GraphQL for Sui RPC (Beta)

BETA FEATURE

This content describes a beta feature or service. Beta features and services are in active development, so details are likely to change.

This feature or service is currently available in

- **Devnet**
- **Testnet**
- **Mainnet**

This section explains some of the common concepts when working with GraphQL, such as altering behavior using HTTP headers, re-using query snippets with variables and fragments, consuming paginated queries, and understanding and working within the limits enforced by the service.

Jump to the following sections for more details:

- Headers
- Variables
- Fragments
- Pagination
- Limits

For more details on GraphQL fundamentals, see the introductory documentation from GraphQL and GitHub.

Headers

The service accepts the following optional headers:

- `x-sui-rpc-version` to specify which RPC version to use (currently only one version is supported),
- `x-sui-rpc-show-usage` returns the response with extra query complexity information.

By default, each request returns the service's version in the response header: `x-sui-rpc-version`.

```
$ curl -i -X POST https://sui-mainnet.mystenlabs.com/graphql \
    --header 'x-sui-rpc-show-usage: true'                 \
    --header 'Content-Type: application/json'             \
    --data '{
        "query": "query { epoch { referenceGasPrice } }"
    }'
```

The response for the previous request looks similar to the following:

```
HTTP/2 200
content-type: application/json
content-length: 159
x-sui-rpc-version: 2024.1.0
date: Tue, 30 Jan 2024 23:50:43 GMT
via: 1.1 google
alt-svc: h3=":443"; ma=2592000,h3-29=":443"; ma=2592000

{
  "data": {
    "epoch": {
      "referenceGasPrice": "1000"
    }
  },
  "extensions": {
    "usage": {
      "inputNodes": 2,
      "outputNodes": 2,
      "depth": 2,
      "variables": 0,
      "fragments": 0,
      "queryPayload": 37
    }
  }
}
```

Variables

Variables offer a way to introduce dynamic inputs to a re-usable/static query. Declare variables in the parameters to a `query` or `mutation`, using the `$` symbol and its type (in this example `Int`), which must be a `scalar`, `enum`, or `input` type. In the query body, refer to it by its name (prefixed with the `$` symbol).

If you declare a variable but don't use it or define it (supply a value) in the query, the query fails to execute.

To learn more, read the GraphQL documentation on Variables.

In the following example, a variable supplies the ID of the epoch being queried.

```
query ($epochID: Int) {
  epoch(id: $epochID) {
    referenceGasPrice
  }
}
```

Variables:

```
{
   "epochID": 100
}
```

In the IDE

When using the online IDE, supply variables as a JSON object to the query in the **Variables** pane at the bottom of the main editing window. You receive a warning if you supply a variable but don't declare it.

In requests

When making a request to the GraphQL service (for example, using `curl`), pass the query and variables as two fields of a single JSON object:

```
$ curl -X POST https://sui-testnet.mystenlabs.com/graphql \
    --header 'Content-Type: application/json' \
    --data '{
      "query": "query ($epochID: Int) { epoch(id: $epochID) { referenceGasPrice
} }",
      "variables": { "epochID": 100 }
  }'
```

Fragments

Fragments are reusable units that you can include in queries as needed. To learn more, consult the official GraphQL documentation. The following example uses fragments to factor out a reusable snippet representing a Move value.

```
query DynamicField {
  object(
    address:
"0xb57fba584a700a5bcb40991e1b2e6bf68b0f3896d767a0da92e69de73de226ac"
  ) {
    dynamicField(
      name: {
        type: "0x2::kiosk::Listing",
        bcs: "NLArx1UJguOUYmXgNG8Pv8KbKXLjWtCi6i0Yeq1VhfwA",
      }
    ) {
      ...DynamicFieldSelect
      }
    }
  }

fragment DynamicFieldSelect on DynamicField {
  name {
    ...MoveValueFields
  }
  value {
    ...DynamicFieldValueSelection
  }
}

fragment DynamicFieldValueSelection on DynamicFieldValue {
  __typename
  ... on MoveValue {
    ...MoveValueFields
  }
  ... on MoveObject {
    hasPublicTransfer
    contents {
      ...MoveValueFields
    }
  }
}

fragment MoveValueFields on MoveValue {
  type {
    repr
  }
  data
  bcs
}
```

Pagination

GraphQL supports queries that fetch multiple kinds of data, potentially nested. For example, the following query retrieves the first 20 transaction blocks (along with the digest, the sender's address, the gas object used to pay for the transaction, the gas price, and the gas budget) after a specific transaction block at epoch `97`.

```
query {
  epoch(id: 97) {
    transactionBlocks(first: 10) {
      pageInfo {
        hasNextPage
        endCursor
      }
      nodes {
        digest
        sender {
          address
        }
        effects {
          gasEffects {
            gasObject {
              address
            }
          }
        }
        gasInput {
          gasPrice
          gasBudget
        }
      }
    }
  }
}
```

If there are too many transactions to return in a single response, the service applies a limit on the maximum page size for variable size responses (like the `transactionBlock` query) and you must fetch further results through pagination.

Connections

Fields that return a paginated response accept at least the following optional parameters:

- `first`, a limit on page size that is met by dropping excess results from the end.
- `after`, a cursor that bounds the results from below, exclusively.
- `last`, a limit on page size that is met by dropping excess results from the start.
- `before`, a cursor that bounds the results from above, exclusively.

They also return a type that conforms to the GraphQL Cursor Connections Specification, meaning its name ends in `Connection`, and it contains at least the following fields:

- `pageInfo`, of type PageInfo, which indicates whether there are more pages before or after the page returned.
- `nodes`, the content of the paginated response, as a list of the type being paginated (`TransactionBlock` in the previous example).
- `edges`, similar to `nodes` but associating each node with its cursor.

Cursors

Cursors are opaque identifiers for paginated results. The only valid source for a cursor parameter (like `after` and `before`) is a cursor field from a previous paginated response (like `PageInfo.startCursor`, `PageInfo.endCursor`, or `Edge.cursor`). The underlying format of the cursor is an implementation detail, and is not guaranteed to remain fixed across versions of the GraphQL service, so do not rely on it -- generating cursors out of thin air is not expected or supported.

Cursors are used to bound results from below (with `after`) and above (with `before`). In both cases, the bound is exclusive, meaning it does not include the result that the cursor points to in the bounded region.

Consistency

Cursors also guarantee **consistent** pagination. If the first paginated query reads the state of the network at checkpoint `X`, then a future call to fetch the next page of results using the cursors returned by the first query continues to read from the network at checkpoint `X`, even if data for future checkpoints is now available.

This property requires that cursors that are used together (for example when supplying an `after` and `before` bound) are fixed on the same checkpoint, otherwise the query produces an error.

Available range

The GraphQL service does not support consistent pagination for arbitrarily old cursors. A cursor can grow stale, if the checkpoint it is from is no longer in the **available range**. You can query the upper- and lower-bounds of that range as follows:

```
{
  availableRange {
    first { sequenceNumber }
    last { sequenceNumber }
  }
}
```

The results are the first and last checkpoint for which pagination continues to work and produce a consistent result. At the time of writing the available range offers a 5- to 15-minute buffer period to finish pagination.

Page limits

After results are bounded using cursors, a page size limit is applied using the `first` and `last` parameters. The service requires these parameters to be less than or equal to the max page size limit, and if you provide neither, it selects a default. In addition to setting a limit, `first` and `last` control where excess elements are discarded from. For example, if there are `10` potential results -- `R0`, `R1`, ..., `R9` -- after cursor bounds have been applied, then

- a limit of `first: 3` would select `R0`, `R1`, `R2`, and
- a limit of `last: 3` would select `R7`, `R8`, `R9`.

INFO

It is an error to apply both a `first` and a `last` limit.

Examples

To see these principles put into practice, consult the examples for paginating forwards and paginating backwards in the getting started guide.

Limits

The GraphQL service for Sui RPC is rate-limited on all available instances to keep network throughput optimized and to protect against excessive or abusive calls to the service.

Rate limits

Queries are rate-limited at the number of attempts per minute to ensure high availability of the service to all users.

Query limits

In addition to rate limits, queries are also validated against a number of rules on their complexity, such as the number of nodes, the depth of the query, or their payload size. Query the `serviceConfig` field to retrieve these limits. An example of how to query for some of the available limits follows:

```
{
  serviceConfig {
    maxQueryDepth
    maxQueryNodes
    maxOutputNodes
    maxDbQueryCost
    defaultPageSize
    maxPageSize
    requestTimeoutMs
    maxQueryPayloadSize
    maxTypeArgumentDepth
    maxTypeArgumentWidth
    maxTypeNodes
    maxMoveValueDepth
  }
}
```

Related links

- Querying Sui RPC with GraphQL: Gets you started using GraphQL to query the Sui RPC for on-chain data.
- Migrating to GraphQL: Guides you through migrating Sui RPC projects from JSON-RPC to GraphQL.
- GraphQL for Sui RPC: Auto-generated GraphQL reference for Sui RPC.

Edit this page