The Neo4j HTTP API Docs v4.2

Table of Contents

Introduction . . . 2
Discovery API . . . 3
    Root discovery . . . 3
Cypher transaction API . . . 4
    Transaction flow . . . 4
    Query format . . . 5
    Result formats . . . 6
    Begin a transaction . . . 11
    Run queries inside a transaction . . . 12
    Keeping transactions alive with an empty statement . . . 13
    Commit a transaction . . . 13
    Rollback an open transaction . . . 14
    Begin and commit a transaction in one request . . . 14
    Execute multiple statements . . . 15
    Include query statistics . . . 16
    Return results in graph format . . . 17
    Expired transactions . . . 18
    Handling errors . . . 18
    Handling errors in an open transaction . . . 19
Authentication and authorization . . . 21
    Introduction . . . 21
    Missing authorization . . . 21
    Incorrect authentication . . . 21

This is the manual for the Transactional Cypher HTTP endpoint for Neo4j version 4.2, authored by the Neo4j Team.

This manual covers the following areas:

- Introduction
- Discovery API
- Cypher transaction API
- Authentication and authorization

For information on using the HTTP API to monitor a Causal Cluster, see Operations Manual Endpoints for status information.

Who should read this?

This manual is written for the developer of a client application which accesses Neo4j through the HTTP API.

Introduction

This chapter introduces the Neo4j HTTP API, and how to effectively use it.

The Neo4j transactional HTTP endpoint allows you to execute a series of Cypher statements within the scope of a transaction. The transaction may be kept open across multiple HTTP requests, until the client chooses to commit or roll back. Each HTTP request can include a list of statements, and for convenience you can include statements along with a request to begin or commit a transaction.

The server guards against orphaned transactions by using a timeout. If there are no requests for a given transaction within the timeout period, the server will roll it back. You can configure the timeout in the server configuration, by setting Operations Manual → `dbms.rest.transaction.idle_timeout` to the number of seconds before timeout. The default timeout is 60 seconds.

- Literal line breaks are not allowed inside Cypher statements.
- Cypher queries with `USING PERIODIC COMMIT` (see Cypher Manual → `PERIODIC COMMIT` query hint) may only be executed when creating a new transaction and immediately committing it with a single HTTP request (see Cypher Manual → Begin and commit a transaction in one request for how to do that).
- When a request fails the transaction will be rolled back. By checking the result for the presence/absence of the `transaction` key you can figure out if the transaction is still open.

In order to speed up queries in repeated scenarios, try not to use literals but replace them with parameters wherever possible. This will let the server cache query plans. See Cypher Manual → Parameters for more information.

Responses from the HTTP API can be transmitted as JSON streams, resulting in better performance and lower memory overhead on the server side. To use streaming, supply the header `X-Stream: true` with each request.

Discovery API

This chapter describes the actions that can be performed using the Discovery HTTP endpoint.

Root discovery

Each server provides a root discovery URI that lists a basic index of other URIs, as well as version information.

Example request

```
GET http://localhost:7474/
Accept: application/json
```

Example response

```
200 OK
Content-Type: application/json

{
  "bolt_direct": "bolt://localhost:7687",
  "bolt_routing": "neo4j://localhost:7687",
  "cluster": "http://localhost:7687/db/{databaseName}/cluster",
  "transaction": "http://localhost:7687/db/{databaseName}/tx",
  "neo4j_version": "4.0.0",
  "neo4j_edition": "enterprise"
}
```

Cypher transaction API

This chapter describes the actions that can be performed using the Cypher transaction HTTP endpoint.

This chapter includes the following sections:

Concepts:

- Transaction flow
- Query format
- Result formats

Using the API:

- Begin a transaction
- Run queries inside a transaction
- Keeping transactions alive with an empty statement
- Commit a transaction
- Rollback an open transaction
- Begin and commit a transaction in one request

Additional actions:

- Execute multiple statements
- Include query statistics
- Return results in graph format

Error handling:

- Expired transactions
- Handling errors
- Handling errors in an open transaction

Transaction flow

Cypher transactions are managed over several distinct URIs that are designed to be used in a prescribed pattern. Facilities are provided to carry out the full transaction cycle over a single HTTP request, or over multiple HTTP requests.

The overall flow is illustrated below, with each box representing a separate HTTP request:

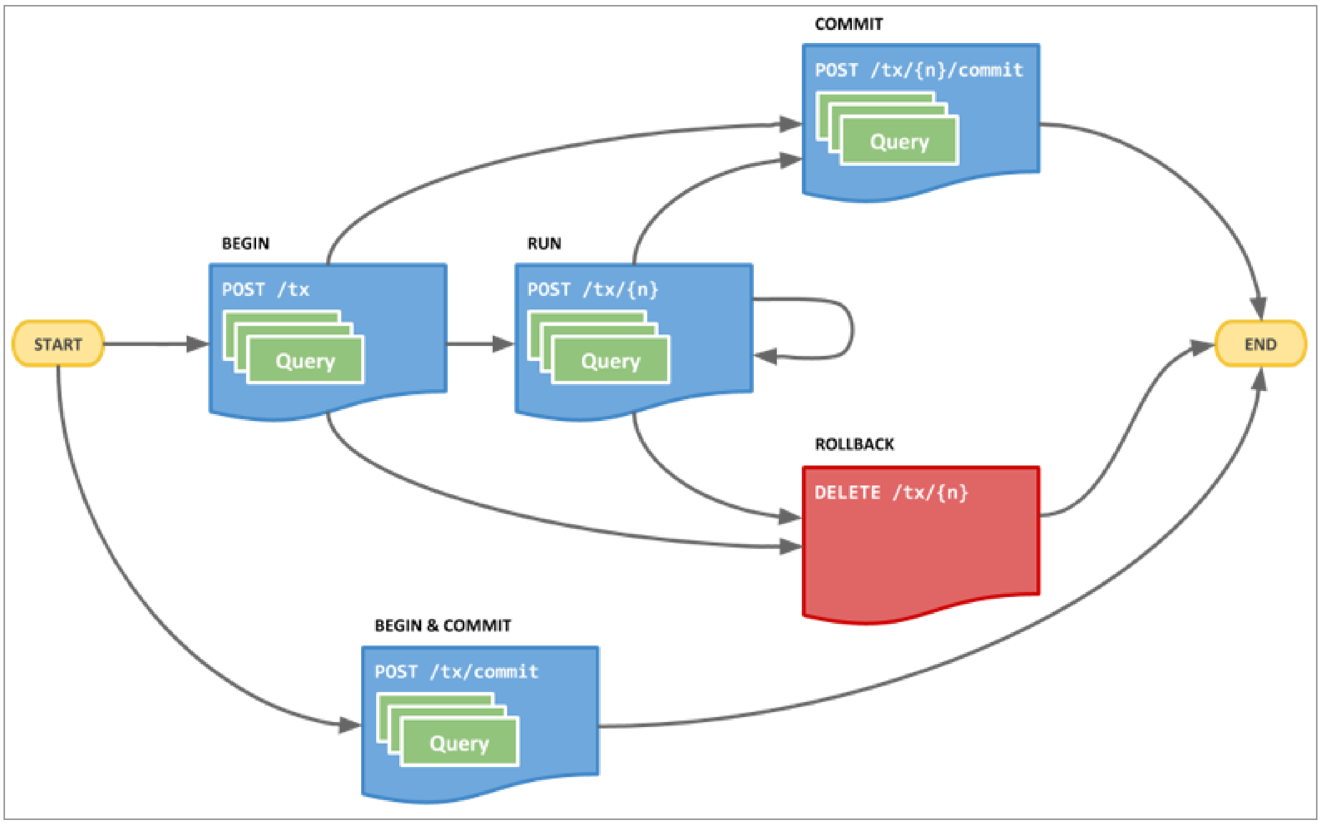

Figure 1. Cypher transaction flow

Transaction lifetime

The state of each transaction is maintained on the server on which the transaction began. Transactions expire automatically after a period of inactivity. By default this is 60 seconds.

To keep a transaction alive without submitting new queries, an empty statement list can be posted to the `/tx/{n}` URI.

Query format

All transaction POST requests can accept one or more Cypher queries within the request payload. This enables a large amount of flexibility in how, and when, queries are sent, and can help to reduce the number of individual HTTP requests overall.

The payload is sent as JSON with the following general structure:

```
{
  "statements": [
    {
      "statement": "...",
      "parameters": {...}
    },
    {
      "statement": "...",
      "parameters": {...}
    },
    ...
  ]
}
```

For example:

```
{
    "statements": [
        {
            "statement": "CREATE (n $props) RETURN n",
            "parameters": {
                "props": {
                    "name": "My Node"
                }
            }
        },
                {
            "statement": "CREATE (n $props) RETURN n",
            "parameters": {
                "props": {
                    "name": "Another Node"
                }
            }
        }
    ]
}
```

Parameters are included as key-value pairs, with each value adopting a type that corresponds to an entry in the mapping table below:

Table 1. HTTP parameter type mappings

JSON Type	Cypher Type
`null`	Null
`boolean`	Boolean
`number`	Float
`string`	String
`array`	List
`object`	Map

Result formats

Default

This format returns JSON with an embedded `results` element. To request this format, place `application/json` in the `Accept` header. This format is the default returned if no `Accept` header is provided.

```
{
    "results": [
        {
            "columns": [],
            "data": [
                {
                    "row": [ row-data ],
                    "meta": [ metadata ]
                },
                {

                }
            ]
        },
        {
             //another statement’s results
        }
    ]
}
```

For example, running the query `UNWIND range(0,2,1) as number RETURN number` will return the following results:

```
{
    "results": [
        {
            "columns": [
                "number"
            ],
            "data": [
                {
                    "row": [
                        0
                    ],
                    "meta": [
                        null
                    ]
                },
                {
                    "row": [
                        1
                    ],
                    "meta": [
                        null
                    ]
                },
                {
                    "row": [
                        2
                    ],
                    "meta": [
                        null
                    ]
                }
            ]
        }
    ],
    // other transactional data
}
```

Jolt

Jolt, short for *JSON Bolt*, is a JSON-based format which encloses the response value’s type together with the value inside a singleton object.

For example:

```
{ "Z": "2" }
```

This labels the value `2` as an integer type.

This format can be returned when adding `application/vnd.neo4j.jolt+json-seq` to the request's `Accept` header.

Strict and sparse

There are two modes of Jolt that can be returned:

- **Strict mode**, where all values are paired with their type.
- **Sparse mode**, which omits typing pairing on values which can suitably be matched to JSON types.

By default, the *sparse mode* is returned. To enable *strict mode*, pass `application/vnd.neo4j.jolt+json-seq;strict=true` in the `Accept` header.

Jolt types

Base types

Type Label	Type	Example
(N/A)	null	`null`
`?`	Boolean	`{ "?": "true" }`
`Z`	Integer	`{ "Z": "123" }`
`R`	Float/Real	`{ "R": "9.87" }` [1]
`U`	String	`{ "U": "A string" }`
`T`	Date/Time	`{ "T": "2002-04-16T12:34:56"}`
`@`	Geospatial	`{"@": "POINT (30 10)"}`
`#`	Base64	`{ "#": "AaBb" }`

[1]*The type label* `R` *is used both to indicate floating point numbers and integers that are outside the range of 32-bit signed integers.*

Composite types

Type Label	Type	Example
`[]`	List	`{ "[]": [ { "Z": "123" }, … ] }`
`{}`	Dictionary	`{ "{}": { "name": { "U": "Jeff"}, …}}`

Entity types

Node

```
{"()": [ node_id, [ node_labels], {"prop1": "value1", "prop2": "value2"}]}
```

For example:

```
{
 "()": [
   4711,
   [
     "A",
     "B"
   ],
   {
     "prop1": {
       "Z": "1"
     },
     "prop2": {
       "U": "Hello"
     }
   }
 ]
}
```

Relationships

```
{"->": [ rel_id, start_node_id, rel_type, end_node_id, {properties}]}
{"<-": [ rel_id, end_node_id, rel_type, start_node_id, {properties}]}
```

For example:

```
{
 "->": [
   4711,
   123,
   "KNOWS",
   124,
   {
     "since": {
       "Z": "1999"
     }
   }
 ]
}
```

Paths

```
{"..": [{node_1},{rel_1},{node_2},...,{node_n},{rel_n},{node_n+1}]}
```

For example:

```
{
 "..": [
   {
     "()": [
       111,
       [],
       {}
     ]
   },
   {
     "->": [
       9090,
       111,
       "KNOWS",
       222,
       {
         "since": {
           "Z": "1999"
         }
       }
     ]
   },
   {
     "()": [
       222,
       [],
       {}
     ]
   }
 ]
}
```

Container format

Jolt results will be returned in a new container format based on events. A typical response will contain:

```
{"header":{"fields":["name","age"]}}
{"data":[{"U":"Bob"},{"Z":"30"}]}
{"data":[{"U":"Alice"},{"Z":"40"}]}
{"data":[{"U":"Eve"},{"Z":"50"}]}
...
{"summary":{}}
{"info":{"commit":"commit/uri/1"}}
```

Each event is a separate JSON document separated by a single new line character:

Event	Function
`header`	Marks the start of a result set for a statement, and contains query fields.
`data`	For each record returned in the result set there will be a `data` json object. Depending on the query, each query can return multiple `data` objects. The order of values in the array match the fields received in the header.
`summary`	Marks the end of a result set for a statement. Can contain query plan information if requested.

Event	Function
`info`	Final event to appear after processing all statements (unless an error has occurred), and can contain transaction information (e.g. a commit URI).
`error`	Errors which occur during the processing of the transaction.

Multiple result sets in a request

When there are multiple queries in a single request there will be multiple `header`, `data`, and `summary` outputs for each query.

For example, posting the following request:

```
{
  "statements" : [
    { "statement" : "RETURN 1 as resultA"},
    { "statement" : "UNWIND range(1,3,1) as resultB RETURN resultB"}
  ]
}
```

will yield the following result response:

```
{"header":{"fields":["resultA"]}}
{"data":[{"Z":"1"}]}
{"summary":{}}
{"header":{"fields":["resultB"]}}
{"data":[{"Z":"1"}]}
{"data":[{"Z":"2"}]}
{"data":[{"Z":"3"}]}
{"summary":{}}
{"info":{}}
```

Results sets will be returned in the same order as passed in the original request.

Begin a transaction

A new transaction can be started by posting zero or more Cypher queries to the transaction endpoint. The server will respond with the results of your queries, as well as the location of your new transaction.

Transactions expire automatically after a period of inactivity (i.e. queries and a commit). By default this is 60 seconds.

To keep a transaction alive without submitting new queries, an empty statement list can be posted to the transaction URI.

Example request

- **POST** http://localhost:35105/db/neo4j/tx
- **Accept:** application/json;charset=UTF-8
- **Content-Type:** application/json

```
{
  "statements" : [ {
    "statement" : "CREATE (n $props) RETURN n",
    "parameters" : {
      "props" : {
        "name" : "My Node"
      }
    }
  } ]
}
```

Example response

- **201:** Created
- **Content-Type:** application/json;charset=utf-8
- **Location:** http://localhost:35105/db/neo4j/tx/14

```
{
  "results" : [ {
    "columns" : [ "n" ],
    "data" : [ {
      "row" : [ {
        "name" : "My Node"
      } ],
      "meta" : [ {
        "id" : 10,
        "type" : "node",
        "deleted" : false
      } ]
    } ]
  } ],
  "errors" : [ ],
  "commit" : "http://localhost:35105/db/neo4j/tx/14/commit",
  "transaction" : {
    "expires" : "Wed, 17 Feb 2021 16:32:39 GMT"
  }
}
```

Run queries inside a transaction

Once you have an open transaction by calling `db/{name}/tx`, you can run additional statements that form part of your transaction by calling the newly created transaction endpoint. The endpoint will be in the form `db/{name}/tx/{txid}`, where `txid` is provided in the response of the initial call to begin the transaction.

Example request

- **POST** http://localhost:35105/db/neo4j/tx/16
- **Accept:** application/json;charset=UTF-8
- **Content-Type:** application/json

```
{
  "statements" : [ {
    "statement" : "CREATE (n) RETURN n"
  } ]
}
```

Example response

- **200:** OK
- **Content-Type:** application/json;charset=utf-8

```
{
  "results" : [ {
    "columns" : [ "n" ],
    "data" : [ {
      "row" : [ { } ],
      "meta" : [ {
        "id" : 11,
        "type" : "node",
        "deleted" : false
      } ]
    } ]
  } ],
  "errors" : [ ],
  "commit" : "http://localhost:35105/db/neo4j/tx/16/commit",
  "transaction" : {
    "expires" : "Wed, 17 Feb 2021 16:32:39 GMT"
  }
}
```

Keeping transactions alive with an empty statement

If you need to extend the timeout while processing a transaction, you can send a POST to the transaction's endpoint with a blank HTTP body.

For example:

Example request

- **POST** http://localhost:35105/db/neo4j/tx/2
- **Accept:** application/json;charset=UTF-8
- **Content-Type:** application/json

```
{
  "statements" : [ ]
}
```

Example response

- **200:** OK
- **Content-Type:** application/json;charset=utf-8

```
{
  "results" : [ ],
  "errors" : [ ],
  "commit" : "http://localhost:35105/db/neo4j/tx/2/commit",
  "transaction" : {
    "expires" : "Wed, 17 Feb 2021 16:32:38 GMT"
  }
}
```

Commit a transaction

When you have executed all the statements for the transaction, and want to commit the changes to the database, you can use `POST db/{name}/tx/{txid}/commit`, which can also include any final statements to execute before committing:

Example request

- **POST** http://localhost:35105/db/neo4j/tx/commit

- **Accept:** application/json;charset=UTF-8
- **Content-Type:** application/json

```
{
  "statements" : [ {
    "statement" : "MATCH (n) WHERE ID(n) = $nodeId RETURN n",
    "parameters" : {
      "nodeId" : 5
    }
  } ]
}
```

Example response

- **200:** OK
- **Content-Type:** application/json;charset=utf-8

```
{
  "results" : [ {
    "columns" : [ "n" ],
    "data" : [ {
      "row" : [ { } ],
      "meta" : [ {
        "id" : 5,
        "type" : "node",
        "deleted" : false
      } ]
    } ]
  } ],
  "errors" : [ ]
}
```

Rollback an open transaction

Given that you have an open transaction, you can send a rollback request. The server will roll back the transaction. Any attempt to run additional statements in this transaction will fail immediately.

Example request

- **DELETE** http://localhost:35105/db/neo4j/tx/3
- **Accept:** application/json;charset=UTF-8

Example response

- **200:** OK
- **Content-Type:** application/json;charset=utf-8

```
{
  "results" : [ ],
  "errors" : [ ]
}
```

Begin and commit a transaction in one request

If there is no need to keep a transaction open across multiple HTTP requests, you can begin a transaction, execute statements, and commit within a single HTTP request.

Example request

- **POST** http://localhost:35105/db/neo4j/tx/commit
- **Accept:** application/json;charset=UTF-8
- **Content-Type:** application/json

```
{
  "statements" : [ {
    "statement" : "MATCH (n) WHERE ID(n) = $nodeId RETURN n",
    "parameters" : {
      "nodeId" : 6
    }
  } ]
}
```

Example response

- **200:** OK
- **Content-Type:** application/json;charset=utf-8

```
{
  "results" : [ {
    "columns" : [ "n" ],
    "data" : [ {
      "row" : [ { } ],
      "meta" : [ {
        "id" : 6,
        "type" : "node",
        "deleted" : false
      } ]
    } ]
  } ],
  "errors" : [ ]
}
```

Execute multiple statements

You can send multiple Cypher statements in the same request. The response will contain the result of each statement.

Example request

- **POST** http://localhost:35105/db/neo4j/tx/commit
- **Accept:** application/json;charset=UTF-8
- **Content-Type:** application/json

```
{
  "statements" : [ {
    "statement" : "MATCH (n) WHERE ID(n) = $nodeId RETURN n",
    "parameters" : {
      "nodeId" : 2
    }
  } ]
}
```

Example response

- **200:** OK
- **Content-Type:** application/json;charset=utf-8

```
{
  "results" : [ {
    "columns" : [ "n" ],
    "data" : [ {
      "row" : [ { } ],
      "meta" : [ {
        "id" : 2,
        "type" : "node",
        "deleted" : false
      } ]
    } ]
  } ],
  "errors" : [ ]
}
```

Include query statistics

By setting `includeStats` to `true` for a statement, query statistics will be returned for it.

Example request

- **POST** http://localhost:35105/db/neo4j/tx/commit
- **Accept:** application/json;charset=UTF-8
- **Content-Type:** application/json

```
{
  "statements" : [ {
    "statement" : "CREATE (n) RETURN id(n)",
    "includeStats" : true
  } ]
}
```

Example response

- **200:** OK
- **Content-Type:** application/json;charset=utf-8

```
{
  "results" : [ {
    "columns" : [ "id(n)" ],
    "data" : [ {
      "row" : [ 4 ],
      "meta" : [ null ]
    } ],
    "stats" : {
      "contains_updates" : true,
      "nodes_created" : 1,
      "nodes_deleted" : 0,
      "properties_set" : 0,
      "relationships_created" : 0,
      "relationship_deleted" : 0,
      "labels_added" : 0,
      "labels_removed" : 0,
      "indexes_added" : 0,
      "indexes_removed" : 0,
      "constraints_added" : 0,
      "constraints_removed" : 0,
      "contains_system_updates" : false,
      "system_updates" : 0
    }
  } ],
  "errors" : [ ]
}
```

Return results in graph format

If you want to understand the graph structure of nodes and relationships returned by your query, you can specify the `graph` results data format. This is useful when you want to visualize the graph structure. The format collates all the nodes and relationships from all columns of the result, and also flattens collections of nodes and relationships, including paths.

Example request

- **POST** http://localhost:35105/db/neo4j/tx/commit
- **Accept:** application/json;charset=UTF-8
- **Content-Type:** application/json

```
{
  "statements" : [ {
    "statement" : "CREATE ( bike:Bike { weight: 10 } ) CREATE ( frontWheel:Wheel { spokes: 3 } ) CREATE (
backWheel:Wheel { spokes: 32 } ) CREATE p1 = (bike)-[:HAS { position: 1 } ]->(frontWheel) CREATE p2 =
(bike)-[:HAS { position: 2 } ]->(backWheel) RETURN bike, p1, p2",
    "resultDataContents" : [ "row", "graph" ]
  } ]
}
```

Example response

- **200:** OK
- **Content-Type:** application/json;charset=utf-8

```
{
  "results" : [ {
    "columns" : [ "bike", "p1", "p2" ],
    "data" : [ {
      "row" : [ {
        "weight" : 10
      }, [ {
        "weight" : 10
      }, {
        "position" : 1
      }, {
        "spokes" : 3
      } ], [ {
        "weight" : 10
      }, {
        "position" : 2
      }, {
        "spokes" : 32
      } ] ],
      "meta" : [ {
        "id" : 7,
        "type" : "node",
        "deleted" : false
      }, [ {
        "id" : 7,
        "type" : "node",
        "deleted" : false
      }, {
        "id" : 0,
        "type" : "relationship",
        "deleted" : false
      }, {
        "id" : 8,
        "type" : "node",
        "deleted" : false
      } ], [ {
        "id" : 7,
        "type" : "node",
        "deleted" : false
      }, {
        "id" : 1,
        "type" : "relationship",
        "deleted" : false
      }, {
        "id" : 9,
        "type" : "node",
        "deleted" : false
      } ] ],
      "graph" : {
        "nodes" : [ {
          "id" : "7",
          "labels" : [ "Bike" ],
          "properties" : {
            "weight" : 10
          }
        }, {
          "id" : "8",
          "labels" : [ "Wheel" ],
          "properties" : {
            "spokes" : 3
          }
        }, {
          "id" : "9",
          "labels" : [ "Wheel" ],
          "properties" : {
            "spokes" : 32
          }
        } ],
        "relationships" : [ {
          "id" : "0",
          "type" : "HAS",
          "startNode" : "7",
          "endNode" : "8",
          "properties" : {
            "position" : 1
          }
        }, {
          "id" : "1",
          "type" : "HAS",
          "startNode" : "7",
          "endNode" : "9",
          "properties" : {
            "position" : 2
          }
        } ]
      }
    } ]
  } ],
  "errors" : [ ]
}
```

Expired transactions

If an attempt is made to commit a transaction which has timed out, you will see the following error:

```
404 Not Found
Content-Type: application/json

{
    "results": [],
    "errors": [
        {
            "code": "Neo.ClientError.Transaction.TransactionNotFound",
            "message": "Unrecognized transaction id. Transaction may have timed out and been rolled back."
        }
    ]
}
```

Handling errors

The result of any request against the transaction endpoint is streamed back to the client. Therefore, the server does not know whether the request will be successful or not when it sends the HTTP status code.

Because of this, all requests against the transactional endpoint will return `200` or `201` status code, regardless of whether statements were successfully executed. At the end of the response payload, the server includes a list of errors that occurred while executing statements. If the list is empty, the request completed successfully.

If errors occur while executing statements, the server will roll back the transaction.

In this example, we send an invalid statement to the server in order to demonstrate error handling.

For more information on the status codes, see neo4j-status-codes-4.2.pdf.

Example request

- **POST** http://localhost:35105/db/neo4j/tx/15/commit
- **Accept:** application/json;charset=UTF-8
- **Content-Type:** application/json

```
{
  "statements" : [ {
    "statement" : "This is not a valid Cypher Statement."
  } ]
}
```

Example response

- **200:** OK
- **Content-Type:** application/json;charset=utf-8

```
{
  "results" : [ ],
  "errors" : [ {
    "code" : "Neo.ClientError.Statement.SyntaxError",
    "message" : "Invalid input 'This': expected \n  \"RETURN\"\n  \"CREATE\"\n  \"DELETE\"\n  \"SET\"\n
\"REMOVE\"\n  \"DETACH\"\n  \"MATCH\"\n  \"WITH\"\n  \"UNWIND\"\n  \"USE\"\n  \"CALL\"\n  \"LOAD\"\n
\"FROM\"\n  \"FOREACH\"\n  \"MERGE\"\n  \"OPTIONAL\"\n  \"USING\" (line 1, column 1 (offset: 0))\n\"This
is not a valid Cypher Statement.\"\n ^"
  } ],
  "commit" : "http://localhost:35105/db/neo4j/tx/15/commit"
}
```

Handling errors in an open transaction

If there is an error in a request, the server will roll back the transaction. You can tell if the transaction is still open by inspecting the response for the presence/absence of the `transaction` key.

Example request

- **POST** http://localhost:35105/db/neo4j/tx/13
- **Accept:** application/json;charset=UTF-8
- **Content-Type:** application/json

```
{
  "statements" : [ {
    "statement" : "This is not a valid Cypher Statement."
  } ]
}
```

Example response

- **200:** OK
- **Content-Type:** application/json;charset=utf-8

```
{
  "results" : [ ],
  "errors" : [ {
    "code" : "Neo.ClientError.Statement.SyntaxError",
    "message" : "Invalid input 'This': expected \n  \"RETURN\"\n  \"CREATE\"\n  \"DELETE\"\n  \"SET\"\n
\"REMOVE\"\n  \"DETACH\"\n  \"MATCH\"\n  \"WITH\"\n  \"UNWIND\"\n  \"USE\"\n  \"CALL\"\n  \"LOAD\"\n
\"FROM\"\n  \"FOREACH\"\n  \"MERGE\"\n  \"OPTIONAL\"\n  \"USING\" (line 1, column 1 (offset: 0))\n\"This
is not a valid Cypher Statement.\"\n ^"
  } ],
  "commit" : "http://localhost:35105/db/neo4j/tx/13/commit"
}
```

Authentication and authorization

This chapter describes the authentication and authorization required to use the HTTP API.

This chapter includes the following sections:

- Introduction
- Missing authorization
- Incorrect authentication

Introduction

Authentication and authorization are enabled by default in Neo4j (refer to Operations Manual → Authentication and authorization). With authentication and authorization enabled, requests to the HTTP API must be authorized using the username and password of a valid user.

Missing authorization

If an `Authorization` header is not supplied, the server will reply with an error.

Example request

- **POST** http://localhost:44923/db/neo4j/tx/commit
- **Accept:** application/json;charset=UTF-8
- **Content-Type:** application/json

```
{
  "statements" : [ {
    "statement" : "CREATE (n:MyLabel) RETURN n"
  } ]
}
```

Example response

- **401:** Unauthorized
- **Content-Type:** application/json;charset=utf-8
- **WWW-Authenticate:** Basic realm="Neo4j"

```
{
  "errors" : [ {
    "code" : "Neo.ClientError.Security.Unauthorized",
    "message" : "No authentication header supplied."
  } ]
}
```

If authentication and authorization have been disabled, HTTP API requests can be sent without an `Authorization` header.

Incorrect authentication

If an incorrect username or password is provided, the server replies with an error.

Example request

- **POST** http://localhost:35851/db/neo4j/tx/commit
- **Accept:** application/json;charset=UTF-8
- **Authorization:** Basic bmVvNGo6aW5jb3JyZWN0
- **Content-Type:** application/json

```
{
  "statements" : [ {
    "statement" : "CREATE (n:MyLabel) RETURN n"
  } ]
}
```

Example response

- **401:** Unauthorized
- **Content-Type:** application/json;charset=utf-8
- **WWW-Authenticate:** Basic realm="Neo4j"

```
{
  "errors" : [ {
    "code" : "Neo.ClientError.Security.Unauthorized",
    "message" : "Invalid username or password."
  } ]
}
```