QueryAPI Context object

Overview

Thecontext object is a global object made available to developers building indexers withQueryAPI . It provides helper methods to interact with the resources spun up alongside the indexers, such as the GraphQL Endpoint and the database. There are also helper methods to allow specific API calls.

Under development The formatting and changes in this document are still in progress. These changes are not fully featured yet but will be by the time they hit production. Specifically,auto-complete and thecontext.db.TableName.methodName format.

Main Methods

tip All methods are asynchronous, hence why all examples have theawait keyword in front of the function call.

GraphQL

When an indexer is published, the SQL DDL written under theschema.sql tab is used to spin up a Postgres database. This DB is integrated with Hasura to provide a GraphQL endpoint. This endpoint can be used to interact with your database. Making calls to the Hasura GraphQL endpoint requires an API call, which is restricted in the environment. So, the GraphQL method allows calls to the endpoint related to the indexer.

tip The GraphQL method was previously the only way to interact with the database. Now, theDB methods provide a more accessible functionality. The GraphQL method is better suited for more complex queries and mutations. Also, more information about GraphQL calls can befound here .

Input

await context . graphql (operation , variables) The operation is a string formatted to match a GraphQL query. The variables are any data objects used in the query.

Example

const mutationData =

{ post :

{ account_id : accountId , block_height : blockHeight , block_timestamp : blockTimestamp , content : content , receipt_id : receiptId , } , } ;

// Call GraphQL mutation to insert a new post await context . graphql (mutation createPost(post: dataplatform_near_social_feed_posts_insert_input!){ insert_dataplatform_near_social_feed_posts_one(object: post) { account_id block_height } } , mutationData) ; The above is a snippet from the social feed indexer. In this example, you have a mutation (which mutates or changes data in the database, instead of a query that merely reads) calledcreatePost . The mutation name can be anything. You specify a variablepost and execute a graphQL method, which inserts thepost object and returnsaccount_id andblock_height from the newly inserted object. Finally, you pass inmutationData as the variable, which is automatically linked topost since it's the only field.

tip You can find other examples ofcontext.graphql in thesocial_feed indexers .

Set

Each indexer, by default, has a table calledindexer_storage . It has a field forkey andvalue , functioning like a key-value store. This table can, of course, be removed from the DDL before publishing. However, it kept theset method as an easy way to set some value for a key in that table. This method is used in the default code to set theheight on each invocation.

Input

await context . set (key , value)

Example

const h = block . header ( ) . height ; await context . set ("height" , h) ;

Log

Your indexing code is executed on a cloud compute instance. Therefore, things likeconsole.log are surfaced to the machine itself. To surfaceconsole.log statements not only to the machine but also back to you underIndexer Status , QueryAPI needs to write these logs to the logging table, which is separate from the developer's schema. Thishappens in the runner , and QueryAPI mapsconsole.log statements in the developer's code to call this function instead, so you can simply useconsole.log , and QueryAPI takes care of the rest.

DB

The DB object is a sub-object undercontext . It is accessed throughcontext.db . Previously, the GraphQL method was the only way to interact with the database. However, writing GraphQL queries and mutations is pretty complicated and overkill for simple interactions. So, simpler interactions are instead made available through thedb sub-object. This object is built by reading your schema, parsing its information, and generating methods for each table. See below for what methods are generated for each table. The format to access the below methods is as follows:context.db.[tableName].[methodName] . Concrete examples are also given below.

Note One thing to note is that the process where the code is read is not fully featured. If anALTER TABLE ALTER COLUMN statement is used in the SQL schema, for example, it will fail to parse. Should this failure occur, the context object will still be generated butdb methods will be unavailable. An error will appear on the page sayingtypes could not be generated . A more detailed error can be viewed in the browser's console.

DB Methods

These DB methods are generated when the schema is read. The tables in the schema are parsed, and methods are set under each table name. This makes using the object more intuitive and declarative.

If the schema is invalid for generating types, then an error will pop up both on screen and in the console. Here's an example:

Methods

note Except forupsert , all of the below methods are used insocial_feed_test indexer . However, keep in mind the current code uses the outdated call structure. An upcoming change will switch to the new method ofcontext.db.TableName.methodName instead ofcontext.db.methodName_tableName .

Insert

This method allows inserting one or more objects into the table preceding the function call. The inserted objects are then returned back with all of their information. Later on, we may implement returning specific fields but for now, we are returning them all. This goes for all methods.

Input

await context . db . TableName . insert (objects) Objects can be a single object or an array of them.

Example

const insertPostData =

{ account_id : accountId , block_height : blockHeight , block_timestamp : blockTimestamp , content : content , receipt_id : receiptId } ; // Insert new post to Posts table await context . db . Posts . insert (insertPostData) ; In this example, you insert a single object. But, if you want to insert multiple objects, then you just pass in an array with multiple objects. Such as[insertPostDataA, insertPostDataB] .

Select

This method returns rows that match the criteria included in the call. For now, QueryAPI only supports explicit matches. For example, providing{ colA: valueA, colB: valueB } means that rows wherecolA andcolB match thoseEXACT values will be returned.

There is also alimit parameter which specifies the maximum amount of objects to get. There are no guarantees on ordering. If there are 10 and the limit is 5, any of them might be returned.

Input

await context . db . TableName . select (fieldsToMatch , limit =

null) ThefieldsToMatch is an object that containscolumn names: value , where the value will need to be an exact match for that column. Thelimit parameter defaults tonull , which means no limit. If a value is provided, it overrides thenull value and is set to whatever was passed in. All matching rows up to the limit are returned.

Example

const posts =

await context . db . Posts . select ({ account_id : postAuthor , block_height : postBlockHeight } , 1) ; In this example, any rows in theposts table where theaccount_id column value matchespostAuthor AND block_height matchespostBlockheight will be returned.

Update

This method updates all rows that match thewhereObj values by setting theupdateObj values. It then returns all impacted rows. ThewhereObj is subject to the same restrictions as the select'swhereObj .

Input

await context . db . TableName . update (whereObj , updateObj)

Example

await context . db . Posts . update ({ id : post . id } , { last_comment_timestamp : currentTimestamp }) ; In this example, any rows in theposts table where theid column matches the valuepost.id will have theirlast_comment_timestamp column value overwritten to the value ofcurrentTimestamp . All impacted rows are then returned.

Upsert

Upsert is a combination of insert and update. First, the insert operation is performed. However, if the object already exists, the update portion is called instead. As a result, the input to the function are objects to be inserted, aconflictColumns object, which specifies which column values must conflict for the update operation to occur, and anupdateColumns which specifies which columns have their values overwritten by the incoming object's values. TheconflictColumns andupdateColumns parameters are both arrays. As usual, all impacted rows are returned.

Input

await context . db . upsert (objects , conflictColumns , updateColumns) The Objects parameter is either one or an array of objects. The other two parameters are arrays of strings. The strings should correspond to column names for that table.

Example

const insertPostDataA =

{ id : postId account_id : accountIdA , block_height : blockHeightA , block_timestamp : blockTimeStampA } ;

const insertPostDataB =

{ id : postId account_id : accountIdB , block_height : blockHeightB , block_timestamp : blockTimeStampB } ; // Insert new post to Posts table await context . db . Posts . upsert ([insertPostDataA , insertPostDataB] , ['account_id' , 'id'] , ['block_height' , 'block_timestamp') ; In this example, two objects are being inserted. However, if a row already exists in the table where theaccount_id andid are the same, thenblock_height andblock_timestamp would be overwritten in those rows to the value in the object in the call which is conflicting.

Delete

This method deletes all objects in the row that match the object passed in. Caution should be taken when using this method. It currently only supportsAND and exact match, just like in theselect method . That doubles as a safety measure against accidentally deleting a bunch of data. All deleted rows are returned so you can always insert them back if you get back more rows than expected. (Or reindex your indexer if needed)

Input

await context . db . TableName . delete (whereObj) As stated, only a single object is allowed.

Example

await context . db . delete_post_likes ({ account_id : likeAuthorAccountId , post_id : postId }) ; In this example, any rows whereaccount_id andpost_id match the supplied value are deleted. All deleted rows are returned.

Auto Complete

tip Autocomplete works while writing the schema and before publishing to the chain. In other words, you don't need to publish the indexer to get autocomplete. As mentioned, theDB methods are generated when the schema is read. In addition to that, TypeScript types are generated which represent the table itself. These types are set as the parameter types. This provides autocomplete and strong typing in the IDE. These restrictions are not enforced on the runner side and are instead mainly as a suggestion to help guide the developer to use the methods in a way that is deemed correct by QueryAPI.

Types You can also generate types manually. Clicking the<> button generates the types. It can be useful for debugging and iterative development while writing the schema.

Compatibility

By default, current indexers have acontext object included as a parameter in the top-levelasync function getBlock . This prevents autocomplete, as the localcontext object shadows the global one, preventing access to it. Users need to manually remove thecontext parameter from their indexers to get the autocomplete feature. For example:

async

function

getBlock (block :

Block , context)

{ should become

async

function

getBlock (block :

Block)

{

Examples

Here are some screenshots that demonstrate autocomplete on methods, strong typing, and field names:Edit this page Last updatedonFeb 9, 2024 bygagdiez Was this page helpful? Yes No

Previous Indexing Functions Next Access & Query Historical data