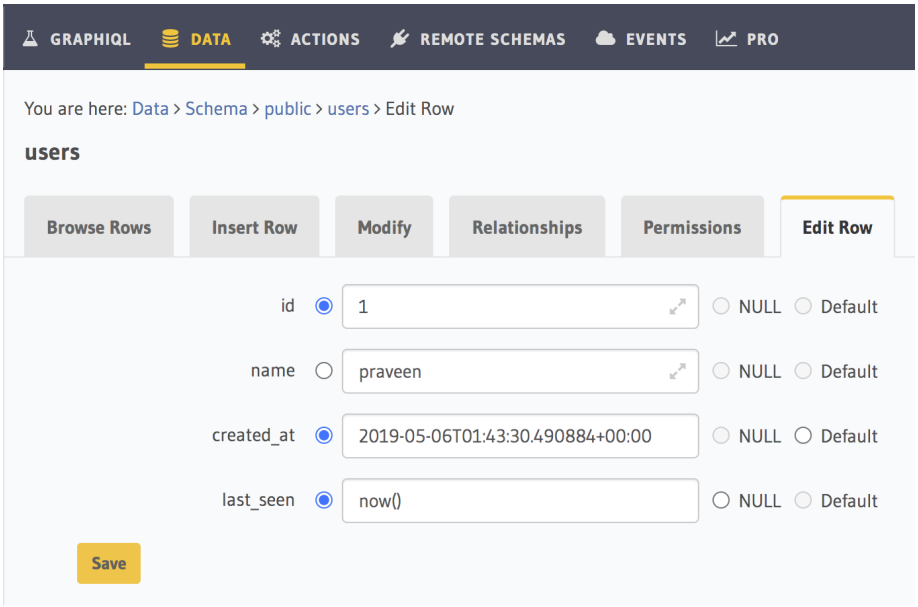
Click on `Run` to create the view.

Subscription to Online Users

Now let's test by making a subscription query to the `online_users` view.

```
subscription {
  online_users {
    id
    last_seen
  }
}
```

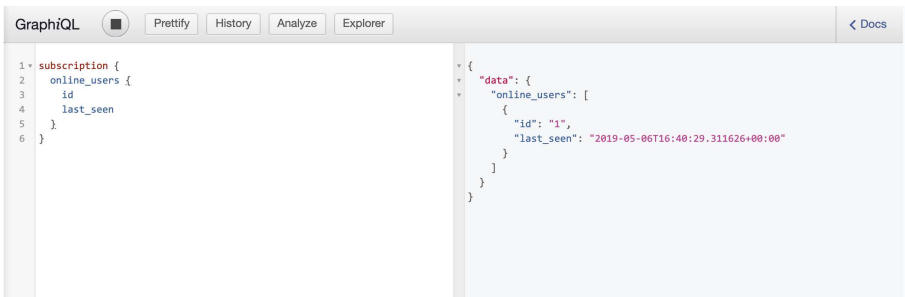
In another tab, update an existing user's `last_seen` value to see the subscription response getting updated.



Update users last_seen

Enter the value as `now()` for the `last_seen` column and click on `Save`.

Now switch back to the tab where your subscription query is running to see the updated response.



Subscription online users

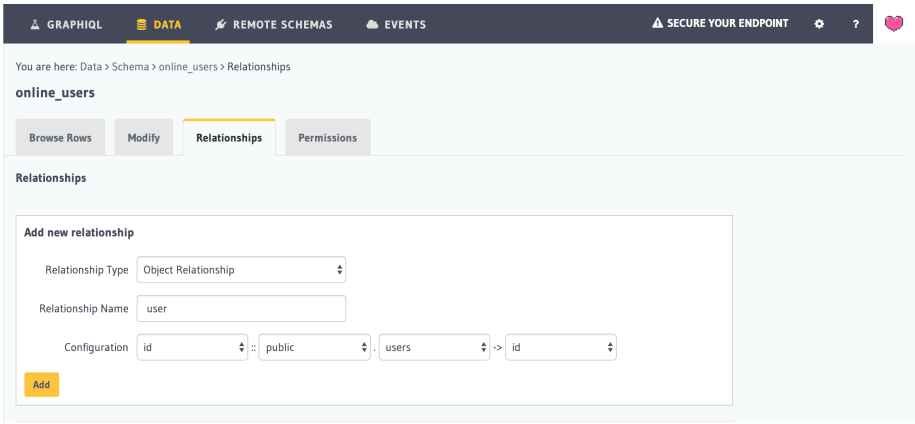
Create relationship to user

Now that the view has been created, we need a way to be able to fetch user information based on the `id` column of the view. Let's create a manual relationship from the view `online_users` to the table `users` using the `id column` of the view.

Head to Console -> Data -> online_users -> Relationships page.

Add new relationship by choosing the relationship type to be `Object Relationship`. Enter the relationship name as `user`. Select the configuration for current column as `id` and the remote table would be `users` and the remote column would be `id` again.

We are just mapping current view's id column to users table's id column to create the relationship.



create relationship from view

Let's explore the GraphQL APIs for the relationship created.

```
query {
  online_users {
    id
    last_seen
    user {
      id
      name
    }
  }
}
```

Great! We are completely done with data modelling for the app.

Authorization

In this part of the tutorial, we are going to define role based access control rules for each of the models that we created.

Access control rules help in restricting querying on a table based on certain conditions.

In this realtime todo app use-case, we need to restrict all querying only for logged in users. Also certain columns in tables do not need to be exposed to the user.

The aim of the app is to allow users to manage their own todos only but should be able to view all the public todos.

We will define all of these based on role based access control rules in the subsequent steps.

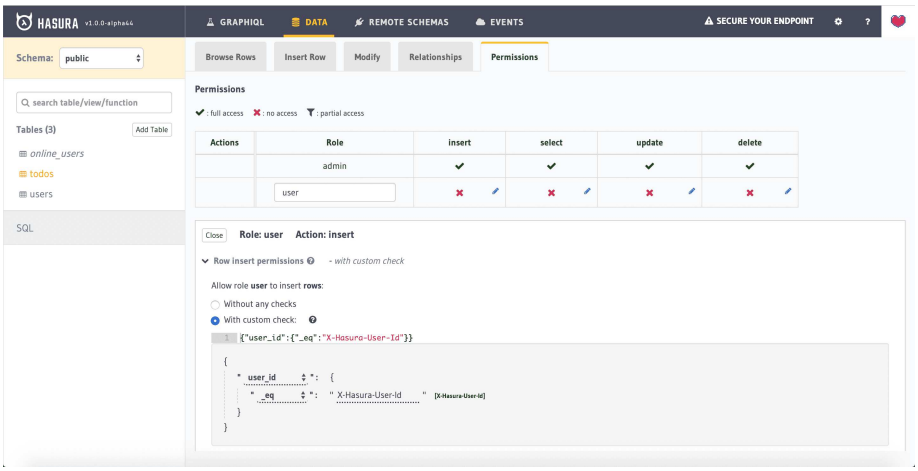
Setup todos table permissions

Head over to the Permissions tab under `todos` table to add relevant permissions.

Insert permission

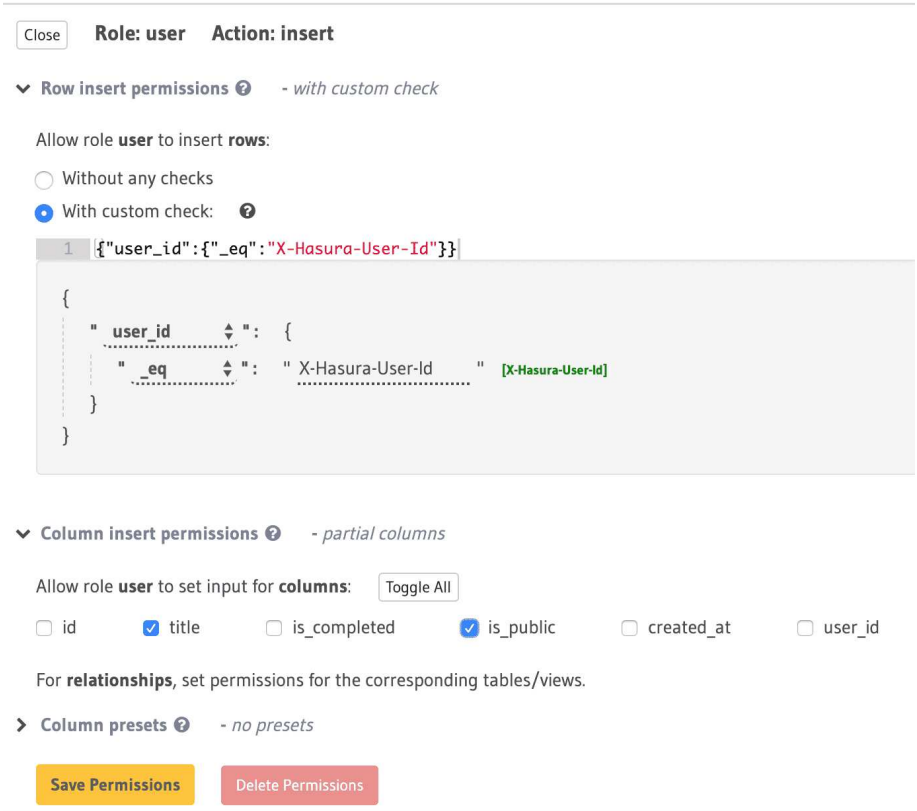
- In the enter new role textbox, type in “user”
- click on edit (pencil) icon for “insert” permissions. This would open up a section below which lets you configure custom checks and allow columns.
- In the custom check, choose the following condition

```
{"user_id":{"_eq":"X-Hasura-User-Id"}}
```



Todos row permission insert

Now under column insert permissions, select the `title` and `is_public` columns.



Todos insert column permission

Finally under column presets, select `user_id` from `from session variable` mapping to `X-HASURA-USER-ID`.

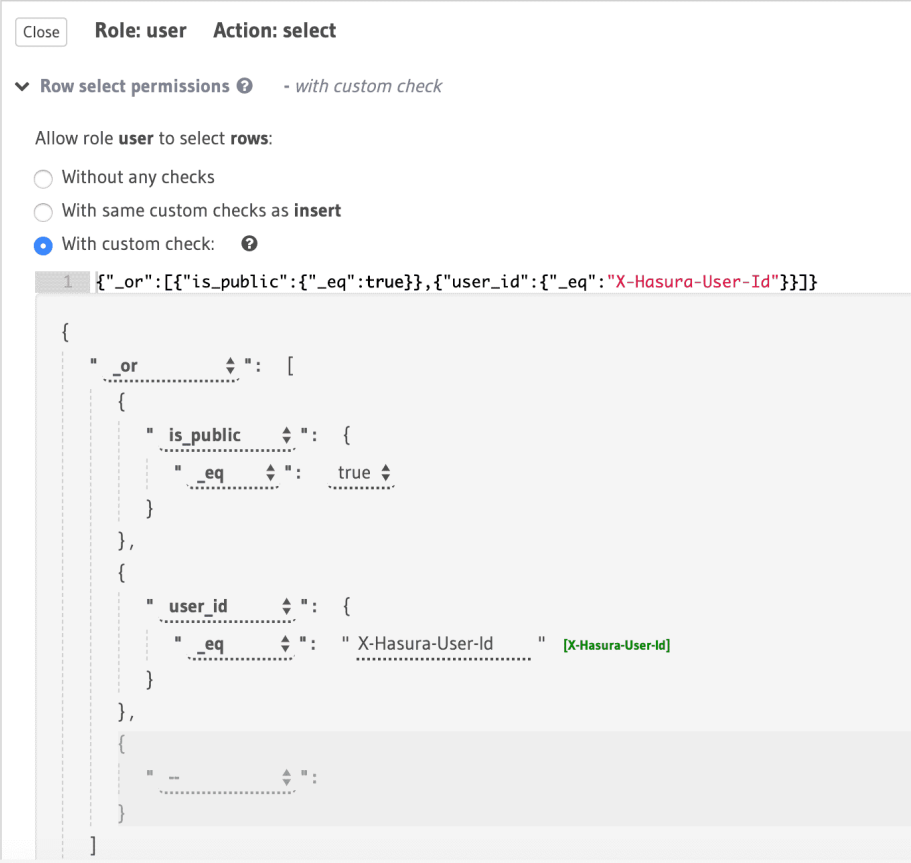
Note: Session variables are key-value pairs returned from the authentication service for each request. When a user makes a request, the session token maps to a `USER-ID`. This `USER-ID` can be used in a permission to show that inserts into a table are only allowed if the `user_id` column has a value equal to that of `USER-ID`, the session variable.

Click on `Save Permissions`.

Select permission

Now click on edit icon for “select” permissions. In the custom check, choose the following condition

```
{"_or":[{"is_public":{"_eq":true}},{"user_id":{"_eq":"X-Hasura-User-Id"}}]}
```



Todos select permission row

Under column select permissions, select all the columns.

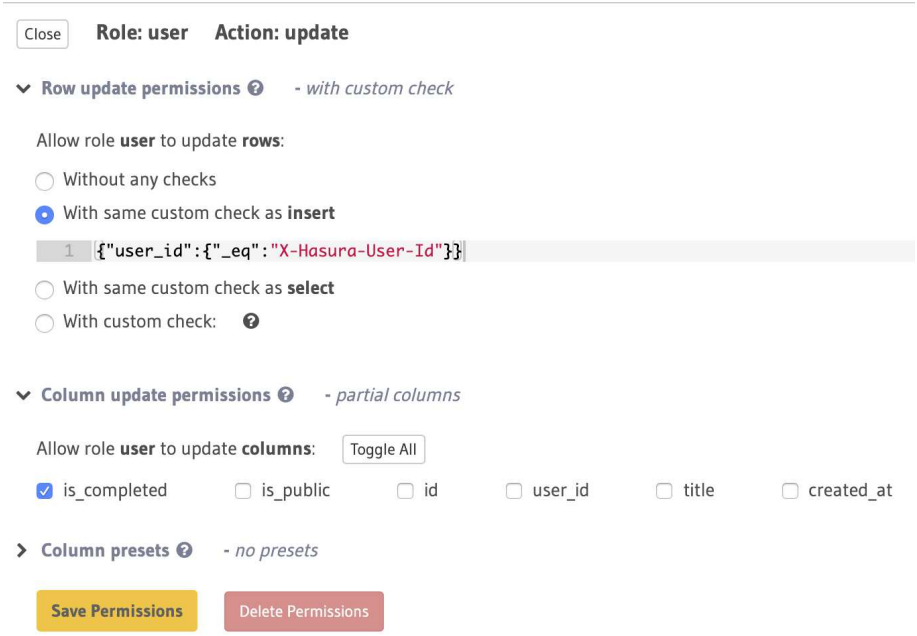
Todos select column permission

Click on `Save Permissions`

Update permission

Now click on edit icon for “update” permissions. In the custom check, choose `With same custom checks as insert`.

And under column update permissions, select the `is_completed` column.

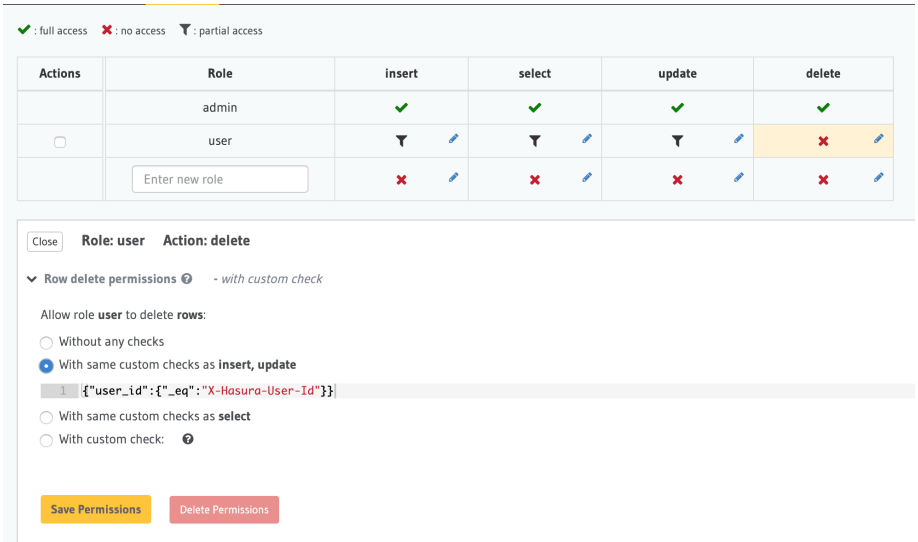


Todos update permission

Click on `Save Permissions` once done.

Delete permission

Finally for delete permission, under custom check, choose `With same custom checks as insert, update`.



Todos delete permission

Click on `Save Permissions` and you are done with access control for `todos` table.

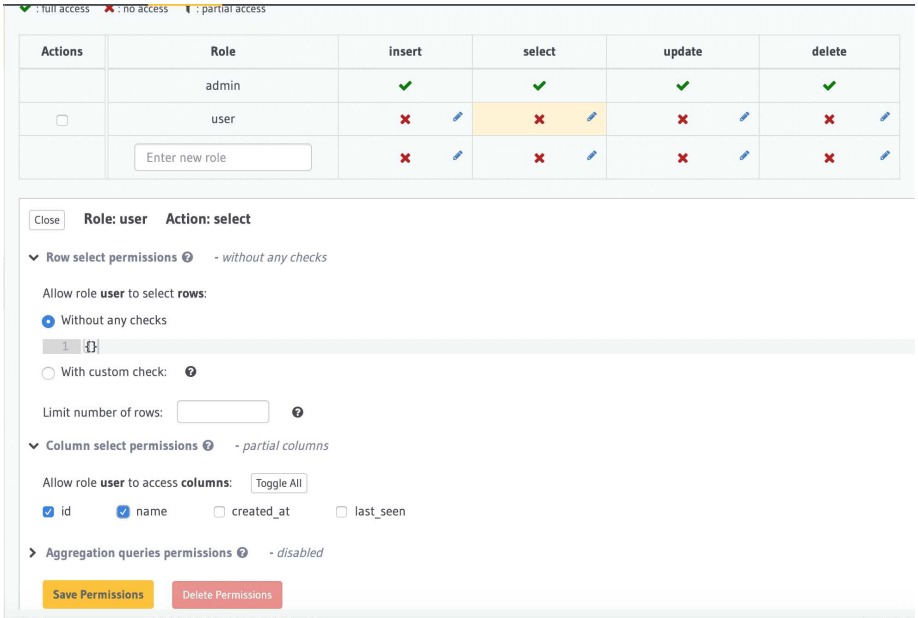
Setup users table permissions

We also need to allow select and update operations into `users` table. On the left sidebar, click on the `users` table to navigate to the users table page and switch to the Permissions tab.

Select permission

Click on the Edit icon (pencil icon) to modify the select permission for role user. This would open up a section below which lets you configure its permissions.

Here the users should be able to access every other user's `id` and `name` data.



users select permission

Click on `Save Permissions`

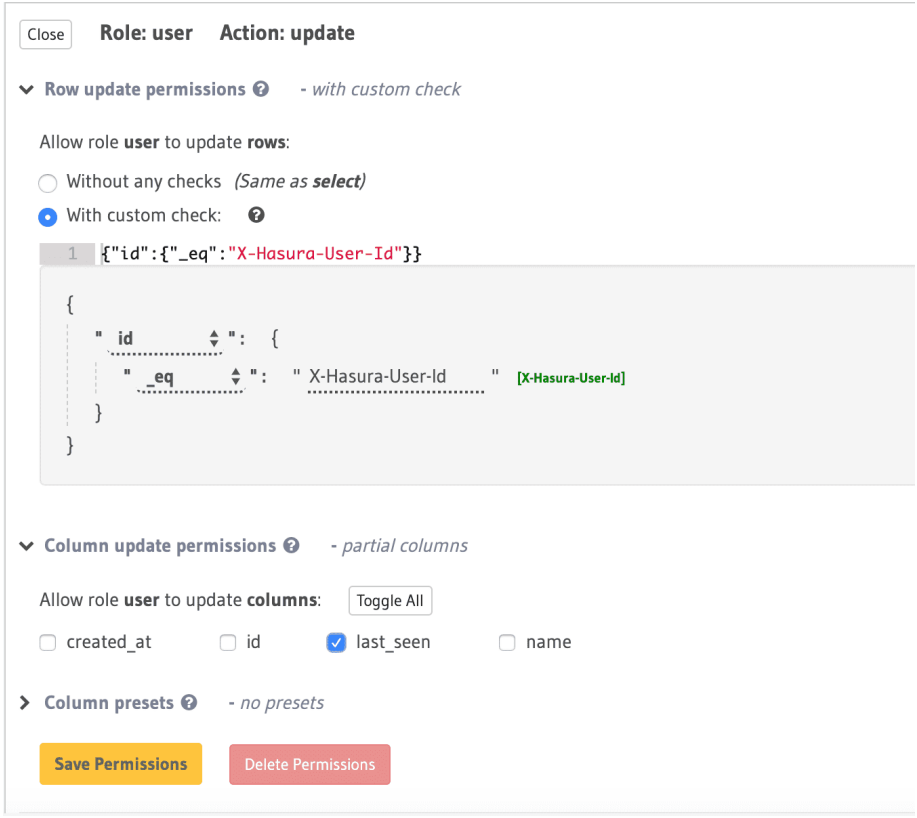
Update permission

The user who is logged in should be able to modify only his own record. So let's set that permission now.

In the Row update permission, under custom check, choose the following condition.

```
{"id":{"_eq":"X-Hasura-User-Id"}}
```

Under column update permissions, select `last_seen` column, as this will be updated from the frontend app.



users update permission

Click on `Save Permissions` and you are done with access control rules for `users` table.

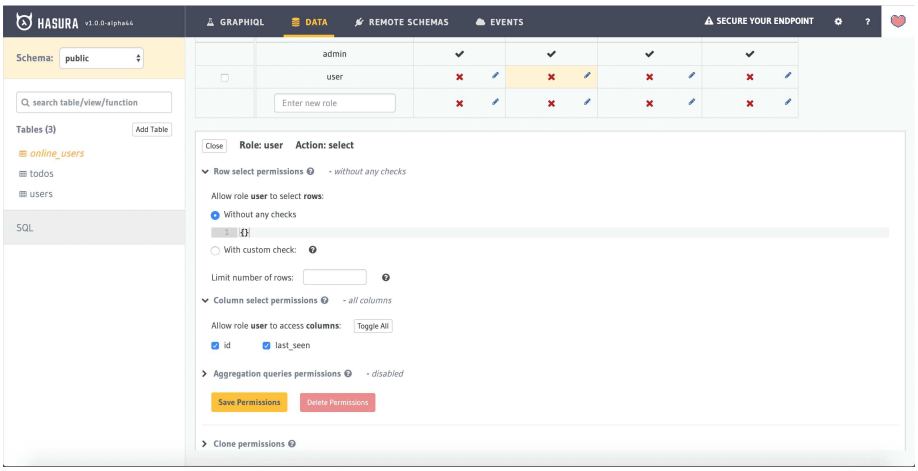
Setup online_users view permissions

Head over to the Permissions tab under `online_users` view to add relevant permissions.

Select permission

Here in this view, we only want the user to be able to select data and not do any mutations. Hence we don't define any permission for insert, update or delete.

For Row select permission, choose `Without any checks` and under Column select permission, choose both the columns `id` and `last_seen`.



online users permission

Click on `Save Permissions`. You have completed all access control rules required for the realtime todo app.

Authentication

In this part, we will look at how to integrate an Authentication provider.

The realtime todo app needs to be protected by a login interface. We are going to use Auth0 as the identity/authentication provider for this example.

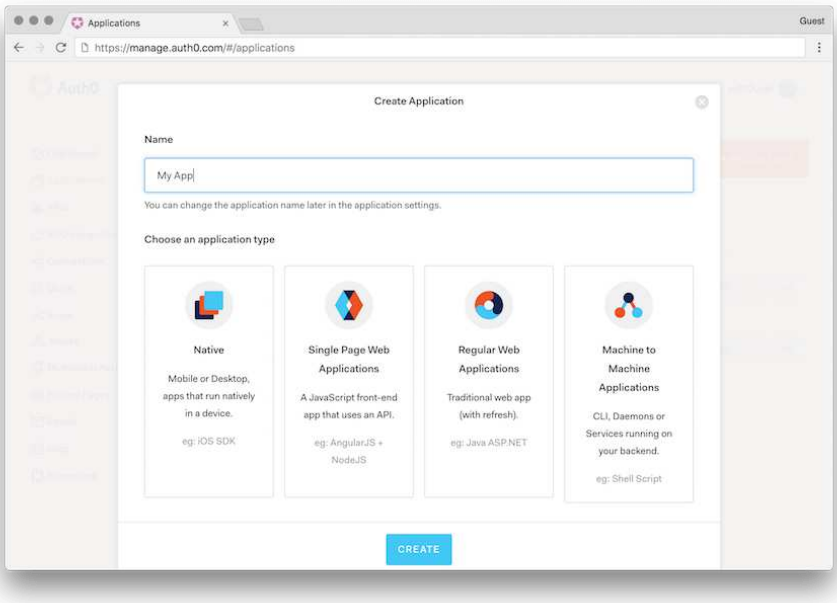
Note: Auth0 has a free plan for up to 7000 active users.

The basic idea is that, whenever a user authenticates with Auth0, the client app receives a token which can be sent in the `Authorization` headers of all GraphQL requests. Hasura GraphQL Engine would verify if the token is valid and allow the user to perform appropriate queries.

Let's get started!

Create Auth0 App

1. Navigate to the Auth0 Dashboard
2. Signup / Login to the account
3. Create a new tenant.
4. Click on the `Applications` menu option on the left and then click the `+ Create Application` button.
5. In the Create Application window, set a name for your application and select `Single Page Web Applications`. (Assuming the frontend app will be an SPA built on react/vue etc)



Create Auth0 App

6. In the settings of the application, we will add appropriate (e.g: http://localhost:3000/callback) URLs as Allowed Callback URLs and Allowed Web Origins. We can also add domain specific URLs as well for the app to work. (e.g: https://myapp.com/callback).

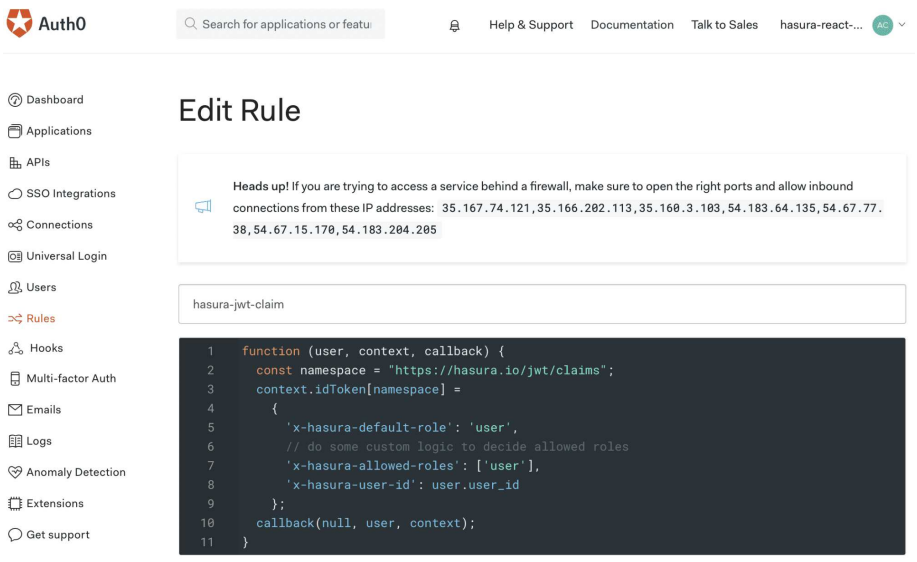
This would be the URL of the frontend app which you will deploy later. You can ignore this, for now. You can always come back later and add the necessary URLs.

Rules for Custom JWT Claims

Custom claims inside the JWT are used to tell Hasura about the role of the caller, so that Hasura may enforce the necessary authorization rules to decide what the caller can and cannot do. In the Auth0 dashboard, navigate to Rules.

Add the following rule to add our custom JWT claims under `hasura-jwt-claim`:

```
function (user, context, callback) {
  const namespace = "https://hasura.io/jwt/claims";
  context.idToken[namespace] =
    {
      'x-hasura-default-role': 'user',
      // do some custom logic to decide allowed roles
      'x-hasura-allowed-roles': ['user'],
      'x-hasura-user-id': user.user_id
    };
  callback(null, user, context);
}
```

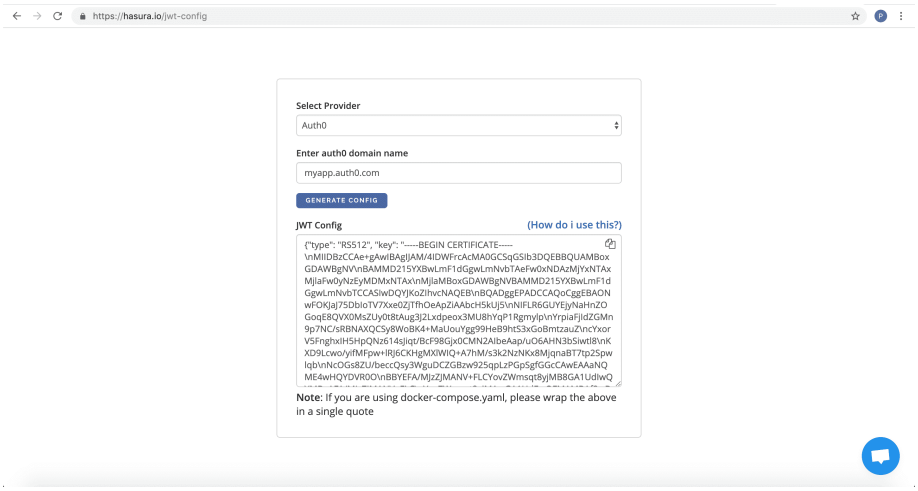


Custom JWT Claims Rule

Connect Hasura with Auth0

In this part, you will learn how to connect Hasura with the Auth0 application that you just created in the previous step.

We need to configure Hasura to use the Auth0 public keys. An easier way to generate the config for JWT is to use the following link - https://hasura.io/jwt-config



jwt-config

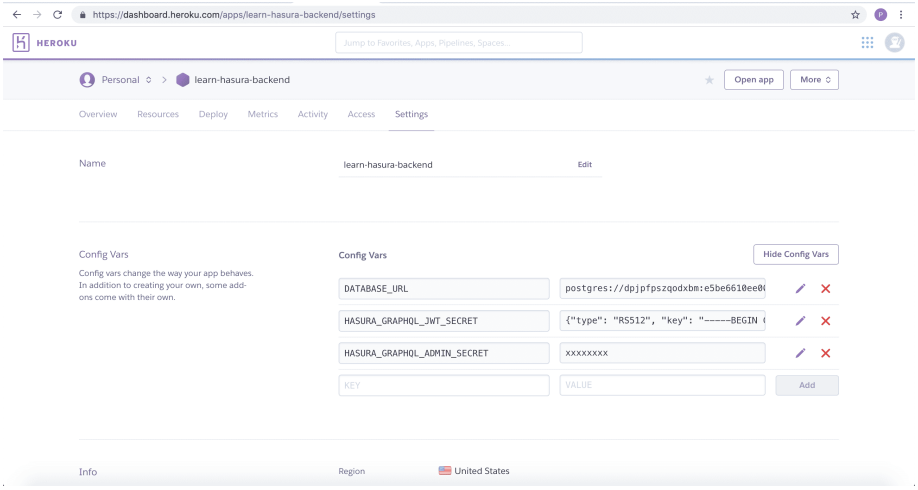
The generated configuration can be used as the value for environment variable `HASURA_GRAPHQL_JWT_SECRET`.

Since we have deployed Hasura GraphQL Engine on Heroku, let's head to Heroku dashboard to configure the admin secret and JWT secret.

Open the "Settings" page for your Heroku app, add a new Config Var called `HASURA_GRAPHQL_JWT_SECRET`, and copy and paste the generate JWT configuration into the value box.

Next, create a new Config Var called `HASURA_GRAPHQL_ADMIN_SECRET` and enter a secret key to protect the GraphQL endpoint. (Imagine this as the password to your GraphQL server).

You should end up with something like the following:



Heroku ENV Config

Great! Now your Hasura GraphQL Engine is secured using Auth0.

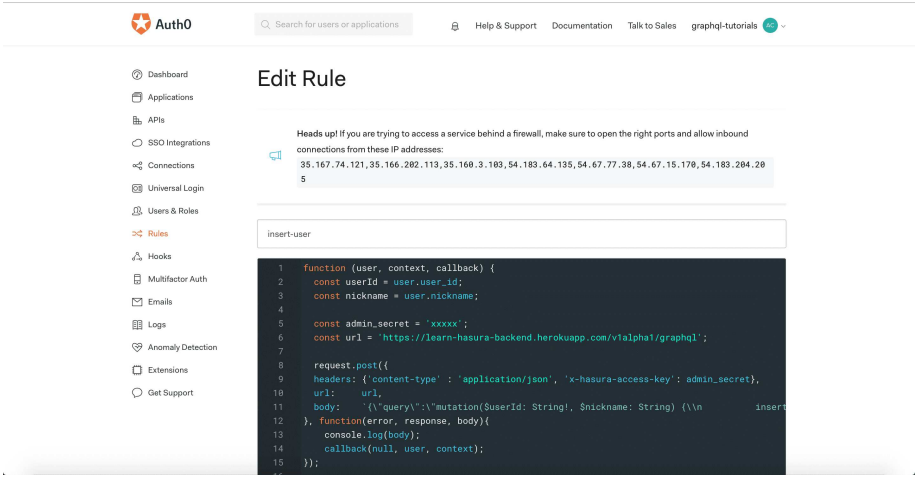
Sync Users with Rules

Auth0 has rules that can be set up to be called on every login request. We need to set up a rule in Auth0 which allows the users of Auth0 to be in sync with the users in our database. The following code snippet allows us to do the same. Again using the Rules feature, create a new blank rule and paste in the following code snippet:

```
function (user, context, callback) {
  const userId = user.user_id;
  const nickname = user.nickname;

  const admin_secret = "xxxx";
  const url = "https://learn-hasura-backend.herokuapp.com/v1/graphql";

  request.post({
      headers: {'content-type' : 'application/json', 'x-hasura-admin-secret': admin_
      url:   url,
      body:    `{\"query\":\"mutation($userId: String!, $nickname: String) {\\n
  }, function(error, response, body){
       console.log(body);
       callback(null, user, context);
  });
}
```



Auth0 insert rule

Note: Modify `x-hasura-admin-secret` and `url` parameters appropriately according to your app. Here we are making a simple request to make a mutation into `users` table.

That's it! This rule will now be triggered on every successful signup or login, and we insert or update the user data into our database using a Hasura GraphQL mutation.

The above request performs a mutation on the users table with the `id` and `name` values.

Test with Auth0 Token

Hasura is configured to be used with Auth0. Now let's test this setup by getting the token from Auth0 and making GraphQL queries with the Authorization headers to see if the permissions are applied.

To get a JWT token for testing,

1. Copy this URL - `https://auth0-domain.auth0.com/login?client=client_id&protocol=oauth2&response_type=token%20id_token&redirect_uri=callback_uri&scop` and update the URL as given below:

- Replace auth0-domain with the one we created in the previous steps.
- Replace client_id with Auth0 application's client_id.
- Replace callback_uri with `http://localhost:3000/callback` for testing. You don't need anything to run on localhost:3000 for this to work.

Make sure http://localhost:3000/callback has been added under Allowed Callback URLs in the Auth0 app settings.

2. Now try entering the updated URL in the browser. It should take you to the Auth0 login screen.

Note: In case logging in gives an error mentioning OIDC-conformant clients, try disabling OIDC Conformant setting (https://auth0.com/docs/api-auth/tutorials/adoption/oidc-conformant) under Advanced Settings -> OAuth.

2. After successfully logging in, you will be redirected to https://localhost:3000/callback#xxxxxxxx&id_token=yyyyyyy. This page will be a 404, unless you are running some other server on that port locally.

3. We care only about the URL parameters. Extract the `id_token` value from this URL. This is the JWT.

jwt-token-auth0-url

4. Test this JWT in jwt.io debugger.

The debugger should give you the decoded payload that contains the JWT claims that have been configured for Hasura under the key `https://hasura.io/jwt/claims`. Now inside this object, the role information will be available under `x-hasura-role` key and the user-id information will be available under `x-hasura-user-id` key.

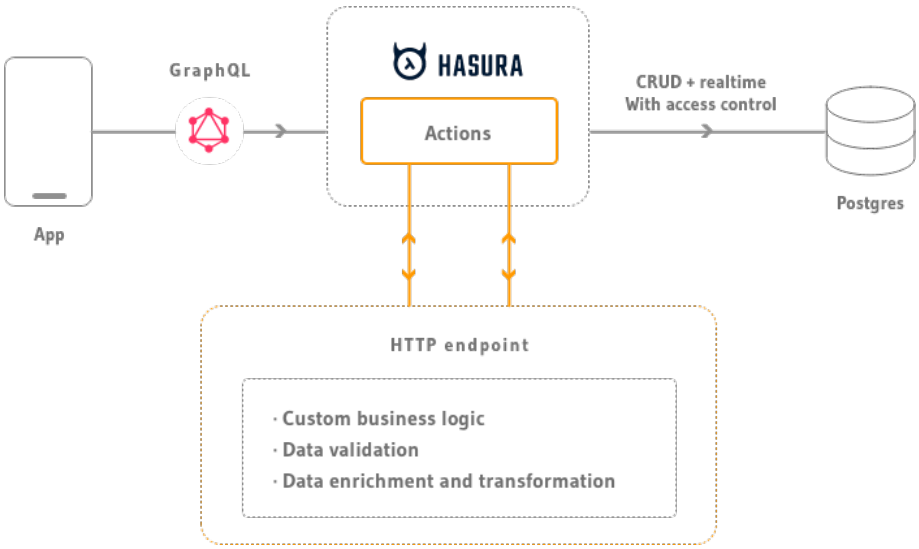
Custom Business Logic

Hasura gives you CRUD + realtime GraphQL APIs with authorization & access control. However, there are cases where you would want to add custom/business logic in your app. For example, in the todo app that we are building, before inserting todos into the public feed we want to validate the text for profanity.

Custom business logic can be handled in a few flexible ways in Hasura:

Actions (Recommended)

Actions are a way to extend Hasura's schema with custom business logic using custom queries and mutations. Actions can be added to Hasura to handle various use cases such as data validation, data enrichment from external sources and any other complex business logic.



Actions architecture

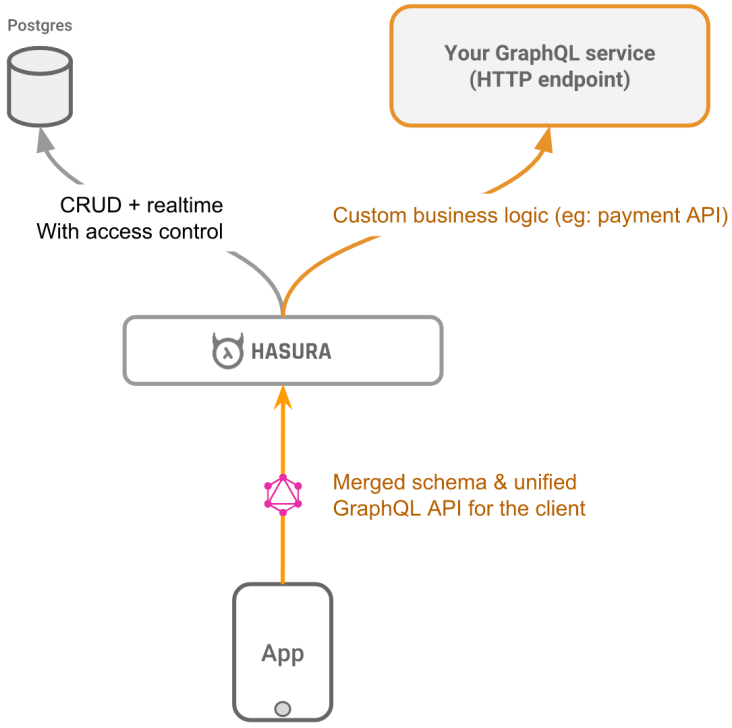
Actions can be either a Query or a Mutation.

- `Query Action` - If you are querying some data from an external API or doing some validations / transformations before sending back the data, you can use a Query Action.
- `Mutation Action` - If you want to perform data validations or do some custom logic before manipulating the database, you can use a Mutation Action.

Remote Schemas

Hasura has the ability to merge remote GraphQL schemas and provide a unified GraphQL

API. Think of it like automated schema stitching. This way, we can write custom GraphQL resolvers and add it as a remote schema.



Remote schema architecture

If you are choosing between Actions and Remote Schemas, here's something to keep in mind: - Use Remote Schemas if you have a GraphQL server or if you're comfortable building one yourself. - Use Actions if you need to call a REST API

Event Triggers

Hasura can be used to create event triggers on tables in the Postgres database. Event triggers reliably capture events on specified tables and invoke webhooks to carry out any custom logic. After a mutation operation, triggers can call a webhook asynchronously.

Use case for the todo app

In the todo app backend that you have built, there are certain custom functionalities you may want to add:

If you want to fetch profile information from Auth0, you need to make an API call to Auth0 with the token. Auth0 only exposes a REST API and not GraphQL. This API has to be exposed to the GraphQL client.

We will add an Action in Hasura to extend the API. We will also see how the same thing can be done with a custom GraphQL server added as a Remote Schema.

- Get notified via email whenever a new user registers in your app. This is an asynchronous operation that can be invoked via Event trigger webhook.

We will see how these 2 use-cases can be handled in Hasura.

Creating Actions

Let's take the first use-case of fetching profile information from Auth0.

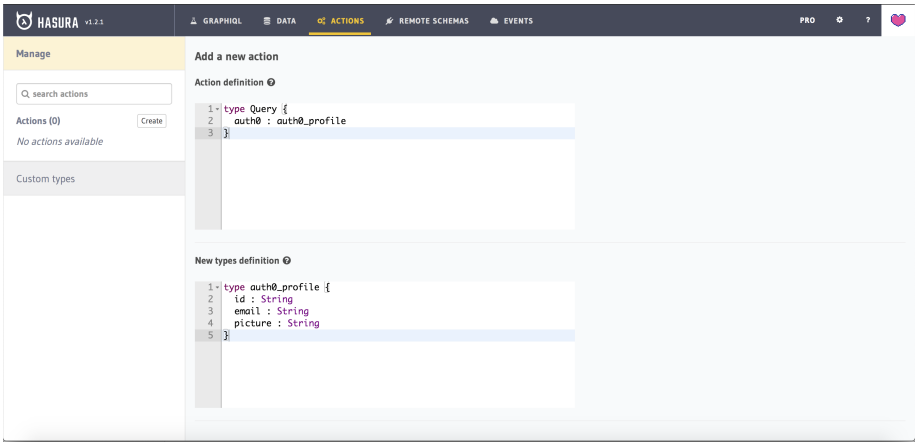
Ideally you would want to maintain a single GraphQL endpoint for all your data requirements.

To handle the use-case of fetching Auth0 profile information, we will write a REST API in a custom Node.js server. This could be written in any language/framework, but we are sticking to Node.js for this example.

Hasura can then merge this REST API with the existing auto-generated GraphQL schema and the client will be able to query everything using the single GraphQL endpoint.

Creating an action

On the Hasura Console, head to the `Actions` tab and Click on `Create` to create a new action.



Action definition

Action definition

We will need to define our Action and the type of action. Since we are just reading data from an API, we will use the Query type for this Action. The definition will have the name of the action (auth0 in this case), input arguments (none in this case) and the response type of the action (`auth0_profile` in this case).

```
type Query {
  auth0 : auth0_profile
}
```

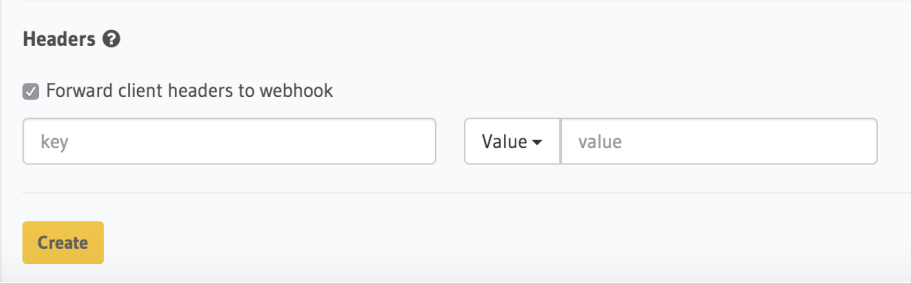
Types definition

We defined that the response type of the action is `auth0_profile`. So what do we want in return from the Auth0 API? We want the `id`, `email` and `picture` fields which aren't stored on our database so far.

```
type auth0_profile {
  id : String
  email : String
  picture : String
}
```

All three fields are of type String. Note that `auth0_profile` is an object type which has 3 keys (id, email and picture) and we are returning this object in our response.

We will change the Handler URL later once we write our REST API and deploy it on a public endpoint.



Create Action

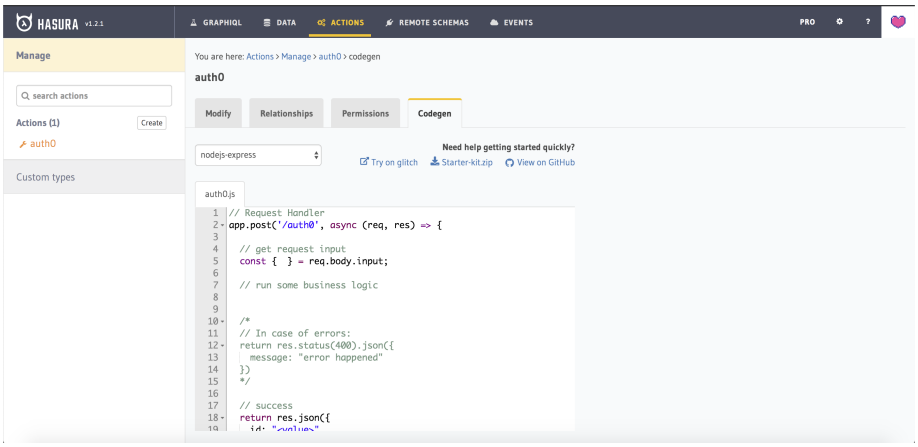
Click on `Create` once you are done configuring the above fields.

Write a REST API

Now that the Action has been created, let's write a REST API in a simple Node.js Express app that can later be configured for this Action.

Head to the `Codegen` tab to quickly get started with boilerplate code :)

Click on `Try on Glitch` to deploy a server. Glitch is a platform to build and deploy apps (Node.js) and is a quick way to test and iterate code on the cloud.



Action codegen

Now replace the contents of `src/server.js` with the following:

```
const express = require("express");
const bodyParser = require("body-parser");
const fetch = require('node-fetch');

const app = express();

const PORT = process.env.PORT || 3000;

app.use(bodyParser.json());

const getProfileInfo = (user_id) => {
    const headers = {'Authorization': 'Bearer '+process.env.AUTH0_MANAGEMENT_API_TOKE
    console.log(headers);
    return fetch('https://' + process.env.AUTH0_DOMAIN + '/api/v2/users/'+user_id,{
        .then(response => response.json())
}

app.post('/auth0', async (req, res) => {

  // get request input
  const { session_variables } = req.body;

  const user_id = session_variables['x-hasura-user-id'];
  // make a rest api call to auth0
  return getProfileInfo(user_id).then( function(resp) {
    console.log(resp);
    if (!resp) {
      return res.status(400).json({
        message: "error happened"
      })
    }
    return res.json({
      email: resp.email,
      picture: resp.picture
    })
  });

});

app.listen(PORT);
```

In the server above, let's break down what's happening:

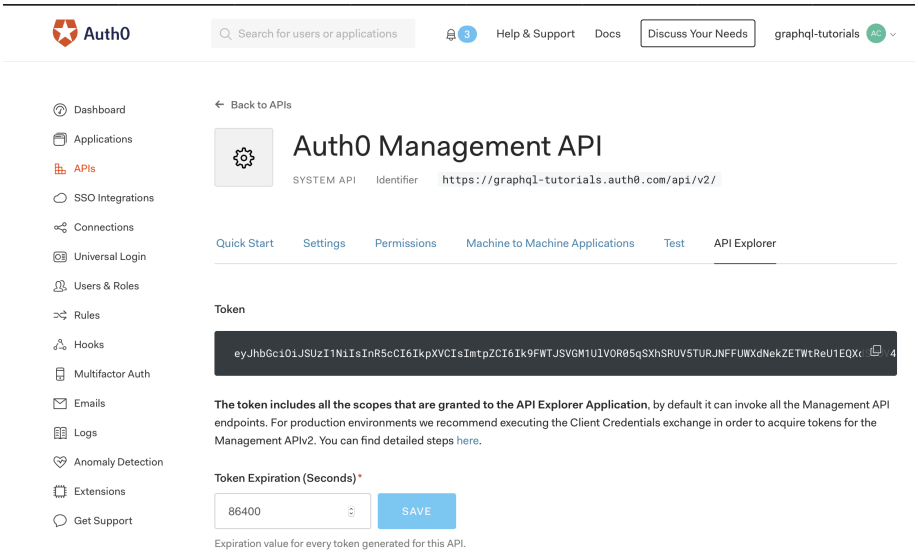
- We receive the payload `session_variables` as the request body from the Action.
- We make a request to the Auth0's Management API, passing in the `user_id` to get details about this user.
- Once we get a response from the Auth0 API in our server, we form the following object `{email: resp.email, picture: resp.picture}` and send it back to the client. Else, we return an error case.

In case you are stuck with the code above, use the following readymade server on Glitch to clone it. You also need to remix the Glitch project to start modifying any code.

Environment variables

In your Glitch app source code, modify the `.env` file to enter the - `AUTH0_MANAGEMENT_API_TOKEN` - `AUTH0_DOMAIN`

values appropriately. The AUTH0_MANAGEMENT_API_TOKEN can be obtained from the Auth0 project.

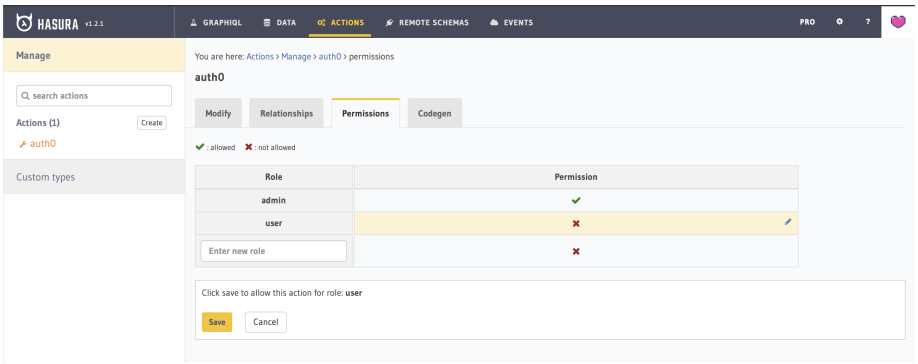


Auth0 Management API

Congrats! You have written and deployed your first Hasura Action to extend the Graph.

Permission

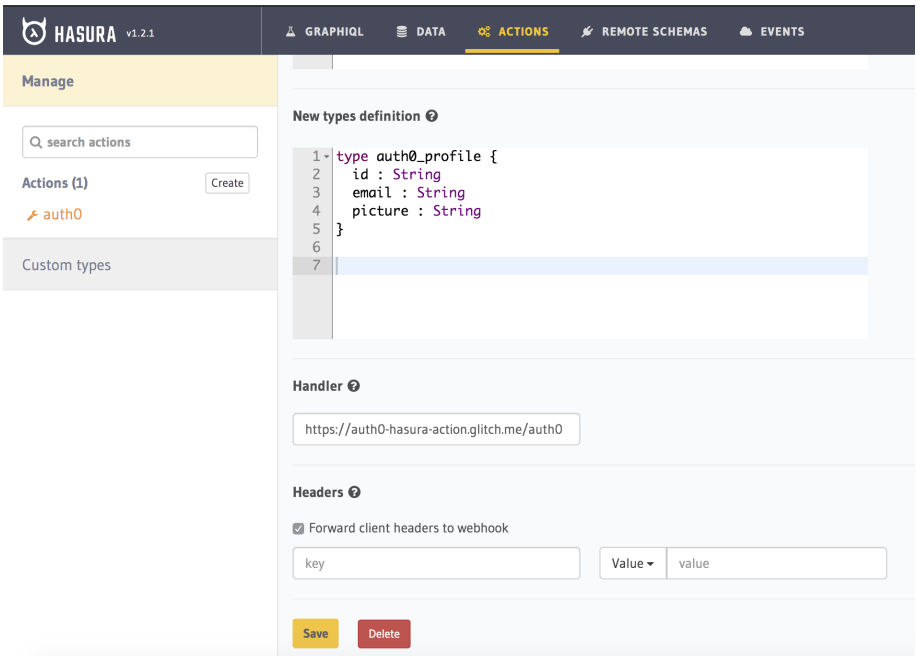
Now to query the newly added type, we need to give Permissions to the `user` role for this query type. Head to the `Permissions` tab of the newly created Action and configure access for the role user.



Action permission

Alright, now how do we query this newly added API?

First, we need to update the webhook url for the Action. Copy the deployed app URL from Glitch and add that as the webhook handler. Don't forget to add the route `/auth0` along with the Glitch app URL.



Action Handler

Now head to GraphiQL and try out the following query:

```
query {
  auth0 {
    email
    picture
  }
}
```

Remember the JWT token that we got after configuring Auth0 and testing it out? Here you also need to pass in the `Authorization` header with the same JWT token to get the right data.

Note: You need to pass in the right header values. You can pass in the Authorization header with the correct token and your Node.js server will receive the appropriate `x-hasura-user-id` value from the session variables for the API to work as expected.

That's it! You have now extended the built-in GraphQL API with your own custom code.

Write custom resolvers

Now we saw how the GraphQL API can be extended using Actions. We mentioned earlier that another way of customising the API graph is through a custom GraphQL server.

Let's take the same use-case of fetching profile information from Auth0.

Hasura has the ability to merge remote GraphQL schemas and provide a unified GraphQL API. To handle the use-case of fetching Auth0 profile information, we will write custom resolvers in a custom GraphQL server. Hasura can then merge this custom GraphQL server with the existing auto-generated schema.

This custom GraphQL server is the `Remote Schema`.

Write GraphQL custom resolver

So let's write a custom resolver which can be later merged into Hasura's GraphQL API.

```
const { ApolloServer } = require('apollo-server');
const gql = require('graphql-tag');
const jwt = require('jsonwebtoken');
const fetch = require('node-fetch');

const typeDefs = gql`
  type auth0_profile {
      email: String
      picture: String
    }

    type Query {
      auth0: auth0_profile
    }
`;

function getProfileInfo(user_id){
    const headers = {'Authorization': 'Bearer '+process.env.AUTH0_MANAGEMENT_API_TOKE
    console.log(headers);
    return fetch('https://' + process.env.AUTH0_DOMAIN + '/api/v2/users/'+user_id,{
        .then(response => response.json())
}


const resolvers = {
    Query: {
        auth0: (parent, args, context) => {
          // read the authorization header sent from the client
          const authHeaders = context.headers.authorization || '';
```

```
          const token = authHeaders.replace('Bearer ', '');
          // decode the token to find the user_id
          try {
            if (!token) {
              return 'Authorization token is missing!';
            }
            const decoded = jwt.decode(token);
            const user_id = decoded.sub;
            // make a rest api call to auth0
            return getProfileInfo(user_id).then( function(resp) {
              console.log(resp);
              if (!resp) {
                return null;
              }
              return {email: resp.email, picture: resp.picture};
            });
          } catch(e) {
            console.log(e);
            return null;
          }
        }
    },
};

const context = ({req}) => {
  return {headers: req.headers};
};

const schema = new ApolloServer({ typeDefs, resolvers, context});

schema.listen({ port: process.env.PORT}).then(({ url }) => {
    console.log(`schema ready at ${url}`);
});
```

In the server above, let's breakdown what's happening:

- We define the GraphQL types for `auth0_profile` and `Query`.
- And then we write a custom resolver for Query type `auth0`, where we parse the `Authorization` headers to get the token.
- We then decode the token using the `jsonwebtoken` library's `jwt` method. This gives the user_id required to fetch auth0 profile information.
- We make a request to the Auth0's Management API, passing in the token and the user_id to get details about this user.
- Once we get a response, we return back the object `{email: resp.email, picture: resp.picture}` as response. Else, we return `null`.

Note Most of the code written is very similar to the REST API code we wrote in the previous section for Actions. Here we are using Apollo Server to write a custom GraphQL server from scratch.

Deploy

Let's deploy the above custom GraphQL server to Glitch. Glitch is a platform to build and

deploy apps (Node.js) and is a quick way to test and iterate code on the cloud. Click on the Deploy to Glitch button below to get started.

Environment variables

After remixing to your own project on Glitch, modify the `.env` file to enter the - `AUTH0_MANAGEMENT_API_TOKEN` - `AUTH0_DOMAIN`

values appropriately.

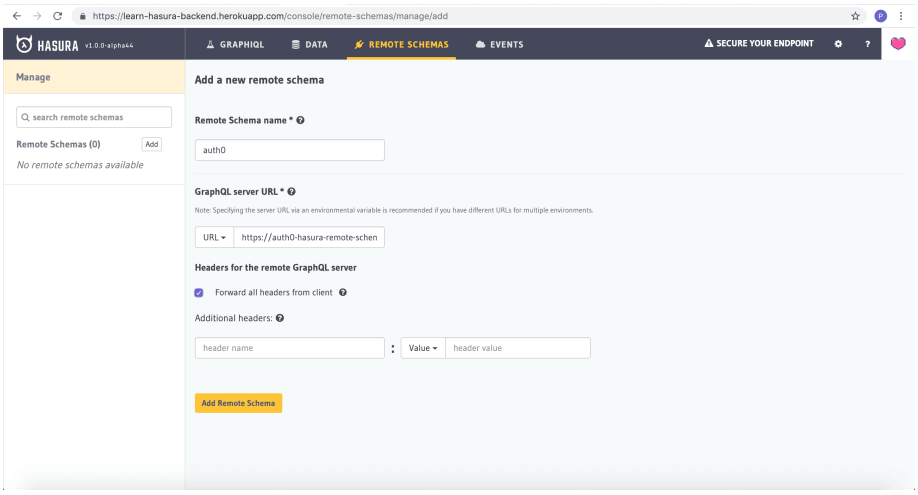
Congrats! You have written and deployed your first GraphQL custom resolver.

Add Remote Schema

We have written the custom resolver and deployed it to Glitch. We have the GraphQL endpoint ready. Let's add it to Hasura as a remote schema.

Add

Head to the `Remote Schemas` tab of the console and click on the `Add` button.



Add remote schema

Give a name for the remote schema (let's say auth0). Under GraphQL Server URL, enter the glitch app url that you just deployed in the previous step.

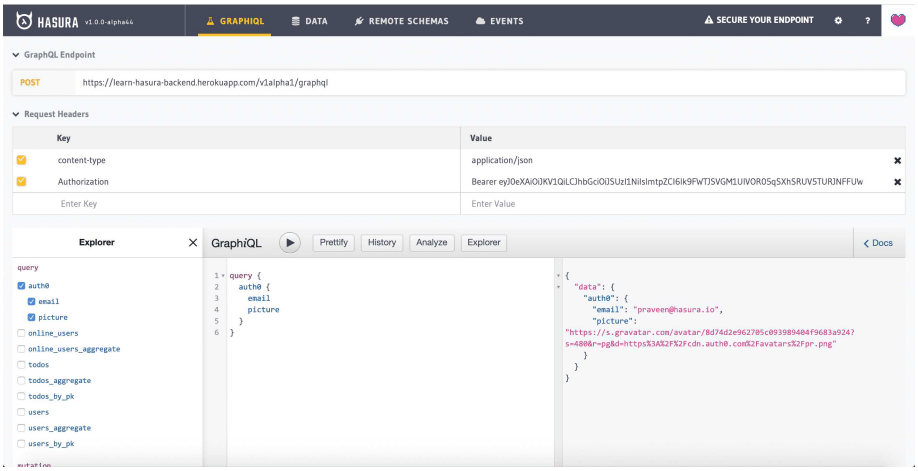
Select `Forward all headers from the client` and click on `Add Remote Schema` .

Try it out

Head to Console GraphiQL tab and explore the following GraphQL query.

```
query {
  auth0 {
    email
    picture
  }
}
```

Remember the JWT token that we got after configuring Auth0 and testing it out? Here you also need to pass in the `Authorization` header with the same JWT token to get the right data.



remote schema query

As you can see, Hasura has merged the custom GraphQL schema with the already existing auto-generated APIs over Postgres.

Write event webhook

Now let's move to the second use-case of sending an email when a user registers on the app.

When the user registers on the app using Auth0, we insert a new row into the `users` table to keep the user data in sync. Remember the Auth0 rule we wrote during signup to make a mutation?

This is an `insert` operation on table `users`. The payload for each event is mentioned here

Now we are going to capture this insert operation to trigger our event.

SendGrid SMTP Email API

For this example, we are going to make use of `SendGrid` 's SMTP server and use `nodemailer` to send the email.

Signup on SendGrid and create a free account.

Create an API Key by following the docs here

Write the webhook

```
const nodemailer = require('nodemailer');
const transporter = nodemailer.createTransport('smtp://'+process.env.SMTP_LOGIN+':'+
const fs = require('fs');
const path = require('path');
const express = require('express');
const bodyParser = require('body-parser');
const app = express();

app.set('port', (process.env.PORT || 3000));

app.use('/', express.static(path.join(__dirname, 'public')));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({extended: true}));

app.use(function(req, res, next) {
    res.setHeader('Access-Control-Allow-Origin', '*');
    res.setHeader('Cache-Control', 'no-cache');
    next();
});

app.post('/send-email', function(req, res) {

  const name = req.body.event.data.new.name;
  // setup e-mail data
  const mailOptions = {
      from: process.env.SENDER_ADDRESS, // sender address
      to: process.env.RECEIVER_ADDRESS, // list of receivers
      subject: 'A new user has registered', // Subject line
      text: 'Hi, This is to notify that a new user has registered under the name of '
      html: '<p>'+'Hi, This is to notify that a new user has registered under the nar
  };
  // send mail with defined transport object
  transporter.sendMail(mailOptions, function(error, info){
      if(error){
          return console.log(error);
      }
      console.log('Message sent: ' + info.response);
      res.json({'success': true});
  });

});

app.listen(app.get('port'), function() {
  console.log('Server started on: ' + app.get('port'));
});
```

Deploy

Environment variables

After remixing to your own project on Glitch, modify the `.env` file to enter the - `SMTP_LOGIN` , - `SMTP_PASSWORD` , - `SMTP_HOST`

values appropriately.

Additionally you should also configure the sender and receiver address using - `SENDER_ADDRESS` - `RECEIVER_ADDRESS`

env variables.

Congrats! You have written and deployed your first webhook to handle database events.

Create Event Trigger

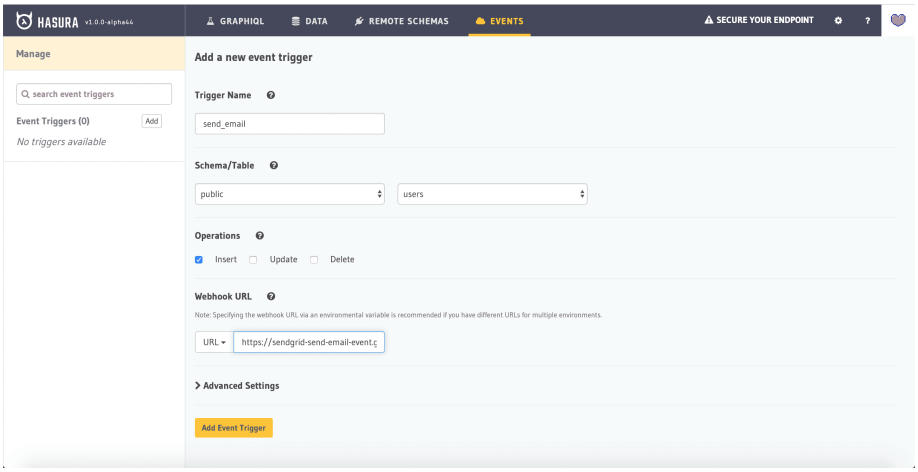
Event triggers can be created using the Hasura console.

Open the Hasura console, head to the Events tab and click on the Create trigger button to open up the interface below to create an event trigger:

Add Event Trigger

Give a name for the event trigger (say send_email) and select the table `users` and select the operation `insert`.

Click on `Create`.



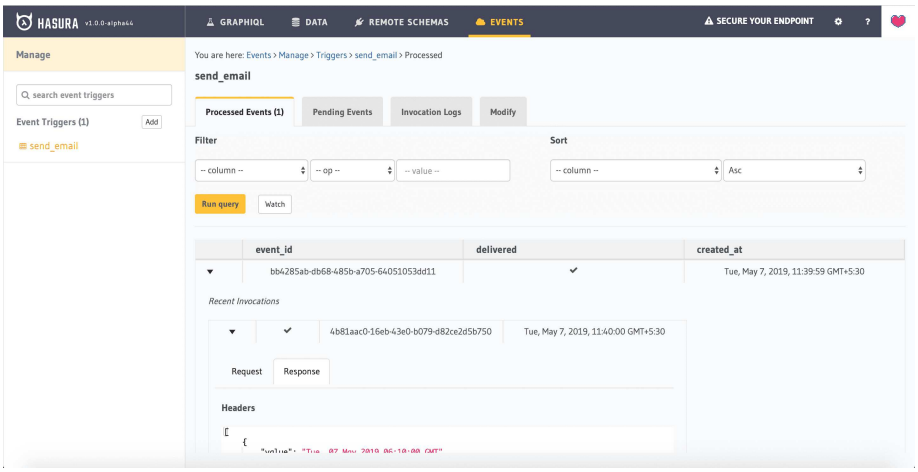
Create event trigger

Try it out

To test this, we need to insert a new row into users table.

Head to Console -> Data -> users -> Insert Row and insert a new row.

Now head to Events tab and click on `send_email` event to browse the processed events.



Test event trigger

Now everytime a new row is inserted into `users` table this event would be invoked.

What next?

- Join Discord for Support
- Take the frontend course
- Spread the word!

Congratulations! You have successfully completed the course.

In case you are wondering what to do next, you have the following options:

Join Discord for Support

We have a Discord Channel for those who are new to GraphQL. Please post your queries or feedback regarding this course. It is a pretty active channel.

Here's the invite link to join discord.

Take the frontend course

We have courses for your favorite frontend frameworks like React, Vue etc using the same backend that you just created.

- React - https://hasura.io/learn/graphql/react
- Vue - https://hasura.io/learn/graphql/vue

Spread the word!

Liked the course? Do spread the word on Twitter!