

QueryAPI Context object

Overview

The context object is a global object made available to developers building indexers with [QueryAPI](#). 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 the context.db.TableName.methodName format.

Main Methods

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

GraphQL

When an indexer is published, the SQL DDL written under the schema.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, the [DB 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 be [found 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:  
dataplatfrom_near_social_feed_posts_insert_input!){ insert_dataplatfrom_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) called createPost . The mutation name can be anything.  
You specify a variable post and execute a graphql method, which inserts the post object and returns account_id  
and block_height from the newly inserted object. Finally, you pass in mutationData as the variable, which is automatically  
linked to post since it's the only field.
```

tip You can find other examples of context.graphql in the [social feed indexers](#).

Set

Each indexer, by default, has a table called indexer_storage . It has a field for key and value , functioning like a key-value store. This table can, of course, be removed from the DDL before publishing. However, it kept the set 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 the height 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 like `console.log` are surfaced to the machine itself. To surface `console.log` statements not only to the machine but also back to you under `Indexer Status`, `QueryAPI` needs to write these logs to the logging table, which is separate from the developer's schema. This [happens in the runner](#), and `QueryAPI` maps `console.log` statements in the developer's code to call this function instead, so you can simply use `console.log`, and `QueryAPI` takes care of the rest.

DB

The `DB` object is a sub-object under `context`. It is accessed through `context.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 the `db` 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 an `ALTER 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 but `db` methods will be unavailable. An error will appear on the page saying `types 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 for [upsert](#), all of the below methods are used in [social feed test indexer](#). However, keep in mind the current code uses the outdated call structure. An upcoming change will switch to the new method of `context.db.TableName.methodName` instead of `context.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 where `colA` and `colB` match those `EXACT` values will be returned.

There is also a `limit` 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 )
```

 The `fieldsToMatch` is an object that contains column names: `value`, where the value will need to be an exact match for that column. The `limit` parameter defaults to `null`, which means no limit. If a value is provided, it overrides the `null` 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 the `posts` table where the `account_id` column value matches `postAuthor` AND `block_height` matches `postBlockHeight` will be returned.

Update

This method updates all rows that match the `whereObj` values by setting the `updateObj` values. It then returns all impacted rows. The `whereObj` is subject to the same restrictions as the `select`'s `whereObj` .

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 the `posts` table where the `id` column matches the value `post.id` will have their `last_comment_timestamp` column value overwritten to the value of `currentTimestamp` . 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, a `conflictColumns` object, which specifies which column values must conflict for the update operation to occur, and an `updateColumns` which specifies which columns have their values overwritten by the incoming object's values. The `conflictColumns` and `updateColumns` 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 the `account_id` and `id` are the same, then `block_height` and `block_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 supports AND and exact match, just like in the [select 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 where `account_id` and `post_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, the [DB 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](#)