

Course Introduction

- [What will I learn?](#)
- [What will we be building?](#)
- [What do I need to take this tutorial?](#)
- [How long will this tutorial take?](#)

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

- [One-click deployment on Heroku](#)
- [Hasura Console](#)

Let's start by deploying Hasura.

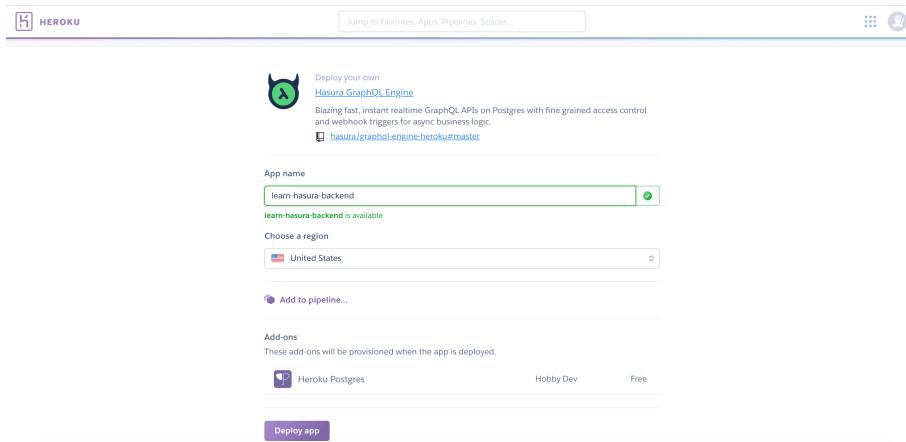
One-click deployment on Heroku

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.

The screenshot shows the Heroku dashboard after a successful deployment. At the top, there's a banner for provisioning add-ons like Heroku Postgres. Below it, a 'Deploy app' section lists several steps: 'Create app' (checked), 'Configure environment' (checked), 'Build app' (checked, with a link to 'Show build log'), 'Run scripts & scale dynos' (checked), and 'Deploy to Heroku' (checked). A message at the bottom says 'Your app was successfully deployed.' with links to 'Manage App' and 'View'.

heroku.com Blogs Careers Documentation Support Terms of Service Privacy Cookies © 2019 Salesforce.com

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:

The screenshot shows the Hasura GraphQL Admin Console interface. At the top, there's a navigation bar with tabs for GRAPHQL, DATA, REMOTE SCHEMAS, and EVENTS. The GRAPHQL tab is selected. Below the navigation is a 'GraphQL Endpoint' dropdown set to 'https://learn-hasura-backend.herokuapp.com/v1alpha1/graphql'. Underneath is a 'Request Headers' section with a table:

Key	Value
content-type	application/json
Enter Key	Enter Value

Below the headers is a 'GraphQL' input field with the word 'Explorer' and a dropdown menu containing 'GraphiQL', 'Prettyify', 'History', 'Analyze', and 'Explorer'. The 'GraphiQL' option is selected. In the main query editor area, there is a 'query' placeholder with some sample GraphQL code:

```
query {  
  # Looks like you do not have any tables.  
  # Click on the "Data" tab on top to create tables  
  # Try out GraphQL queries here after you create tables  
  1  
  2  
  3  
  4}
```

Hasura Console

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

Basic Data Modelling

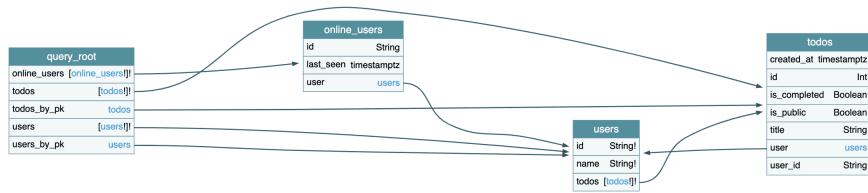
In this part of the course, we will build the data model for a realtime todo app. Our todo app will have the following features:

- 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.

The screenshot shows the Hasura Console interface for creating a new table named `users`. The table has five columns defined in the `SQL` schema:

- `id`: Type `Text`, `default_value` is empty, `Nullable` and `Unique` checkboxes are unchecked.
- `name`: Type `Text`, `default_value` is empty, `Nullable` and `Unique` checkboxes are unchecked.
- `created_at`: Type `Timestamp`, `default_value` is `now()`, `Nullable` and `Unique` checkboxes are unchecked.
- `last_seen`: Type `Timestamp`, `default_value` is `example: now()`, `Nullable` checkbox is checked, and `Unique` checkbox is unchecked.
- `column_name`: Type `- type -`, `default_value` is empty, `Nullable` and `Unique` checkboxes are unchecked.

The `Primary Key` dropdown is set to `id`. There are no foreign keys defined for this table.

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:

The screenshot shows the Hasura GraphQL Explorer interface. The top navigation bar includes tabs for GRAPHIQL, DATA, REMOTE SCHEMAS, and EVENTS, along with a SECURE YOUR ENDPOINT button. The main area has a 'GraphiQL' tab selected. On the left, there's a sidebar with 'GraphQL Endpoint' and a 'Request Headers' section where 'content-type: application/json' is set. The main panel shows a query editor with the following mutation:

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

The results pane on the right shows the response:

```
{  
  "data": {  
    "insert_users": {  
      "affected_rows": 1  
    }  
  }  
}
```

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 GraphQL 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:

The screenshot shows the Hasura GraphQL Explorer interface. At the top, it displays the URL <https://learn-hasura-backend.herokuapp.com/console/api-explorer>. Below the URL, there are tabs for GRAPHQL, DATA, REMOTE SCHEMAS, and EVENTS. A 'SECURE YOUR ENDPOINT' button is also visible. The main area is titled 'GraphQL Endpoint' with the URL <https://learn-hasura-backend.herokuapp.com/v1alpha1/graphql>. Under 'Request Headers', there is a table with one row: 'content-type' set to 'application/json'. On the left, there is a sidebar titled 'Explorer' with various filter options for 'query' and 'mutation' (e.g., 'users', 'distinct_on', 'limit', 'offset', 'order_by', 'where', 'created_at', 'id', 'last_seen', 'name', 'users_aggregate', 'users_by_pk'). The central panel shows the query code and its resulting JSON response. The query is:

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

```
{  
  "data": {  
    "users": [  
      {  
        "id": "1",  
        "name": "Praveen",  
        "created_at": "2019-05-06T09:47:03.660594+00:00"  
      }  
    ]  
  }  
}
```

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 **[GRAPHQL]** tab and see the subscription response returning 2 results.

```

query {
  users {
    id
    name
  }
}

```

```

{
  "data": [
    {
      "id": "1",
      "name": "Praveen"
    },
    {
      "id": "2",
      "name": "Tanmai"
    }
  ]
}

```

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.

The screenshot shows the Hasura Console interface. The top navigation bar includes tabs for GRAPHQL, DATA, REMOTE SCHEMAS, and EVENTS. A 'SECURE YOUR ENDPOINT' button and a help icon are also present. The main area is titled 'Add a new table'. In the 'Tables' section, there is one entry for 'users'. Below it, under 'SQL', is the 'Todos' table being defined. The 'Columns' section lists the following fields:

Column Name	Type	Default Value	Nullable	Unique
id	Integer (auto-increment)	default_value	Nullable	Unique
title	Text	default_value	Nullable	Unique
is_completed	Boolean	false	Nullable	Unique
is_public	Boolean	false	Nullable	Unique
created_at	Timestamp	now()	Nullable	Unique
user_id	Text	default_value	Nullable	Unique
column_name	Text	default_value	Nullable	Unique

The 'Primary Key' dropdown is set to 'id'. The 'Add Table' button is located at the bottom left of the table configuration area.

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:

The screenshot shows the Hasura GraphQL Explorer interface. The top navigation bar includes links for GRAPHIQL, DATA, REMOTE SCHEMAS, and EVENTS, along with a SECURE YOUR ENDPOINT button. The main area has a 'GraphIQL' tab selected. On the left, there's a sidebar with 'GraphIQL Endpoint' and 'Request Headers' (Content-Type: application/json). The main panel shows a query editor with the following code:

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

The results pane on the right displays the response:

```
{ "data": { "insert.todos": { "affected_rows": 1 } } }
```

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:

The screenshot shows the Hasura GraphQL Explorer interface. The top navigation bar includes tabs for GRAPHQL, DATA, REMOTE SCHEMAS, and EVENTS. A 'SECURE YOUR ENDPOINT' button is also present. The main area has a 'GraphiQL' tab selected. Below it, a 'Request Headers' section shows a 'content-type' header set to 'application/json'. The 'Explorer' sidebar on the left lists various schema elements like 'todos', 'users', and 'mutations'. The central code editor contains a GraphQL query:

```

query {
  todos {
    id
    title
    is_public
    is_completed
    user_id
  }
}

```

To the right of the code editor is the response pane, which displays the JSON result:

```

{
  "data": {
    "todos": [
      {
        "id": 1,
        "title": "My First Todo",
        "is_public": false,
        "is_completed": false,
        "user_id": "1"
      }
    ]
  }
}

```

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.

The screenshot shows the Hasura GraphQL Engine interface. The top navigation bar includes 'GRAPHQL', 'DATA', 'REMOTE SCHEMAS', and 'EVENTS'. The 'DATA' tab is selected. Below it, the schema is set to 'public'. The 'todos' table is selected. A new row is being inserted with the following fields:

- id**: nextval('todos_id_seq'::regclass)
- title**: Second Todo
- is_completed**: False
- is_public**: False
- created_at**: now()
- user_id**: 1

Buttons at the bottom include 'Save' and 'Clear'.

Insert new todo

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

The screenshot shows the Hasura GraphQL Engine API Explorer. The 'GraphQL Endpoint' is set to <https://learn-hasura-backend.herokuapp.com/v1alpha1/graphql>. A subscription query for 'todos' is run, returning the following data:

```

{
  "data": [
    {
      "id": 1,
      "title": "My First Todo",
      "is_public": false,
      "is_completed": false,
      "user_id": "1"
    },
    {
      "id": 2,
      "title": "Second Todo",
      "is_public": false,
      "is_completed": false,
      "user_id": "1"
    }
  ]
}

```

The left sidebar shows mutation and subscription options, including 'delete_todos', 'insert_todos', and 'todo'.

Todo Subscription

Relationships

- [Object Relationships](#)
- [Array 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:

The screenshot shows the Hasura GraphQL Engine interface. The top navigation bar has tabs for GRAPHQL, DATA, REMOTE SCHEMAS, and EVENTS. The SECURE YOUR ENDPOINT button is also visible. Below the navigation, the schema is set to 'public'. The 'Todos' table is selected for modification. The 'Modify' tab is active. The 'user_id' column is currently selected for modification, indicated by a yellow border around its row. The column details show it is a text type with a primary key constraint named `todos_pk`. The 'Add a new column' section is visible at the bottom.

Todos Modify Page

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

The screenshot shows the 'Foreign Keys' configuration dialog. The 'Reference Table' dropdown is set to 'users'. The 'From' section shows the 'user_id' column selected, and the 'To' section shows the 'id' column selected. Under 'On Update Violation', the 'restrict' option is selected. Under 'On Delete Violation', the 'restrict' option is selected. A 'Save' button is at the bottom right of the dialog.

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.

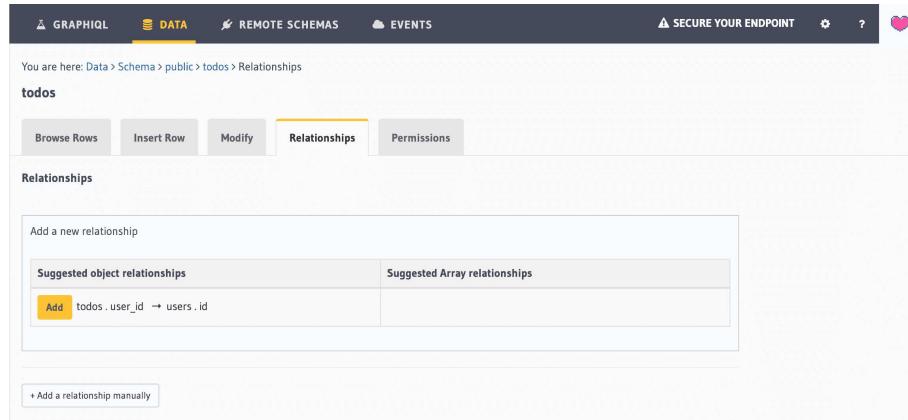
Click on `Save` to create the foreign key.

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:



You are here: Data > Schema > public > todos > Relationships

todos

Browse Rows Insert Row Modify Relationships Permissions

Relationships

Add a new relationship

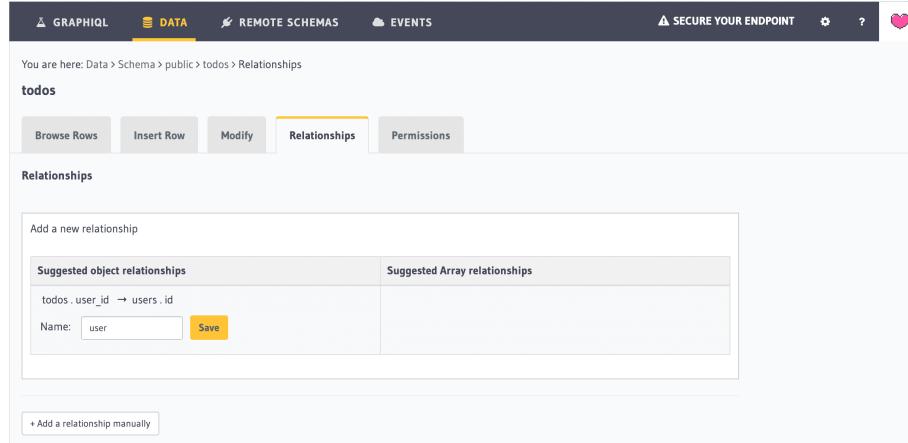
Suggested object relationships	Suggested Array relationships
Add todos . user_id → users . id	

+ Add a relationship manually

Todos Relationships Page

Click on `Add` in the suggested object relationship.

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



You are here: Data > Schema > public > todos > Relationships

todos

Browse Rows Insert Row Modify Relationships Permissions

Relationships

Add a new relationship

Suggested object relationships	Suggested Array relationships
todos . user_id → users . id	

Name: [Save](#)

+ Add a relationship manually

User Object Relationship

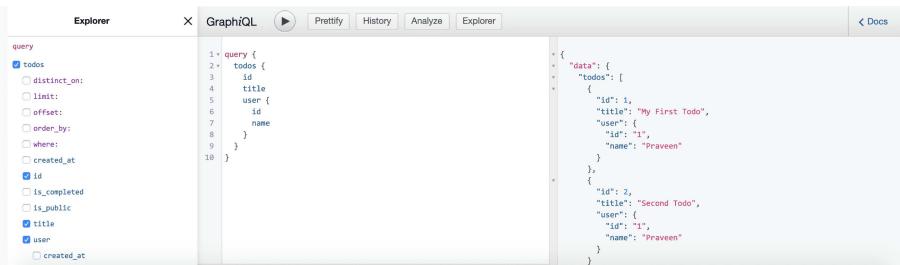
A relationship has now been established between todos and users table.

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:



The screenshot shows a GraphQL IDE interface with the following components:

- Explorer** tab: Shows a sidebar with a tree view of fields and filters. The "todos" field is expanded, showing "id", "title", and "user" sub-fields. The "user" sub-field is also expanded, showing "id" and "name".
- GraphiQL** tab: Contains the GraphQL query text shown above.
- Prettify**, **History**, **Analyze**, and **Explorer** buttons: Located at the top of the main area.
- Docs** button: Located in the top right corner.
- Response Area**: Displays the JSON response to the query. The response is as follows:

```
query {
  todos {
    id
    title
    user {
      id
      name
    }
  }
}
{
  "data": {
    "todos": [
      {
        "id": 1,
        "title": "My First Todo",
        "user": {
          "id": "1",
          "name": "Praveen"
        }
      },
      {
        "id": 2,
        "title": "Second Todo",
        "user": {
          "id": "1",
          "name": "Praveen"
        }
      }
    ]
  }
}
```

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

- [Create View](#)
- [Subscription to Online Users](#)

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.

The screenshot shows the Hasura GraphQL Engine interface. At the top, there are tabs: GRAPHQL, DATA (which is selected), REMOTE SCHEMAS, and EVENTS. Below the tabs, there's a section titled "Raw SQL". Under "Raw SQL", there's a "Notes" section with a bulleted list of instructions for creating views. The "SQL:" section contains the provided SQL code. A yellow highlight covers the WHERE clause. Below the SQL input, there are two checkboxes: "Track this" (checked) and "Cascade metadata". A large yellow "Run!" button is at the bottom.

```
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));
```

Create view online_users

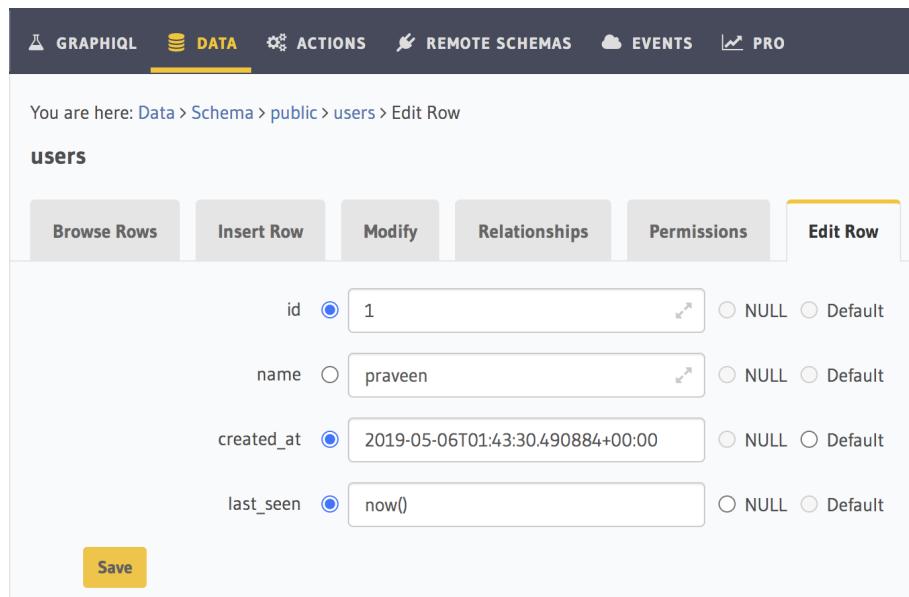
Click on `Run` to create the view.

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.



The screenshot shows the Neo4j Data tab interface. The URL in the address bar is `Data > Schema > public > users > Edit Row`. The page title is "users". Below it are five buttons: "Browse Rows", "Insert Row", "Modify", "Relationships", and "Permissions". The "Edit Row" button is highlighted with a yellow border. The main area contains four input fields for the "users" node:

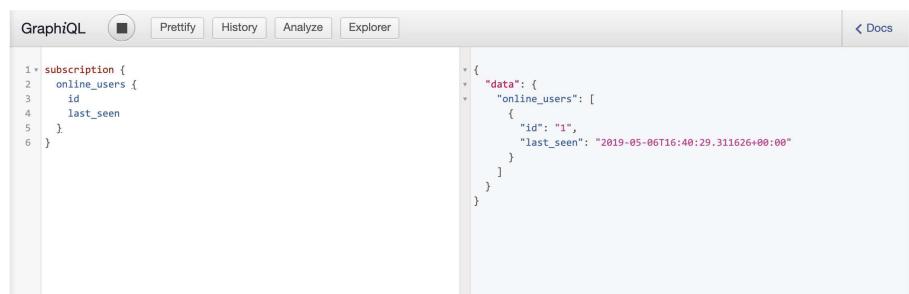
- "id": A radio button is selected next to the value "1". There are three options: "1" (selected), "NULL", and "Default".
- "name": A radio button is selected next to the value "praveen". There are three options: "praveen" (selected), "NULL", and "Default".
- "created_at": A radio button is selected next to the value "2019-05-06T01:43:30.490884+00:00". There are three options: "2019-05-06T01:43:30.490884+00:00" (selected), "NULL", and "Default".
- "last_seen": A radio button is selected next to the value "now()". There are three options: "now()" (selected), "NULL", and "Default".

A yellow "Save" button is located at the bottom left of the form.

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.



The screenshot shows the GraphiQL interface with the "GraphiQL" tab selected. The "subscription" query is pasted into the query editor:

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

The results pane displays the JSON response:

```
{
  "data": {
    "online_users": [
      {
        "id": "1",
        "last_seen": "2019-05-06T16:40:29.311626+00:00"
      }
    ]
  }
}
```

Subscription online users

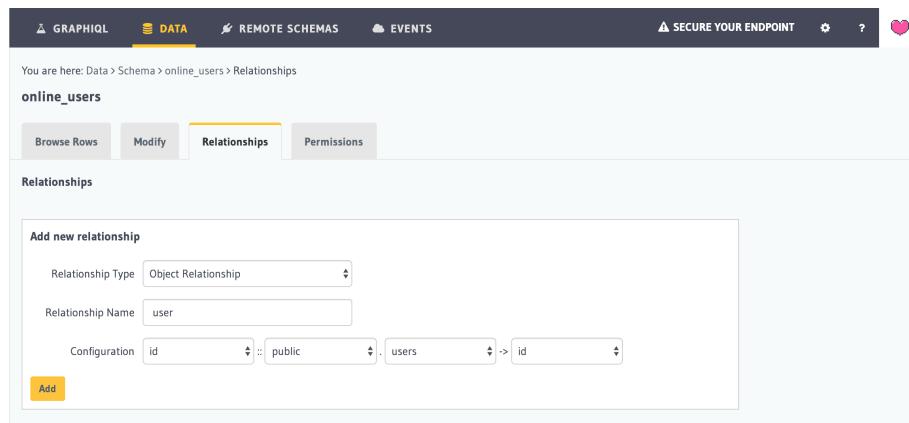
Create relationship to user

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.



The screenshot shows the Data Relationships page for the `online_users` view. The `Relationships` tab is active. A modal window titled "Add new relationship" is open. Inside the modal, the "Relationship Type" dropdown is set to "Object Relationship". The "Relationship Name" input field contains "user". The "Configuration" section shows a mapping: `id` from `online_users` to `public` and `users`, and `id` from `users` back to `online_users`. At the bottom of the modal is a yellow "Add" button.

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

- [Insert permission](#)
- [Select permission](#)
- [Update permission](#)
- [Delete permission](#)

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"}}
```

The screenshot shows the Hasura GraphQL Engine interface. The top navigation bar includes GRAPHQL, DATA, REMOTE SCHEMAS, EVENTS, and SECURE YOUR ENDPOINT. The main area shows a schema named 'public'. Under 'Tables (3)', the 'todos' table is selected. The 'Permissions' tab is active. A table shows permissions for 'admin' and 'user' roles across actions insert, select, update, and delete. For the 'user' role, the 'insert' row has a custom check configuration. The condition is set to `{"user_id": {"_eq": "X-Hasura-User-Id"}}`. Below this, a detailed view of the custom check shows the JSON structure: `{ "user_id": { "eq": "X-Hasura-User-Id" } }`.

Todos row permission insert

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

Role: user Action: insert

Close

Row insert permissions - with custom check

Allow role **user** to insert **rows**:

Without any checks
 With custom check:

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

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

Column insert permissions - partial columns

Allow role **user** to set input for **columns**: Toggle All

id title is_completed is_public created_at user_id

For **relationships**, set permissions for the corresponding tables/views.

Column presets - no presets

Save Permissions **Delete Permissions**

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"}}]}
```

Role: user Action: select

Row select permissions - with custom check

Allow role **user** to select **rows**:

- Without any checks
- With same custom checks as **insert**
- With custom check: [?](#)

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

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

Todos select permission row

Under column select permissions, select all the columns.

Limit number of rows: [?](#)

Column select permissions - **all columns**

Allow role **user** to access **columns**: [Toggle All](#)

created_at **id** **is_completed** **is_public** **title** **user_id**

For **relationships**, set permissions for the corresponding tables/views.

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.

Role: user Action: update

Row update permissions - with custom check

Allow role **user** to update **rows**:

- Without any checks
- With same custom check as **insert**

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

- With same custom check as **select**
- With custom check: [?](#)

Column update permissions - partial columns

Allow role **user** to update **columns**: [Toggle All](#)

is_completed is_public id user_id title created_at

Column presets - no presets

[Save Permissions](#) [Delete Permissions](#)

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](#).

		Actions	Role	insert	select	update	delete
			admin	✓	✓	✓	✓
<input type="checkbox"/>			user	✗	✗	✗	✗
			Enter new role	✗	✗	✗	✗

Role: user Action: delete

Row delete permissions - with custom check

Allow role **user** to delete **rows**:

- Without any checks
- With same custom checks as **insert, update**

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

- With same custom checks as **select**
- With custom check: [?](#)

[Save Permissions](#) [Delete Permissions](#)

Todos delete permission

Click on [Save Permissions](#) and you are done with access control for **todos** table.

Setup users table permissions

- [Select permission](#)
- [Update permission](#)

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.

Actions	Role	insert	select	update	delete
	admin	✓	✓	✓	✓
<input type="checkbox"/>	user	✗	✗	✗	✗
	Enter new role	✗	✗	✗	✗

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.

Role: user Action: update

Row update permissions - with custom check

Allow role **user** to update **rows**:

Without any checks (Same as **select**)

With custom check: [?](#)

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

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

Column update permissions - partial columns

Allow role **user** to update **columns**: [Toggle All](#)

created_at id last_seen name

Column presets - no presets

[Save Permissions](#) [Delete Permissions](#)

users update permission

Click on [Save Permissions](#) and you are done with access control rules for **users** table.

Setup online_users view permissions

- [Select permission](#)

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`.

The screenshot shows the Hasura GraphQL Engine interface with the 'DATA' tab selected. In the sidebar, the 'Schema: public' is chosen, and the 'Tables (3)' section lists 'online_users', 'todos', and 'users'. The 'online_users' table is currently selected. The main area displays a grid of permissions for the 'user' role across various actions (CREATE, READ, UPDATE, DELETE) and columns ('id', 'last_seen'). For the 'READ' action on the 'id' column, the 'Without any checks' option is selected. For the 'READ' action on the 'last_seen' column, both 'id' and 'last_seen' are selected. Other rows in the grid show standard CRUD permissions for the 'admin' and 'user' roles. A modal window titled 'Role: user Action: select' provides detailed configuration for these permissions, including the choice of 'Without any checks' for row select and specific column selection for column select.

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.

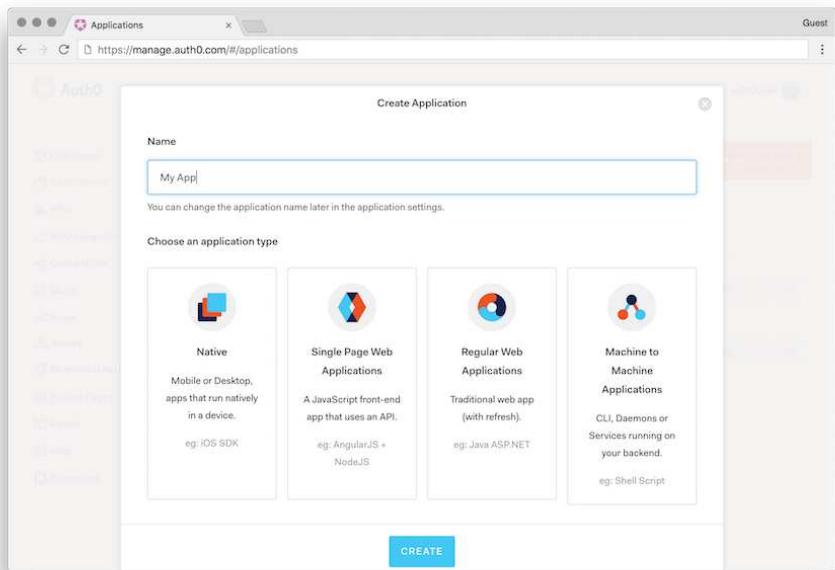
Note: Auth0 has a free plan for up to 7000 active users.

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);
}
```

The screenshot shows the Auth0 dashboard with the 'Edit Rule' page open. The left sidebar contains navigation links like Dashboard, Applications, APIs, SSO Integrations, Connections, Universal Login, Users, Rules (which is selected), Hooks, Multi-factor Auth, Emails, Logs, Anomaly Detection, Extensions, and Get support. The main area has a title 'Edit Rule' and a sub-section 'hasura-jwt-claim'. A note at the top says: 'Heads up! If you are trying to access a service behind a firewall, make sure to open the right ports and allow inbound connections from these IP addresses: 35.167.74.121, 35.166.282.113, 35.160.3.103, 54.183.64.135, 54.67.77.38, 54.67.15.178, 54.183.204.205'. Below this is the code editor containing the provided custom JWT claim function.

```
1  function (user, context, callback) {
2    const namespace = "https://hasura.io/jwt/claims";
3    context.idToken[namespace] =
4    {
5      'x-hasura-default-role': 'user',
6      // do some custom logic to decide allowed roles
7      'x-hasura-allowed-roles': ['user'],
8      'x-hasura-user-id': user.user_id
9    };
10   callback(null, user, context);
11 }
```

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>

The screenshot shows a web browser window with the URL <https://hasura.io/jwt-config>. The page has a header "Select Provider" with "Auth0" selected. Below it is a field "Enter auth0 domain name" containing "myapp.auth0.com". A blue button "GENERATE CONFIG" is visible. To the right of the input fields is a link "(How do i use this?)". The main content area contains a large block of JSON-like text representing the generated JWT configuration. At the bottom of this area is a note: "Note: If you are using docker-compose.yaml, please wrap the above in a single quote". A blue message icon is located at the bottom right of the page.

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:

Config Vars

Config vars change the way your app behaves. In addition to creating your own, some add-ons come with their own.

KEY	VALUE
DATABASE_URL	postgres://dpjpfpszqodxbm:e5be6610e0f
HASURA_GRAPHQL_JWT_SECRET	{"type": "RS512", "key": "-----BEGIN -----END -----"}
HASURA_GRAPHQL_ADMIN_SECRET	xxxxxxxx
KEY	VALUE

Hide Config Vars

Info Region United States

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);
    });
  }
}
```

```
insert-user
1  function (user, context, callback) {
2    const userId = user.user_id;
3    const nickname = user.nickname;
4
5    const admin_secret = "xxxx";
6    const url = "https://learn-hasura-backend.herokuapp.com/v1/graphql";
7
8    request.post({
9      headers: {'content-type': 'application/json', 'x-hasura-access-key': admin_secret},
10     url: url,
11     body: `{"query": "mutation($userId: String!, $nickname: String) {\\n
12   }, function(error, response, body){
13     console.log(body);
14     callback(null, user, context);
15   });
16 }
```

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_id=client_id&protocol=oauth2&response_type=token%20id_token&redirect_uri=callback_uri&scope=openid%20profile%20email`

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#xxxxxxxxx&id_token=yyyyyy`. 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.



The screenshot shows a jwt.io debugger interface. A JWT token is pasted into the input field. The `id_token` parameter is highlighted with a red box. The token itself is a long string of characters.

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

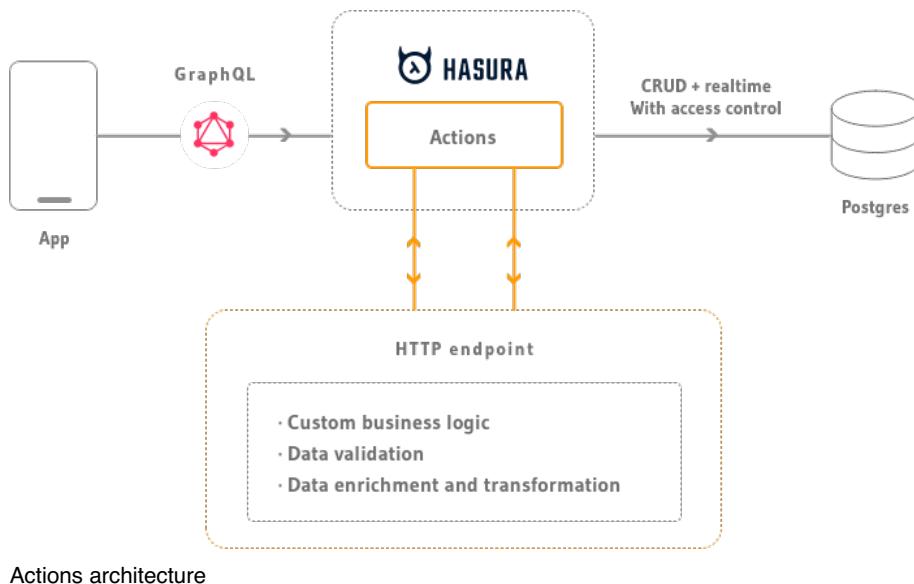
- [Actions \(Recommended\)](#)
- [Remote Schemas](#)
- [Event Triggers](#)

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

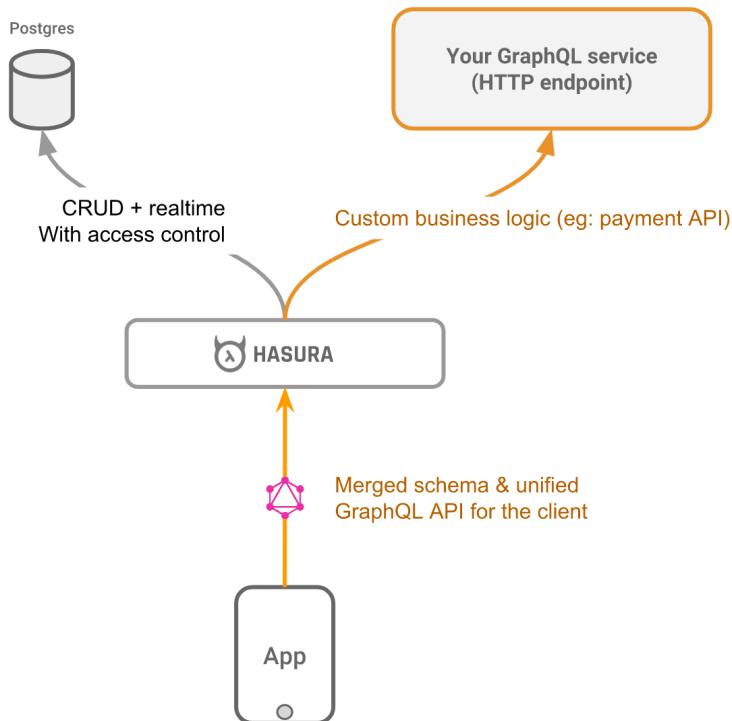
Actions can be either a Query or a Mutation.

- [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

- [Creating an action](#)
 - Action definition
 - Types definition
- [Write a REST API](#)
 - Environment variables
- [Permission](#)

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.

The screenshot shows the Hasura Console interface with the 'ACTIONS' tab selected. A modal window titled 'Add a new action' is open. The 'Action definition' section contains the following GraphQL code:

```
1 - type Query {
2   auth0: auth0_profile
3 }
```

The 'New types definition' section contains the following GraphQL code:

```
1 - type auth0_profile {
2   id: String
3   email: String
4   picture: String
5 }
```

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.

Headers ?

Forward client headers to webhook

key	Value ▾	value
-----	---------	-------

Create

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.

The screenshot shows the Hasura GraphQL Engine interface. The top navigation bar has tabs for GRAPHQL, DATA, ACTIONS (which is selected), REMOTE SCHEMAS, and EVENTS. Below the navigation, there's a breadcrumb trail: You are here: Actions > Manage > auth0 > codegen. The main area is titled "auth0" and contains tabs for Modify, Relationships, Permissions, and Codegen (which is selected). A search bar says "search actions" and a "Create" button are visible. Below the tabs, there's a dropdown menu set to "nodejs-express". A link "Need help getting started quickly?" leads to "Try on glitch", "Starter-kit.zip", and "View on GitHub". The code editor shows the "auth0.js" file:

```
auth0.js
1 // Request Handler
2 app.post('/auth0', async (req, res) => {
3   // get request input
4   const { } = req.body.input;
5   // run some business logic
6
7   /*
8    * In case of errors:
9     return res.status(400).json({
10       message: "error happened"
11     })
12   */
13
14   // success
15   return res.json({
16     id: "random"
17   })
18 })
```

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 ${process.env.AUTH0_MANAGEMENT_API_TOKEN}`};
  console.log(headers);
  return fetch(`https://${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.

The screenshot shows the Auth0 Management API Token page. On the left, there's a sidebar with various management options like Dashboard, Applications, APIs, SSO Integrations, Connections, Universal Login, Users & Roles, Rules, Hooks, Multifactor Auth, Emails, Logs, Anomaly Detection, Extensions, and Get Support. The main area is titled "Auth0 Management API" and shows the "SYSTEM API Identifier" as `https://graphql-tutorials.auth0.com/api/v2/`. Below this, there are tabs for Quick Start, Settings, Permissions, Machine to Machine Applications, Test, and API Explorer. The "API Explorer" tab is selected. It displays a token value: `eyJhbGciOiJSUzI1NiIsInR5cCI6IkpxVCIsImtpZCI6Ik9FWTJSVGM1U1V0R05qSXhSRUV5TURJNFFUWxdNekZETWtReU1E0Xc...`. A note below says: "The token includes all the scopes that are granted to the API Explorer Application, by default it can invoke all the Management API endpoints. For production environments we recommend executing the Client Credentials exchange in order to acquire tokens for the Management APIv2. You can find detailed steps [here](#)." There's also a "Token Expiration (Seconds)" input field set to 86400, with a "SAVE" button and a note: "Expiration value for every token generated for this API".

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`.

The screenshot shows the Hasura Actions permission configuration screen. At the top, there are tabs for GRAPHQL, DATA, ACTIONS (which is selected), REMOTE SCHEMAS, and EVENTS. Below this, there's a breadcrumb navigation: "You are here: Actions > Manage > auth0 > permissions". The "auth0" action is selected. Under "Actions (1)", there's a list item "auth0". The "Custom types" section is empty. On the right, there are four tabs: Modify, Relationships, Permissions (selected), and Codegen. Under "Permissions", there are two roles: "admin" and "user". The "admin" role has a green checkmark under "Permission". The "user" role has a red X under "Permission". There's a note: "Click save to allow this action for role: user" with "Save" and "Cancel" buttons. A legend at the bottom left indicates: "✓ : allowed" and "✗ : not allowed".

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.

The screenshot shows the Hasura GraphQL Engine interface. The top navigation bar includes 'GRAPHQL', 'DATA', 'ACTIONS' (which is highlighted in yellow), 'REMOTE SCHEMAS', and 'EVENTS'. On the left, a sidebar titled 'Manage' shows 'Actions (1)' for 'auth0'. A search bar says 'search actions' and a 'Create' button is visible. Below this is a 'Custom types' section. The main area is titled 'New types definition' and contains the following GraphQL code:

```
1 - type auth0_profile {  
2   id : String  
3   email : String  
4   picture : String  
5 }  
6  
7 |
```

Below the code, there's a 'Handler' field containing 'https://auth0-hasura-action.glitch.me/auth0', a 'Headers' section with a checked checkbox for 'Forward client headers to webhook', and a 'key' input field. At the bottom are 'Save' and 'Delete' buttons.

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

- [Write GraphQL custom resolver](#)
- [Deploy](#)
 - [Environment variables](#)

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 ${process.env.AUTH0_MANAGEMENT_API_TOKEN}`;
  console.log(headers);
  return fetch(`https://${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 ', '');
      const decodedToken = jwt.verify(token, process.env.AUTH0_SECRET);
      const user_id = decodedToken.sub;
      return getProfileInfo(user_id);
    }
  }
};

module.exports = { typeDefs, resolvers }
```

```

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

- [Add](#)
- [Try it out](#)

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.

The screenshot shows the 'Add a new remote schema' interface. On the left, there's a sidebar with 'Manage' and a search bar. The main area has fields for 'Remote Schema name' (set to 'auth0') and 'GraphQL server URL' (set to 'https://auth0-hasura-remote-schen'). Under 'Headers for the remote GraphQL server', there's a checked checkbox for 'Forward all headers from client'. At the bottom is a yellow 'Add Remote Schema' 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 GraphQL 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.

The screenshot shows the Hasura GraphQL Engine interface. At the top, there's a navigation bar with tabs for GRAPHQL, DATA, REMOTE SCHEMAS, and EVENTS. On the right, there are buttons for SECURE YOUR ENDPOINT, a gear icon, and a question mark icon. Below the navigation bar, there's a section for "GraphQL Endpoint" with a "POST" button and a URL input field containing "https://learn-hasura-backend.herokuapp.com/v1alpha1/graphql". Underneath this, there's a "Request Headers" table with two rows: "content-type" set to "application/json" and "Authorization" set to a long token. A note says "Enter Key" next to the Authorization header. Below the headers is a "Explorer" tab and a "GraphiQL" tab, which is currently selected. The GraphiQL interface has three tabs: "Prettyify", "History", "Analyze", and "Explorer". The "Prettyify" tab is active. In the "Query" pane, there's a collapsed "auth0" object with fields like "email", "picture", and "auth0". The "Variables" pane is empty. The "Results" pane displays a JSON response:

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

```
{
  "data": {
    "auth0": {
      "email": "praveen@hasura.io",
      "picture": "https://s.gravatar.com/avatar/8d74d2e962705c093989404f9f683a924?s=480&r=pg&d=https%3A%2F%2Fcdn.auth0.com%2Favatars%2Fpr.png"
    }
  }
}
```

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

- [SendGrid SMTP Email API](#)
 - [Write the webhook](#)
- [Deploy](#)
- [Environment variables](#)

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+':'+process.env.SMTP_PASSWORD+'@'+process.env.SMTP_HOST);
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 ' + name,
    html: '<p>' + 'Hi, This is to notify that a new user has registered under the name of ' + name + '</p>'
  };
  // 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



[Deploy with Glitch](#)

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

- [Add Event Trigger](#)
- [Try it out](#)

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`.

The screenshot shows the 'Add a new event trigger' form in the Hasura console. The 'Trigger Name' field is filled with 'send_email'. In the 'Schema/Table' section, 'public' is selected from the schema dropdown and 'users' is selected from the table dropdown. Under the 'Operations' section, the 'Insert' checkbox is checked, while 'Update' and 'Delete' are unchecked. A note at the top of the operations section says: 'Note: Specifying the webhook URL via an environmental variable is recommended if you have different URLs for multiple environments.' Below this note, the 'Webhook URL' field contains the value 'https://sendgrid-send-email.event.gql'. At the bottom right of the form is a prominent yellow 'Add Event Trigger' button.

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.

The screenshot shows the Hasura GraphQL Engine interface. The top navigation bar includes links for GRAPHQL, DATA, REMOTE SCHEMAS, EVENTS (which is highlighted in yellow), SECURE YOUR ENDPOINT, and a help icon. The main content area is titled "Events > Manage > Triggers > send_email > Processed". A search bar at the top left says "search event triggers" and has a dropdown menu with "Event Triggers (1)" and an "Add" button. Below the search bar, the trigger name "send_email" is listed. The interface features several tabs: "Processed Events (1)" (which is active and highlighted in yellow), "Pending Events", "Invocation Logs", and "Modify". There are also "Filter" and "Sort" tools. A "Run query" button is highlighted in yellow, and a "Watch" button is also present. The main table displays one row of data:

	event_id	delivered	created_at
▼	bb4285ab-db68-485b-a705-64051053dd11	✓	Tue, May 7, 2019, 11:39:59 GMT+5:30

Below the table, there's a "Recent Invocations" section with a "Request" and "Response" tab. The "Request" tab shows a JSON object:

```
[{"event_id": "bb4285ab-db68-485b-a705-64051053dd11", "delivered": true, "created_at": "2019-05-07T11:39:59Z"}]
```

The "Response" tab shows a JSON object:

```
{}
```

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!