HTTP API

Table of Contents

Introduction . . . 2
Using the HTTP API . . . 3
- Begin and commit a transaction in one request . . . 3
- Execute multiple statements . . . 4
- Begin a transaction . . . 5
- Execute statements in an open transaction . . . 6
- Reset transaction timeout of an open transaction . . . 7
- Commit an open transaction . . . 8
- Rollback an open transaction . . . 8
- Include query statistics . . . 9
- Return results in graph format . . . 9
- Handling errors . . . 11
- Handling errors in an open transaction . . . 12

Authentication and authorization . . . 14
- Introduction Deprecated . . . 14
- Authenticate to access the server Deprecated . . . 14
- Missing authorization Deprecated . . . 14
- Incorrect authentication Deprecated . . . 15
- Required password changes Deprecated . . . 15
- User status on first access Deprecated . . . 16
- User status Deprecated . . . 16
- Changing the user password Deprecated . . . 17

Neo4j v3.5

Transactional Cypher HTTP endpoint.

This manual covers the following areas:

- Introduction
- Using the HTTP API
- Authentication and authorization

Who should read this?

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

Introduction

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 → Configuration settings `dbms.rest.transaction.idle_timeout` to the number of seconds before timeout. The default timeout is `60` seconds.

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.

- Literal line breaks are not allowed inside Cypher statements.
- Open transactions are not shared among members of an HA cluster. Therefore, if you use this endpoint in an HA cluster, you must ensure that all requests for a given transaction are sent to the same Neo4j instance.
- Cypher queries with `USING PERIODIC COMMIT` (see 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.

Using the HTTP API

> *This section describes the actions that can be performed using the Transactional Cypher HTTP endpoint.*

This section includes the following topics:

- Begin and commit a transaction in one request
- Execute multiple statements
- Begin a transaction
- Execute statements in an open transaction
- Reset transaction timeout of an open transaction
- Commit an open transaction
- Rollback an open transaction
- Include query statistics
- Return results in graph format
- Handling errors
- Handling errors in an open transaction

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:7474/db/data/transaction/commit
- **Accept:** application/json;charset=UTF-8
- **Content-Type:** application/json

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

Example response

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

```
{
  "results" : [ {
    "columns" : [ "id(n)" ],
    "data" : [ {
      "row" : [ 6 ],
      "meta" : [ null ]
    } ]
  } ],
  "errors" : [ ]
}
```

Execute multiple statements

It is possible to send multiple Cypher statements in the same request. The response will contain the result of each statement.

Example request

- **POST** http://localhost:7474/db/data/transaction/commit
- **Accept:** application/json;charset=UTF-8
- **Content-Type:** application/json

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

Example response

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

```
{
  "results" : [ {
    "columns" : [ "id(n)" ],
    "data" : [ {
      "row" : [ 2 ],
      "meta" : [ null ]
    } ]
  }, {
    "columns" : [ "n" ],
    "data" : [ {
      "row" : [ {
        "name" : "My Node"
      } ],
      "meta" : [ {
        "id" : 3,
        "type" : "node",
        "deleted" : false
      } ]
    } ]
  } ],
  "errors" : [ ]
}
```

Begin a transaction

You begin a new transaction by posting zero or more Cypher statements to the transaction endpoint. The server will respond with the result of your statements, as well as the location of your open transaction.

Example request

- **POST** http://localhost:7474/db/data/transaction
- **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
- **Location:** http://localhost:7474/db/data/transaction/10

```
{
  "commit" : "http://localhost:7474/db/data/transaction/10/commit",
  "results" : [ {
    "columns" : [ "n" ],
    "data" : [ {
      "row" : [ {
        "name" : "My Node"
      } ],
      "meta" : [ {
        "id" : 10,
        "type" : "node",
        "deleted" : false
      } ]
    } ]
  } ],
  "transaction" : {
    "expires" : "Mon, 27 Sep 2021 08:28:00 +0000"
  },
  "errors" : [ ]
}
```

Execute statements in an open transaction

Given that you have an open transaction, you can make a number of requests, each of which executes additional statements, and keep the transaction open by resetting the transaction timeout.

Example request

- **POST** http://localhost:7474/db/data/transaction/12
- **Accept:** application/json;charset=UTF-8
- **Content-Type:** application/json

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

Example response

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

```
{
  "commit" : "http://localhost:7474/db/data/transaction/12/commit",
  "results" : [ {
    "columns" : [ "n" ],
    "data" : [ {
      "row" : [ { } ],
      "meta" : [ {
        "id" : 11,
        "type" : "node",
        "deleted" : false
      } ]
    } ]
  } ],
  "transaction" : {
    "expires" : "Mon, 27 Sep 2021 08:28:00 +0000"
  },
  "errors" : [ ]
}
```

Reset transaction timeout of an open transaction

A transaction expires automatically after a period of inactivity. This can be prevented by resetting the transaction timeout.

The timeout may be reset by sending a keep-alive request to the server, which executes an empty list of statements. The request will reset the transaction timeout and return the new time at which the transaction will expire. The format of the timestamp is: `Day, MM Mon YYYY HH:MI:SS +nnnn`; for example: `Mon, 16 Jul 2018 08:29:31 +0000`.

Example request

- **POST** http://localhost:7474/db/data/transaction/2
- **Accept:** application/json;charset=UTF-8
- **Content-Type:** application/json

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

Example response

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

```
{
  "commit" : "http://localhost:7474/db/data/transaction/2/commit",
  "results" : [ ],
  "transaction" : {
    "expires" : "Mon, 27 Sep 2021 08:28:00 +0000"
  },
  "errors" : [ ]
}
```

Commit an open transaction

Given you have an open transaction, you can send a commit request. Optionally, you can submit additional statements along with the request that will be executed before committing the transaction.

Example request

- **POST** http://localhost:7474/db/data/transaction/6/commit
- **Accept:** application/json;charset=UTF-8
- **Content-Type:** application/json

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

Example response

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

```
{
  "results" : [ {
    "columns" : [ "id(n)" ],
    "data" : [ {
      "row" : [ 5 ],
      "meta" : [ null ]
    } ]
  } ],
  "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:7474/db/data/transaction/3
- **Accept:** application/json;charset=UTF-8

Example response

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

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

Include query statistics

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

Example request

- **POST** http://localhost:7474/db/data/transaction/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

```
{
  "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
    }
  } ],
  "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:7474/db/data/transaction/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

```
{
  "results" : [ {
    "columns" : [ "bike", "p1", "p2" ],
    "data" : [ {
      "row" : [ {
        "weight" : 10
      }, [ {
        "weight" : 10
      }, {
        "position" : 1
      }, {
        "spokes" : 3
      } ], [ {
        "weight" : 10
      }, {
        "position" : 2
      }, {
        "spokes" : 32
      } ] ],
      "meta" : [ {
        "id" : 8,
        "type" : "node",
        "deleted" : false
      }, [ {
        "id" : 8,
        "type" : "node",
        "deleted" : false
      }, {
        "id" : 0,
        "type" : "relationship",
        "deleted" : false
      }, {
        "id" : 9,
        "type" : "node",
        "deleted" : false
      } ], [ {
        "id" : 8,
        "type" : "node",
        "deleted" : false
      }, {
        "id" : 1,
```

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

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.

Example request

- **POST** http://localhost:7474/db/data/transaction/11/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

```
{
  "results" : [ ],
  "errors" : [ {
    "code" : "Neo.ClientError.Statement.SyntaxError",
    "message" : "Invalid input 'T': expected <init> (line 1, column 1 (offset: 0))\n\"This is not a valid
Cypher Statement.\"\n ^"
  } ]
}
```

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:7474/db/data/transaction/9
- **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

```
{
  "commit" : "http://localhost:7474/db/data/transaction/9/commit",
  "results" : [ ],
  "errors" : [ {
    "code" : "Neo.ClientError.Statement.SyntaxError",
    "message" : "Invalid input 'T': expected <init> (line 1, column 1 (offset: 0))\n\"This is not a valid
Cypher Statement.\"\n ^"
  } ]
}
```

Authentication and authorization

The functionality described in this section has been deprecated and will be removed in Neo4j 4.0.

Introduction Deprecated

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

When Neo4j is newly installed, the default user `neo4j` has the default password `neo4j`. The default password must be changed before access to resources will be permitted. See Changing the user password for how to set a new password.

Authenticate to access the server Deprecated

Authenticate by sending a username and a password to Neo4j using HTTP Basic Auth. Requests should include an `Authorization` header with a value of `Basic <payload>`, where `payload` is a base64-encoded string of `username:password`.

Example request

- GET http://localhost:7474/user/neo4j
- **Accept:** application/json;charset=UTF-8
- **Authorization:** Basic bmVvNGo6c2VjcmV0

Example response

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

```
{
  "password_change_required" : false,
  "password_change" : "http://localhost:7474/user/neo4j/password",
  "username" : "neo4j"
}
```

Missing authorization Deprecated

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

Example request

- GET http://localhost:7474/db/data/
- **Accept:** application/json;charset=UTF-8

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 Deprecated

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

Example request

- **POST** http://localhost:7474/db/data/
- **Accept:** application/json;charset=UTF-8
- **Authorization:** Basic bmVvNGo6aW5jb3JyZWN0

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."
  } ]
}
```

Required password changes Deprecated

In some cases, for example the very first time Neo4j is accessed, the user will be required to choose a new password. The database will signal that a new password is required and deny access.

See Changing the user password for how to set a new password.

Example request

- **GET** http://localhost:7474/db/data/

- **Accept:** application/json;charset=UTF-8
- **Authorization:** Basic bmVvNGo6bmVvNGo=

Example response

- **403:** Forbidden
- **Content-Type:** application/json;charset=utf-8

```
{
  "password_change" : "http://localhost:7474/user/neo4j/password",
  "errors" : [ {
    "code" : "Neo.ClientError.Security.Forbidden",
    "message" : "User is required to change their password."
  } ]
}
```

User status on first access Deprecated

On first access, and using the default password, the user status will indicate that the users password requires changing.

Example request

- GET ttp://localhost:7474/user/neo4j
- **Accept:** application/json;charset=UTF-8
- **Authorization:** Basic bmVvNGo6bmVvNGo=

Example response

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

```
{
  "password_change_required" : true,
  "password_change" : "http://localhost:7474/user/neo4j/password",
  "username" : "neo4j"
}
```

User status Deprecated

Given that you know the current password, you can ask the server for the user status.

Example request

- GET http://localhost:7474/user/neo4j
- **Accept:** application/json;charset=UTF-8
- **Authorization:** Basic bmVvNGo6c2VjcmV0

Example response

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

```
{
  "password_change_required" : false,
  "password_change" : "http://localhost:7474/user/neo4j/password",
  "username" : "neo4j"
}
```

Changing the user password Deprecated

Given that you know the current password for a user, you can ask the server to change that user's password. You can choose any password as long as it is different from the current password.

Example request

- **POST** http://localhost:7474/user/neo4j/password
- **Accept:** application/json;charset=UTF-8
- **Authorization:** Basic bmVvNGo6bmVvNGo=
- **Content-Type:** application/json;charset=UTF-8

```
{
  "password" : "secret"
}
```

Example response

- **200:** OK

License