Application Layer GraphQL-API Documentation

Ceyoniq

Version 10.1.1508, 09.07.2026

Table of Contents

Overview . . . 1
1. Introduction . . . 2
2. GraphQL Endpoint, Header and Parameter . . . 3
2.1. Endpoint URL . . . 3
2.2. Common Request Header and Query Parameter . . . 4
2.3. Response Header . . . 4
3. GraphQL Tools . . . 5
3.1. GraphQL-Playground . . . 5
3.2. GraphQL-Voyager . . . 5
4. GraphQL Examples . . . 7
4.1. With cURL . . . 7
Programming . . . 8
5. GraphQL Client . . . 9
5.1. Authentication Service . . . 9
5.1.1. Login . . . 9
5.1.2. Logout . . . 10
5.2. Configuration Service . . . 10
5.2.1. retrieve Common Settings . . . 10
5.2.2. update Common Settings . . . 11
5.2.3. retrieve Document Area settings . . . 12
5.2.4. create a Document Area . . . 12
5.2.5. update a Document Area . . . 13
5.2.6. delete a Document Area . . . 14
5.3. User Management Service . . . 15
5.3.1. all users of a domain . . . 15
5.3.2. create new user . . . 16
5.4. Repository Service . . . 16
5.4.1. root folder of a Document Area . . . 16
5.4.2. children of a folder . . . 17
5.4.3. search for elements . . . 18
5.4.4. modify properties . . . 18
5.5. Authority Management Service . . . 19
5.5.1. all permissions of a role . . . 19
5.5.2. update permissions . . . 20
5.5.3. remove permissions . . . 21
5.6. Uploading Binary Files . . . 22
5.6.1. File Upload with ApplicationLayer . . . 23
5.7. Downloading Files . . . 25

Overview

This document describes the GraphQL-API provided by the Application Layer. The GraphQL API is for developers who want to integrate the Application Layer into their application and for administrators who want to script interactions with the Application Layer Server.

The Application Layer's GraphQL API provides access to **almost all** services and resources of the Application Layer via queries/mutations. To use the GraphQL API, your application will make a POST HTTP request - which contains the GraphQL's queries/mutations - to the GraphQL's endpoint `http(s)://hostname:port/nscalealinst1/graphql`, and parse the response. The response format is JSON. Your queries/mutations will have the same names as the names of the service's methods in the Application Layer. Mutation's names have as prefix the name of their root service.

GraphQL is schema based, and has a strong type system, Because of that, it is possible to introspect the schema. With the schema introspection, GraphQL has many advantages. With schema introspection you can use a various number of tools for the development, such as tools for testing/introspection and for schema visualization. Application Layer comes with two such tools, namely `GraphQL-Playground` and `GraphQL-Voyager`. More details about these tools will be mentioned later in this document.

Chapter 1. Introduction

Accessing GraphQL API is performed via HTTP. GraphQL API uses only one HTTP method: POST request method. Other than REST API, GraphQL has only one endpoint, which is: `http(s)://hostname:port/nscalealinst1/graphql`

Clients can request data or performs internal server operations/changes by sending queries or mutations to the GraphQL endpoint through POST requests. In GraphQL there are three operations: `query`, `mutation`, `subscription`. The Application Layer GraphQL API supports only two of these operation: query and mutation.

With query data can be requested/retrieved, and with mutation changes/modifications can be made to the server/data.

Say that we want to request some information about the common setting from the configuration service in the Application Layer, so our GraphQL query will look like the following:

```
query {
  configurationService {
    commonSetting {
      deletionPolicy
      lastModified
      idleSessionLifeTime
    }
  }
}
```

This GraphQL query should be sent within a POST request in the body of the request. A GraphQL POST request of this query should look like the following:

```
POST /nscalealinst1/graphql HTTP/1.1
Host: hostname:port
Content-Type: application/json;charset=UTF-8
Authorization: Basic YWRtaW46YWRtaW4=
Accept: application/json;charset=UTF-8

{"operationName":null,"variables":{},"query":"{\n  configurationService {\n
commonSetting {\n      deletionPolicy\n      lastModified\n      idleSessionLifeTime\n
}\n  }\n}\n"}
```

It's important to include the authorization information in the headers of request. The authorization information consist of the nscale user name and password, decoded to Base64 (aka: Basic HTTP authentication).

The Application-Layer GraphQL-API supports only the JSON representation format.



Changing/Uploading binary data is possible since nscale Applicationlayer **8.3**.

Chapter 2. GraphQL Endpoint, Header and Parameter

2.1. Endpoint URL

The main and only endpoint URL for a Application-Layer GraphQL API has the following structure:

`http(s)://hostname:port/nscalealinst1/graphql`

The Application Layer only uses a self-signed certificate by default. You can replace this server certificate with a digitally signed certificate by a certificate authority (CA).

The self-signed certificate can be downloaded at `http(s)://hostname:port/server.certificate`

Table 1. Port

Number	Description
8080	plain HTTP port
8443	secure HTTPS port

The logical application layer instance is part of the URL. Your application should make this entry configurable.

Table 2. Instance

Name	Description
nscalealinst1	The default application-layer instance name (can be changed)
[any-name]	Secondary logical instance name

In GraphQL API there are several Application Layer services available. Here are the available services in GraphQL:

Table 3. Core Services

Name	Description
administrative	administrative functionality: administrative access on resources, e.g. to copy or move resources from one document area to another
authentication	authentication functionality: session and authentication infos
authoritymanagement	authority management functionality: roles
collaboration	collaboration functionality: groups, teamspaces and calendars
configuration	configuration functionality: dictionary, layouts, property definitions, value sets
masterdata	masterdata functionality: external data
monitoring	monitoring functionality: monitoring data, invoke generic

Name	Description
messaging	messaging functionality: subscribe resources and workflow, read messages
repository	repository functionality: folder, link and document management
usermanagement	user management functionality: principals, org. entities, groups and users.
workflow	workflow functionality: processes and taks

The document `graphql_schema.graphql` contains the full GraphQL schema.

2.2. Common Request Header and Query Parameter

The following table describes headers that can be used by GraphQL requests.

Table 4. Request Header

Header	Description
Authorization	The information required for request authentication
Accept	Media type that is acceptable for the response (content negotiation).
Content-Type	Media type of the body of the request.

Table 5. Query Parameter

Header	Description
appid	The client application id of the client. If no appid is given, the client requires the 'nscale SDK' license.
autoclose=true	The application cannot use the session cookie. Close session after request.
clientversion	The client application version.

2.3. Response Header

The following table describes response headers that are common to GraphQL responses.

Table 6. Response Header

Header	Description
Content-Length	Length of the message (without the headers) according to RFC 2616
Content-Type	The content type of the resource in case the request content in the body

Chapter 3. GraphQL Tools

3.1. GraphQL-Playground

GraphQL-Playground is a graphical, interactive, in-browser GraphQL IDE that helps you during the development. With GraphQL-Playground you can send queries/mutations and get responses ***live*** from the Application Layer server.

GraphQL-Playground supports ***auto-completion***, which means that you will be able to explore the schema of GraphQL API and with that you will be able to explore the services of the Application Layer, and their methods in such a way that it will be almost self-explaining and self-documenting. It will make it super easy to understand how to use the methods of the services in the Application Layer and which result to expect.

GraphQL-Playground also supports ***error highlighting***, so that you will always be sure that you are writing the correct working queries/mutations - for your Applications - that deliver the wished and expected results every time!

cURL input can be generated with GraphQL-Playground so that your queries in Playground can be directly executed with cURL. All what you need to do is to click on `COPY CURL` on Playground tool, and the query/mutation on Playground will be included on a generated cURL input that will be saved on the system clipboard.

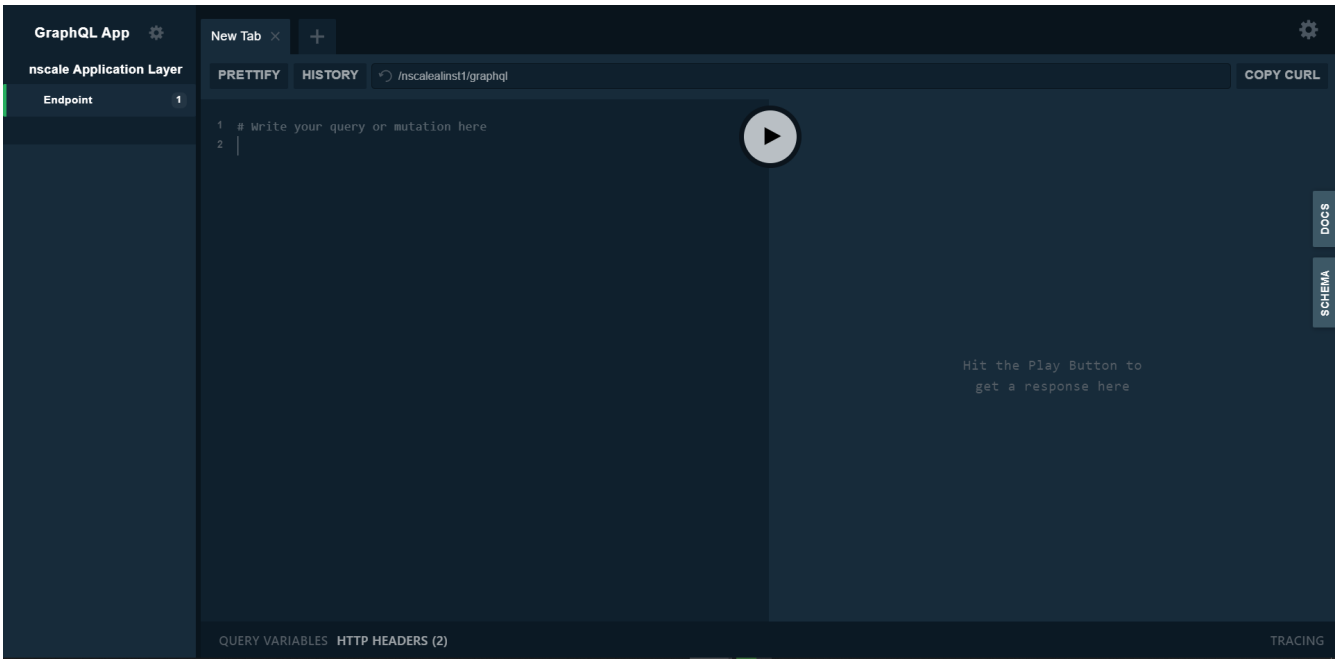


Figure 1. GraphQL-Playground IDE

3.2. GraphQL-Voyager

GraphQL-Voyager is used to ***visualize*** the GraphQL schema in a way that is similar to UML Diagrams. This helps you even more understanding how the schema, and the underlying structure of the Application Layer is built.

GraphQL-Voyager let you filter Application Layer data types in such a manner that you can get a

specific parts of the whole graph and thus reduce the complexity of the graph.

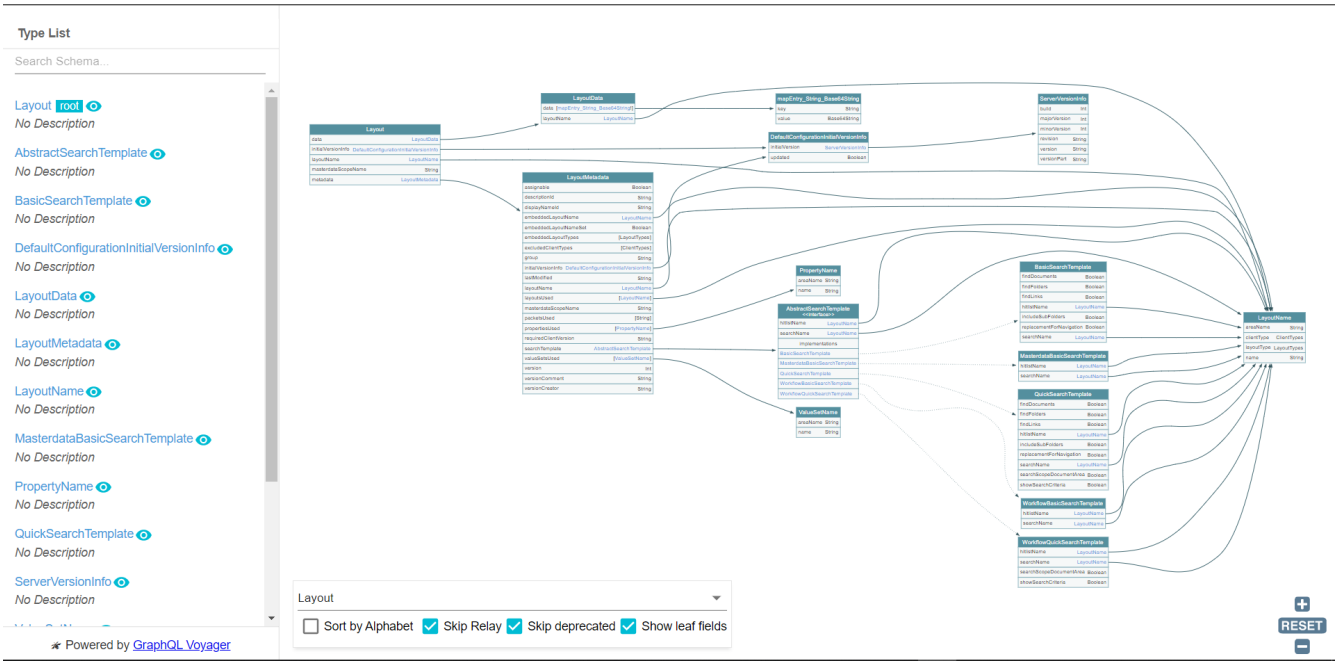


Figure 2. GraphQL-Voyager - filtered the graph of the data type `Layout`

Chapter 4. GraphQL Examples

4.1. With cURL

First try to access the application layer via the cURL command line tool.

Here is an example session:

```
curl -v --basic -u admin:admin
     -H "Content-Type:application/json"
     -d "{\"query\":\"query {\n  configurationService {\n  commonSetting {\n
         deletionPolicy\n      lastModified\n      idleSessionLifeTime\n
         }\n  }\n}\"}"
     http://localhost:8080/nscalealinst1/graphql
```

```
POST /nscalealinst1/graphql HTTP/1.1
Host: localhost:8080
Authorization: Basic YWRtaW46YWRtaW4=
User-Agent: curl/7.55.1
Accept: */*
Content-Type:application/json
Content-Length: 150
```

Answer from Application Layer:

```
HTTP/1.1 200
Set-Cookie: JSESSIONID=BE70947D157C8B6E9A7581507A15DBAF; Path=/nscalealinst1; HttpOnly
Content-Type: application/json;charset=UTF-8
Content-Length: 159
Date: Mon, 15 Feb 2021 17:13:13 GMT

{"data":{"configurationService":{"commonSetting":{"deletionPolicy":"DeletePhysical","l
astModified":"2021-02-08T15:28:33.942+01:00","idleSessionLifeTime":60}}}}
```

Programming

This chapter describes details for programming with Application Layer GraphQL API.

If you wish to watch the queries and mutations in the following chapter being written and executed live, read the html version of this document instead.

Chapter 5. GraphQL Client

Let's write GraphQL queries and mutations for a typical client for the GraphQL-API. The queries and mutations will be written with GraphQL-Playground. Note that the user permissions will be considered during the query/mutation executions.

5.1. Authentication Service

5.1.1. Login

The standard HTTP authentication mechanism is used by GraphQL:

- Basic
- NTLM
- Negotiate (Kerberos or NTMLv2)
- OpenID Connect (ADFS)

Additional authentication schemas:

- Implicit (can be used for impersonation)
- AuthID (ID/secure-card based login)
- KNM (Kyocera Network Manager)
- SAML (planned)

The server uses a session cookie. The client should use this cookie for following requests.



GraphQL-Playground auto completion

It will be so easy to write the query with the auto-completion features that comes with GraphQL-Playground. Auto-completion is performed by clicking the following key combination: `(ctrl + space)`.

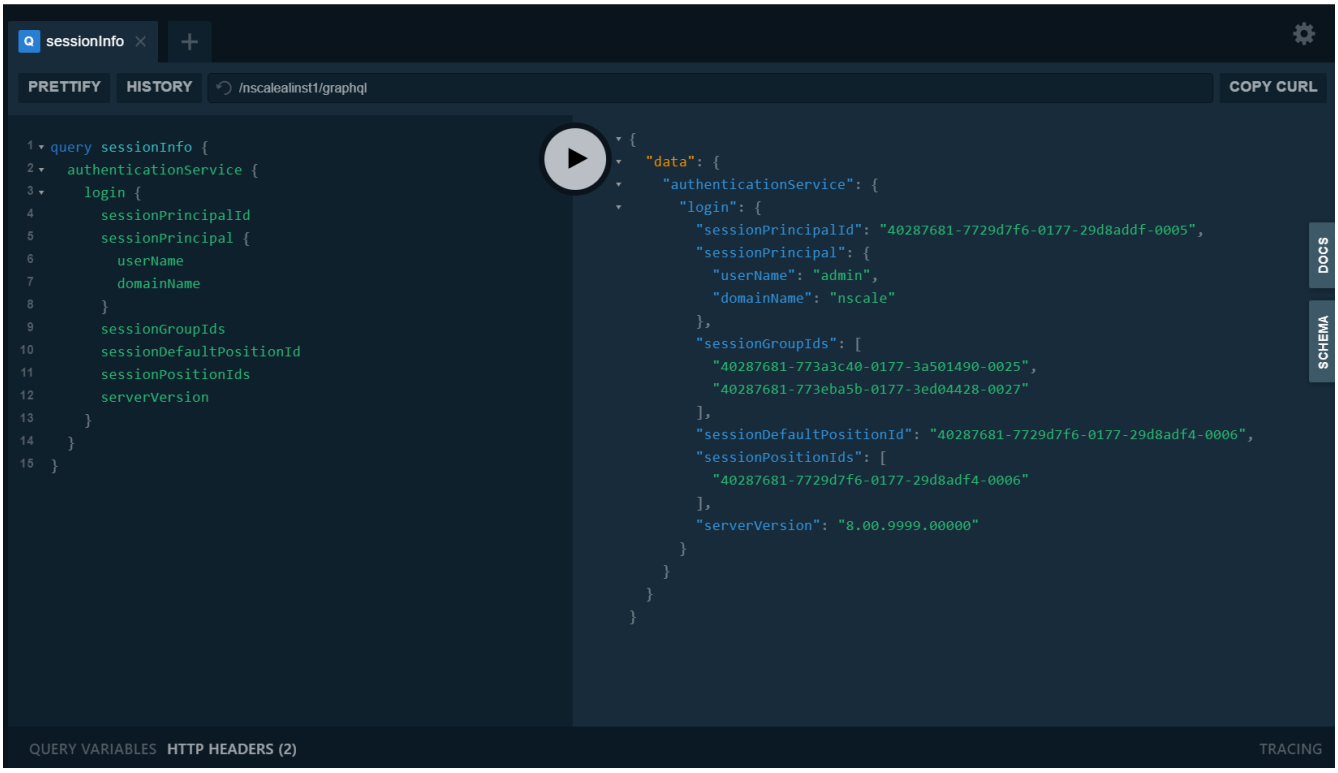


Figure 3. GraphQL query for getting some informations about the user current session

The above GraphQL query retrieves additional information of the session access rights (more fields can be selected!).

5.1.2. Logout

```
mutation sessionLogout {
    AuthenticationService_logout
}
```

This mutation will close the server session and the HTTP session for the GraphQL adapter.



Warning

just disconnecting the HTTP connection will leave an open session in the server.

5.2. Configuration Service

5.2.1. retrieve Common Settings

Let's say we want to get some common setting informations from Application Layer server. For some reasons we want only the following informations: `deletion policy`, `idle session lifetime (in minutes)` and `jdbc batch size`. We would write the following query with GraphQL-Playground (see figure 4).

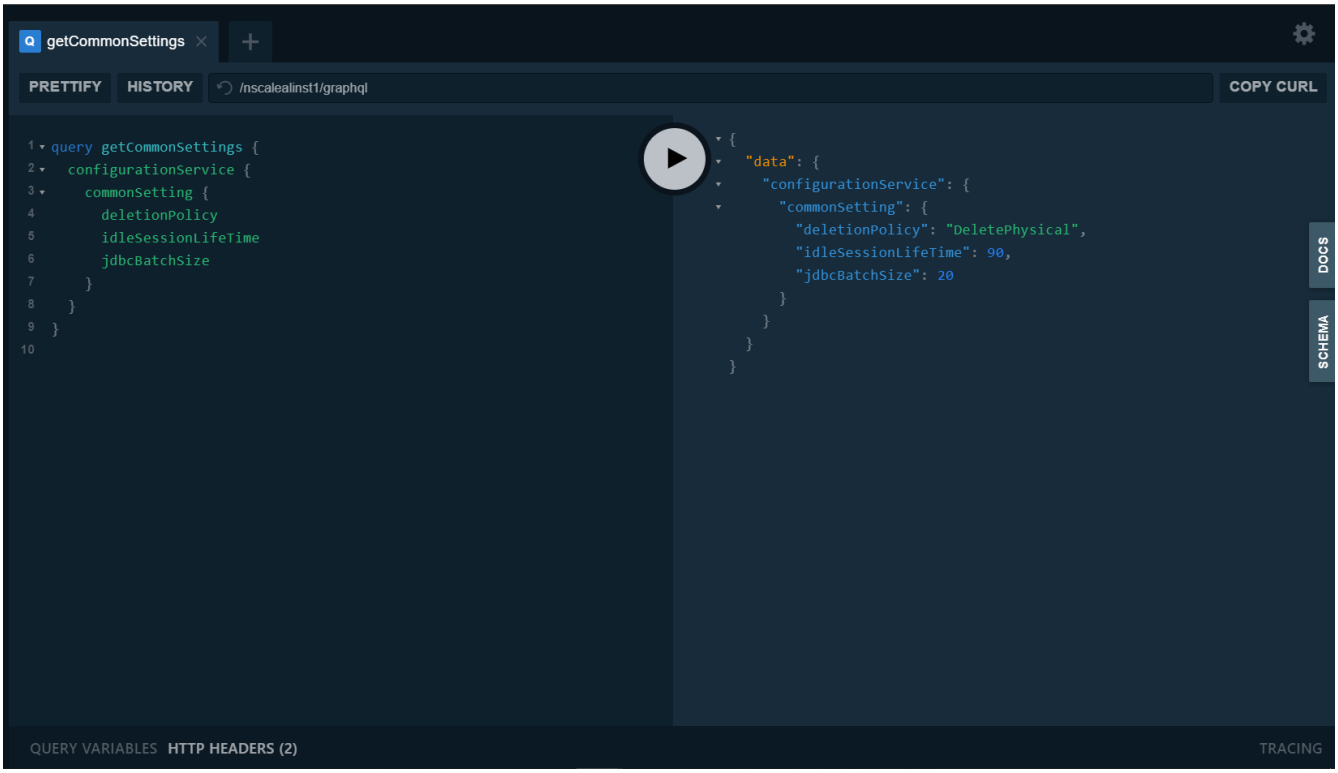


Figure 4. GraphQL query to request specific common settings

5.2.2. update Common Settings

To make a change to data or settings in the Application Layer, we can write a GraphQL-Mutation to perform the wished changes. For instance, we can change the following common settings: `idleSessionLifeTime`, `deletionPolicy`, `createPreviews` with the following mutation:

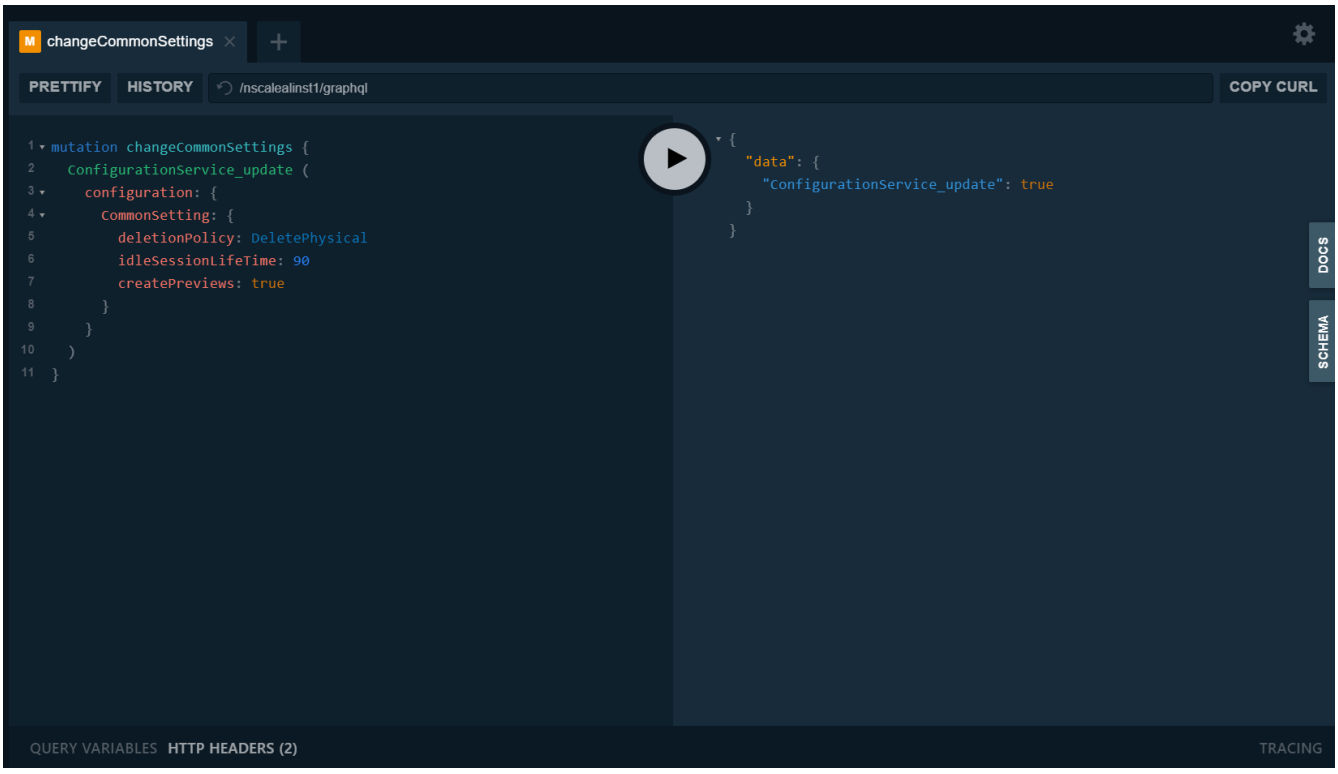


Figure 5. GraphQL mutation to change specific common settings



graphql input & input fields

> to fill graphql input, you should write `:` after the input name, and if the input is of a complex type, every input field should be filled in the same way (see the above figure)

Other configuration objects can be updated the same way as `CommonSetting`. Note that CommonSetting is a singletone configuration object, that means it has no identifier and there is only one such configuration object that exists in the whole Application Layer. Other configuration objects may have identifiers, so be aware!

5.2.3. retrieve Document Area settings

In the case that we want to get specific informations about the document area `DA`, like `defaultContentType`, `expirationRulesDateInheritancePolicy`,`lastModified` and `appendNotePolicy` we would write the following GraphQL query:

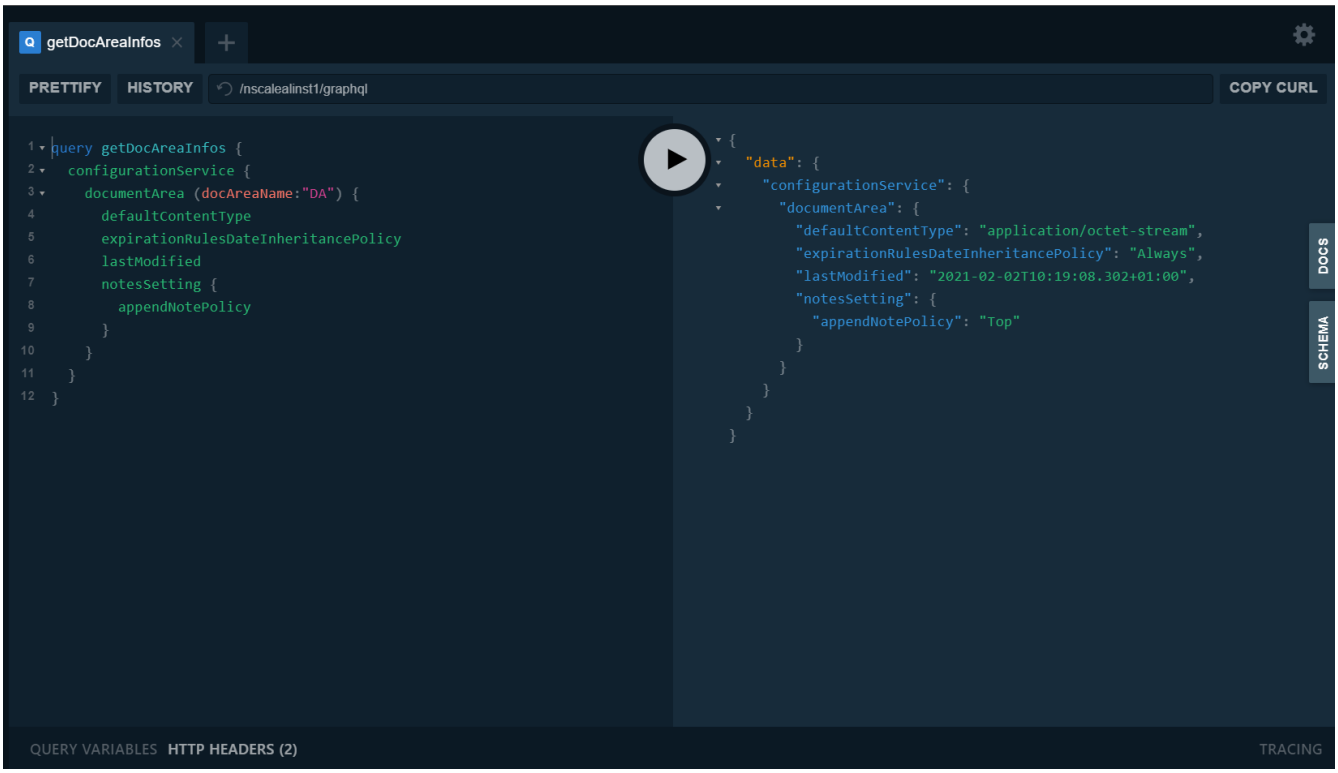


Figure 6. GraphQL query to get document area settings



selecting nested fields

note that `appendNotePolicy` is not a direct field of the type `DocumentArea`, it's a nested field of type `NotesSetting` and can be selected just like the normal fields inside `DocumentArea`.

If we want to get all Document Areas, we would write a query that use the Configuration Service operation `documentAreas` and we could select any wished fields (just like the query above).

5.2.4. create a Document Area

To create a new document area with some initial configuration, you will need to write a mutation that looks like the following:

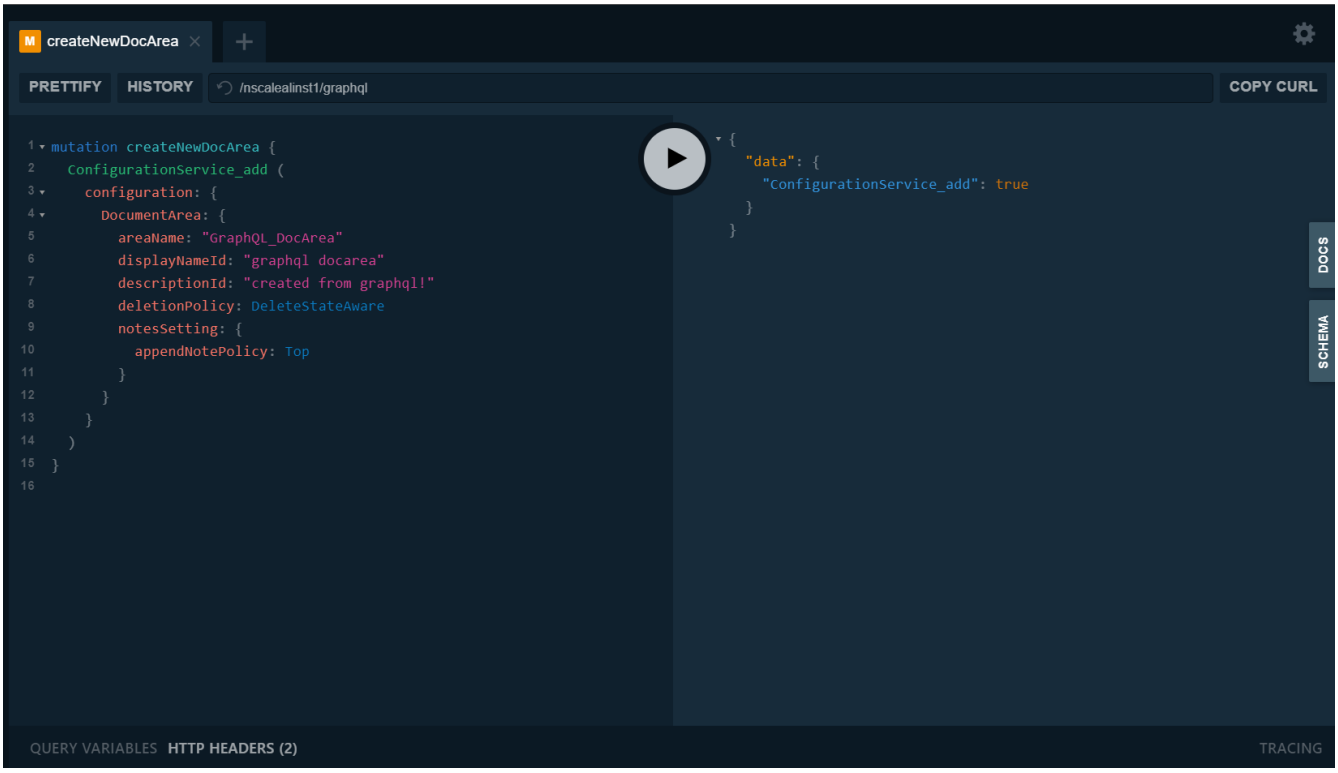


Figure 7. GraphQL mutation for creating new document area



identifiers of the configuration objects

note that some configuration objects have identifiers, and cannot be access or created without these identifiers. The `DocumentArea` is a configuration object that has `areaName` as identifier, and will be required when creating, getting, updating or deleting a document area.

5.2.5. update a Document Area

To update a Document Area we would write a mutation that use the operation `ConfigurationService_update`, like the following:

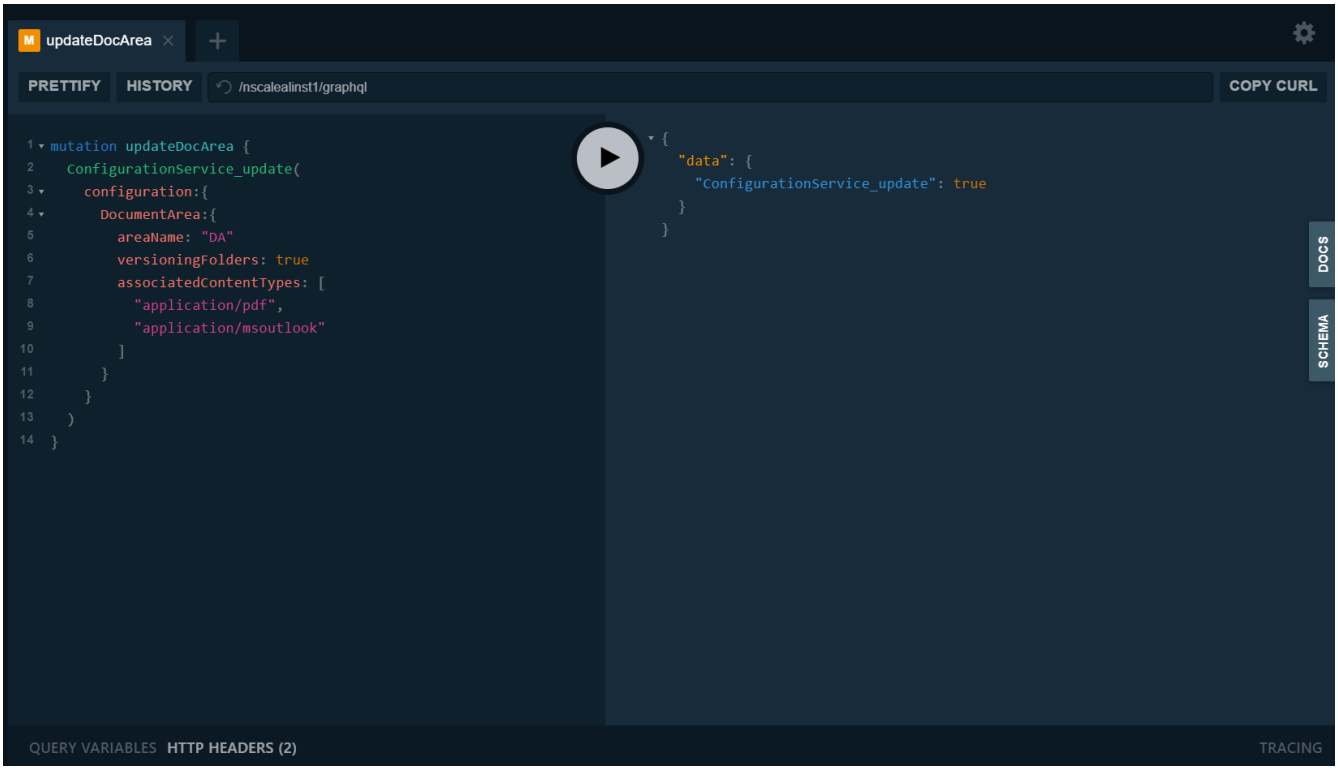


Figure 8. GraphQL mutation for updating a Document Area



updating lists and arrays inside a configuration object

when updating lists, sets or arrays inside a configuration object, the list **won't be merged** with the new values! The updated list will have only the new values, so be very careful when updating big lists! When you have big lists, you would programmatically save them in variables and then add the new elements to them and then make the update. To update a list and make it empty, you can do this by writing `[]`.

5.2.6. delete a Document Area

To delete a document area, you will need only one field to write, namely the identifier field `areaName`:

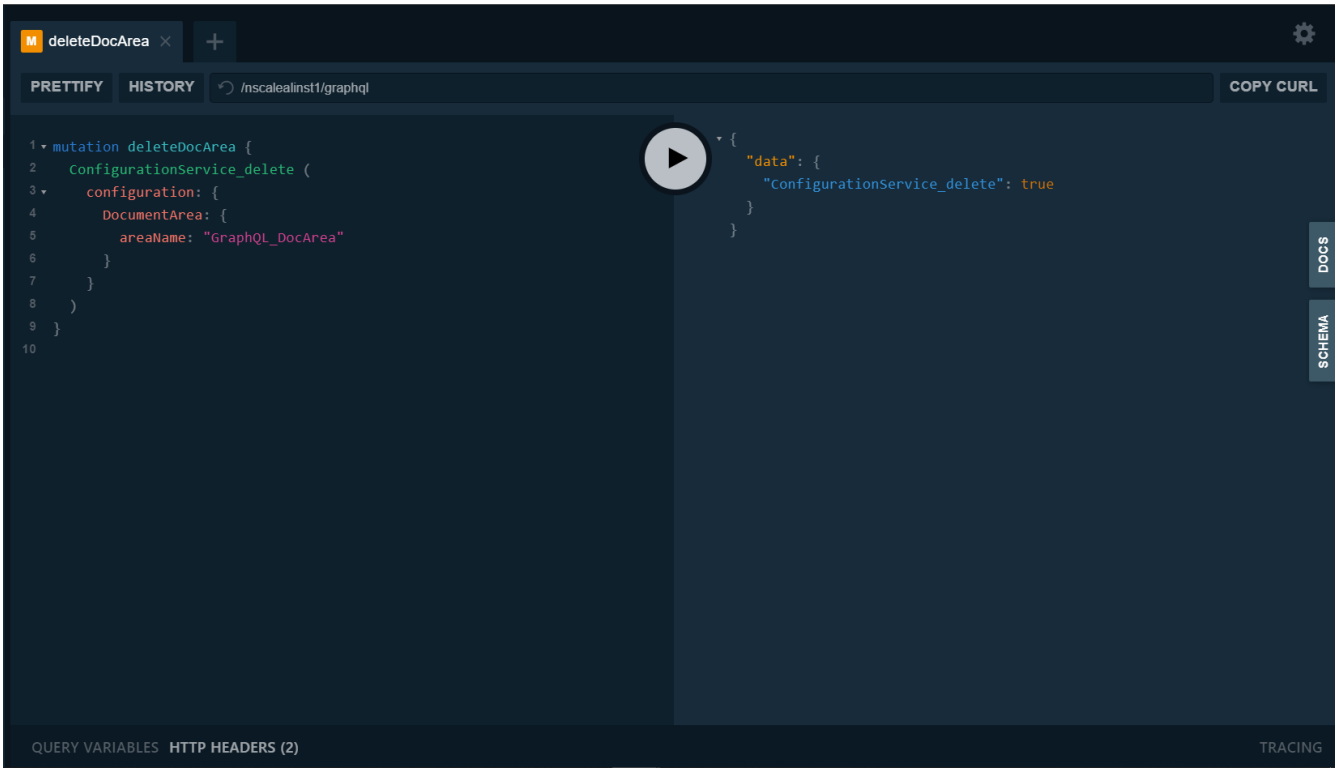


Figure 9. GraphQL mutation for deleting a document area

5.3. User Management Service

5.3.1. all users of a domain

To show all users of the default nscale domain, and some of their informations, like `login` name and `principalId`, you would write a query that looks like the following:

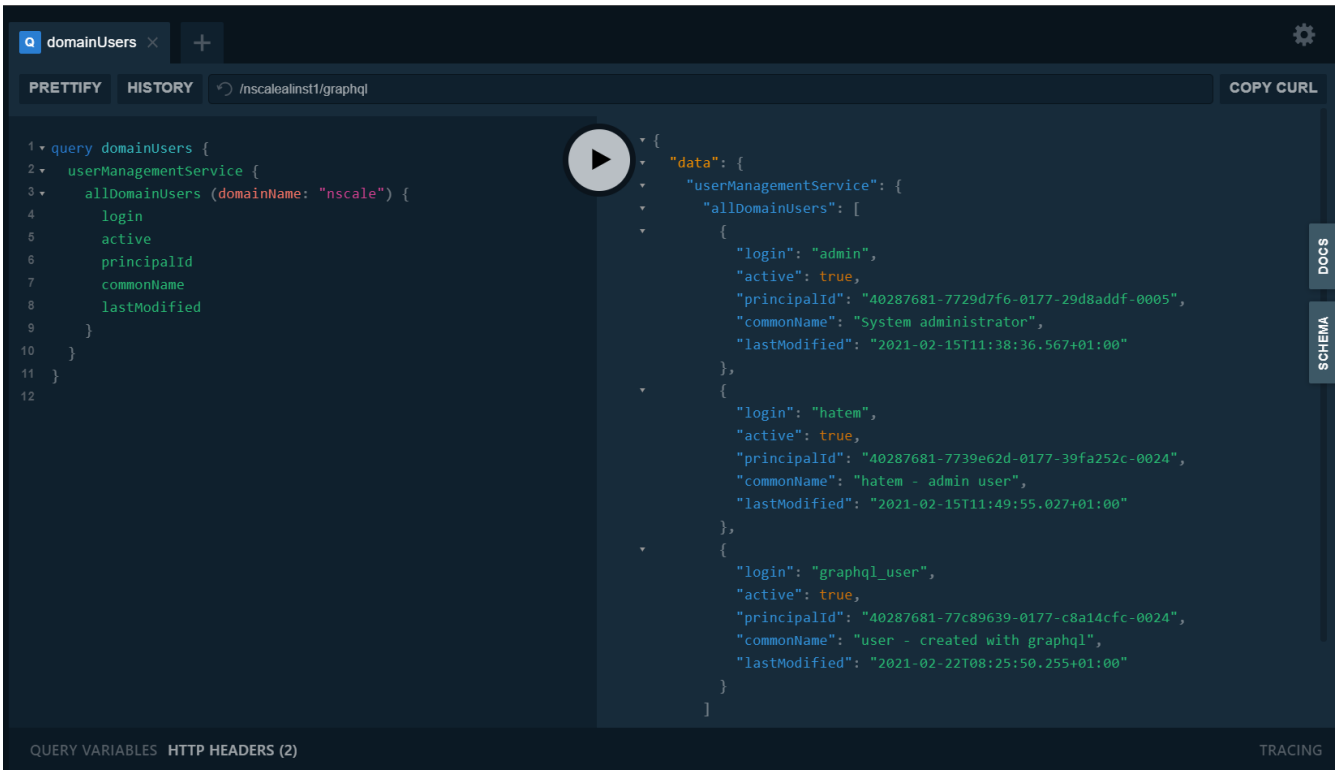


Figure 10. GraphQL query for getting all domain users



graphql alias

you can rename client wise any field by using **graphql-alias**. To use them, write any chosen name and then add `:` before the actual field name, like this: `aliasName:actualFieldName`. In the above example we could rename the field `login` to `username`, just like that: `username:login`. You can use graphql-alias even with whole queries or operations and as a result you can execute one operation many times in one query in one request!

5.3.2. create new user

For creating a new user, we write a mutation that use the user management service's operation `createUser(user:UserInput, password:String)`. Note that a user has as identifier `(name, domainName)` or `(name, domainName, principalId)`, so make sure to write one of them when creating a user.

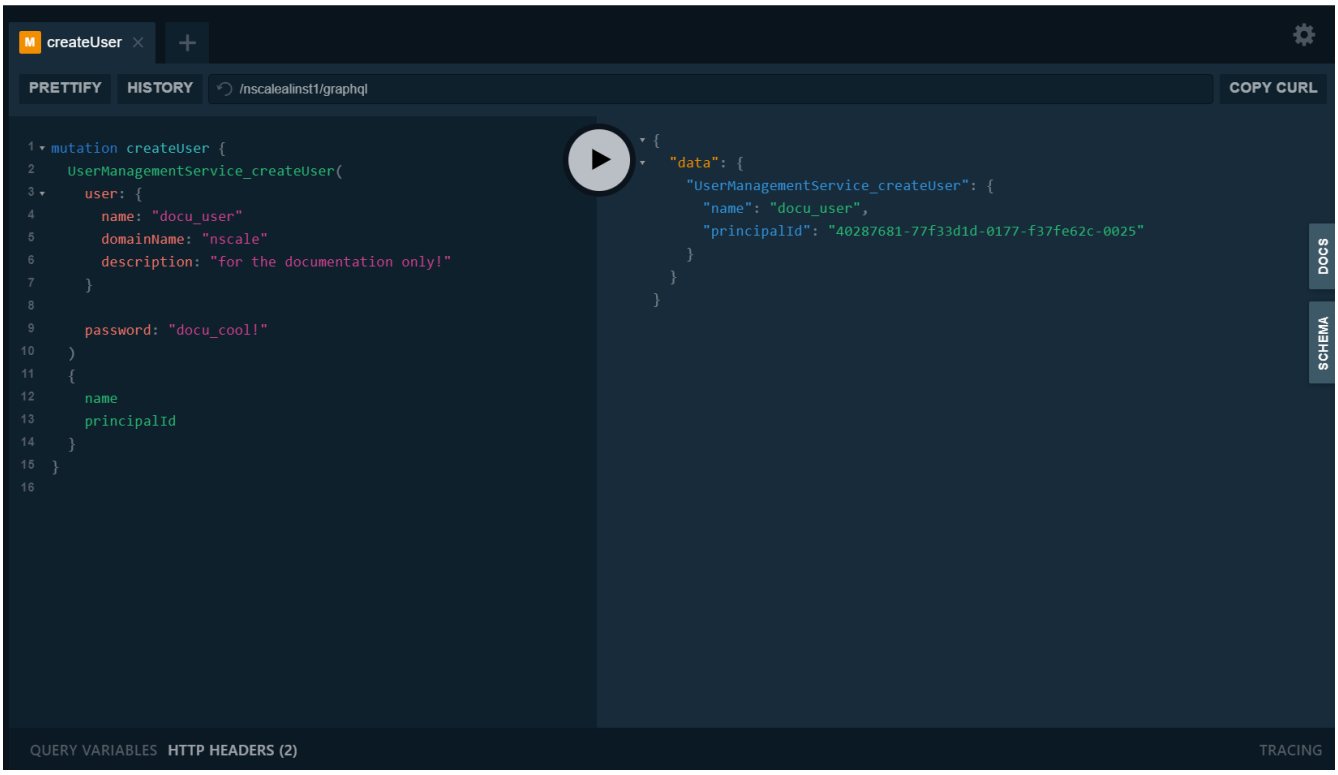


Figure 11. GraphQL mutation for creating a new nscale user



mutation return value

not all mutations are of type void, some mutation return objects as a result of the execution and this can be useful in many cases, such as getting the generated `principalId` after creating a new user.

5.4. Repository Service

5.4.1. root folder of a Document Area

To start on a document area you must retrieve the root entry point (resource key).

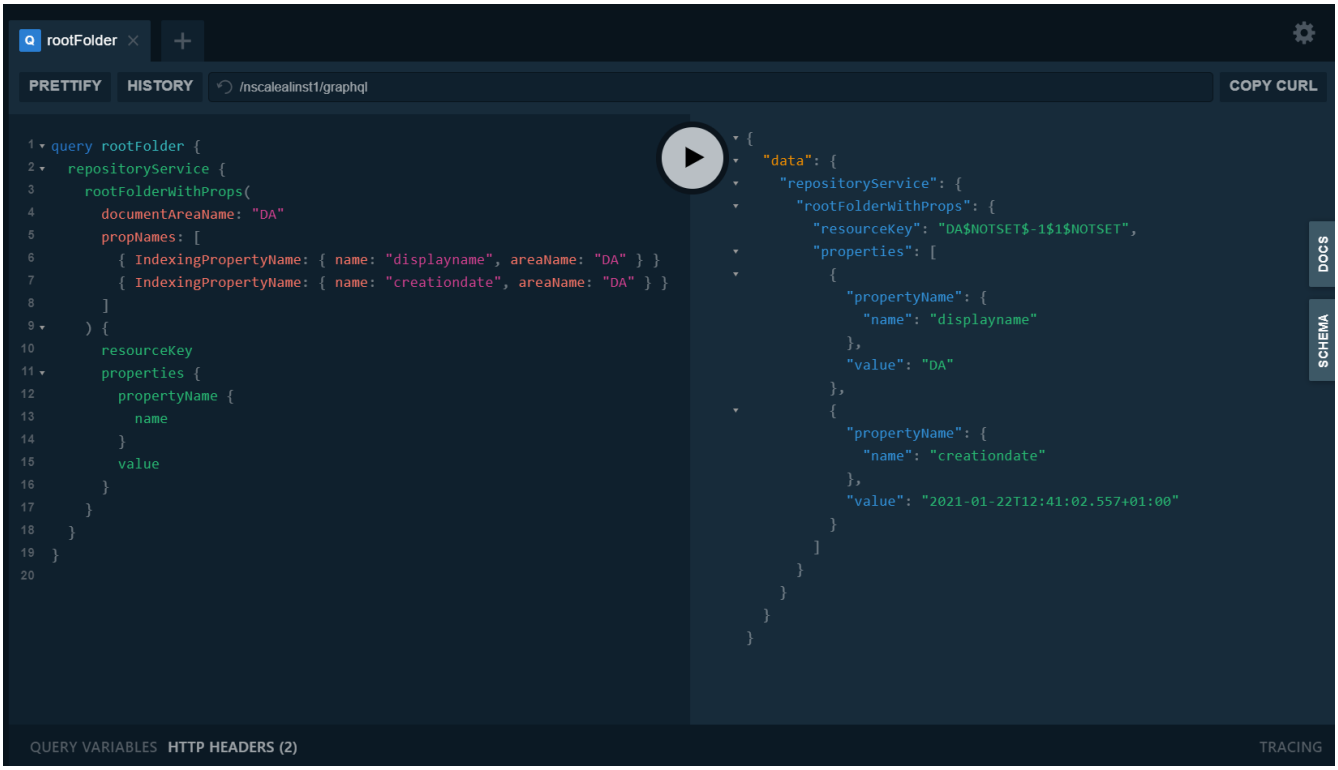


Figure 12. GraphQL query that retrieve the root folder of a document area

5.4.2. children of a folder

To display the child elements of a folder, you must retrieve the possible children. With `SearchControl` you can specify, which properties should be returned and how they should be ordered and which condition should be used. With `filter`, you can even write a condition, in this example the condition is to return only the files that has the extension of a MS-Word documents. Note that GraphQL-alias were used in the result. Note that there is a comment inside the field `sortOrder`. Comments can be written inside GraphQL-Queries or Mutations, and they begin with the character `#`.

```
query folderChildren {
  repositoryService {
    children(
      folderKey: "DA$NOTSET$2$1$NOTSET"
      searchControl: {
        propNames: [
          { IndexingPropertyName: { name: "displayname", areaName: "DA" } }
          { IndexingPropertyName: { name: "objectclass", areaName: "DA" } }
          { IndexingPropertyName: { name: "creationdate", areaName: "DA" } }
        ]
        paging: { pageNumber: 1, pageSize: 20 }
        searchScope: OneLevel
        sortOrder: {
          # result is sorted by 'displayname', ascending
          propertyName: {
            IndexingPropertyName: { name: "displayname", areaName: "DA" }
          }
          sortOrder: ASC
```

```
        }
        filter: "displayname like \"*.doc*\""
      }
    ) {
      resultTable {
        resourceKeys
        firstElement: row(rowIndex: 0)
        displayNames: column(columnIndex: 0)
        objectClasses: column(columnIndex: 1)
        creationDates: column(columnIndex: 2)
      }
    }
  }
}
```

The result is a table with resource keys and their properties. Try to execute the above query to find out how the result table is structured. Try to use the auto-completion functionality when writing your query.

5.4.3. search for elements

Searching in folders is analog to childern. The only difference is that searching method has many variants, such as with `fulltext query`, or with `nql query` instead of `SearchControl`.

5.4.4. modify properties

To modify element properties, we write a mutation that does several tasks:

- locking the resource
- modifying resource's properties
- unlocking the resource

So if you want to modify the property value of `displayname`, you would write the following mutation:

```
mutation modifyProperties {
  RepositoryService_lock(resourceKey: "DA$NOTSET$309$2$NOTSET")

  RepositoryService_updateProperties(
    resourceKey: "DA$NOTSET$309$2$NOTSET"
    props: [
      {
        propertyName: {
          IndexingPropertyName: { name: "displayname", areaName: "DA" }
        }
        value: "new_docu"
      }
    ]
  )
```

```
  RepositoryService_unlock(resourceKey:"DA$NOTSET$309$2$NOTSET")
}
```

If everything worked properly, void operations should return `true`, and the response would be something like this:

response from Application Layer

```
{
  "data": {
    "RepositoryService_lock": true,
    "RepositoryService_updateProperties": true,
    "RepositoryService_unlock": true
  }
}
```



mutation execution

Operations inside a graphql-mutation are executed in a serial way. That guarantied that the operation are executed in the wished order and that makes a lot of sense in most of the cases, like the above example. In the other hand, operations inside queries are not executed serially.

5.5. Authority Management Service

5.5.1. all permissions of a role

To get all permissions of a specific role, the role name and area name are required. With the graphql operation `getPermissions` a list `[Permissions]` of all types of permissions is returned. Since the type `Permissions` is a graphql **interface**, a special graphql syntax is required to select the result. The interface **Permissions** has the following subtypes/implementations:

- SystemAdminPermissions
- UserAdminPermissions
- DocAreaAdminPermissions
- DocumentPermissions
- FolderPermissions
- LinkPermissions
- MasterdataPermissions
- WorkflowPermissions
- BusinessProcessPermissions

So the graphql query would look like the following:

```
query getRolePermissions {
```

```
  authorityManagementService {
    allPermissions(roleNames: [{ areaName: "*", name: "SystemAdmin" }]) {
      ...on DocAreaAdminPermissions {
        __typename
        docAreaAdmin
        docAreaName
        layoutAdmin
        monitoringAdmin
        physicallyDelete
      }

      ...on UserAdminPermissions {
        __typename
        auditAdmin
        domainAdmin
        domainName
      }

      ...on SystemAdminPermissions {
        __typename
        configurationAdmin
        monitoringAdmin
        revisionAdmin
      }

      # ...on MasterdataPermissions {...}
    }
  }
}
```



syntax for interfaces

when retrieving data from an operation that has an interface type as return type, selecting the data should be done with the special graphql syntax `⋯on <Subtype>{⋯}`. With this syntax you can select subtype specific fields that cannot be selected otherwise.

From the above query we can see that the area name is not specific, and is a star *. This indicates that the role was a global role and applies to all document areas in the ApplicationLayer instance.

5.5.2. update permissions

To update permissions of a role, use the mutation `AuthorityManagementService_updatePermissions`. It should be noted, that some permissions have identifiers, so these identifiers should be always given, whenever we want to update or remove a permission.

The following query illustrates how updating permissions works:

```
mutation updateRolePermissions {
  AuthorityManagementService_updatePermissions (
```

```
    roleName: { areaName:"*", name:"SystemAdmin" }
    permissions:[
      {
        SystemAdminPermissions: {
          monitoringAdmin: false
          revisionAdmin: true
        }
      }

      {
        DocAreaAdminPermissions: {
          docAreaName: "DA"
          layoutAdmin: false
          physicallyDelete: false
        }
      }
    ]
  )
}
```

The mutation `updatePermissions` can be also used to add new permission to a given role. Use the mutation `setPermissions` to initialize permissions for a role.

5.5.3. remove permissions

Permissions can be also removed for a given role. The identifiers of the permissions are the only required information when removing a permission for a given role. A graphql query for removing permissions should look like the following:

```
mutation removeRolePermissions {
  AuthorityManagementService_removePermissions(
    roleName: { areaName: "*", name: "SystemAdmin" }
    permissions: [
      {
        DocAreaAdminPermissions: {
          docAreaName: "DA"
        }
      }
    ]
  )
}
```



new operations

the mutations `updatePermissions` and `removePermissions` are only available since the ApplicationLayer version `8.0.5200`.

5.6. Uploading Binary Files

Since graphql supports only JSON representation format, it is difficult and inefficient to upload binary files within JSON format. In this case the binary file payload would be encoded into `base64` string and will be then written in the JSON request. The encoded `base64` strings are usually very large, and are in most cases larger than the original binary file content. It is an expensive and inefficient way to upload binary files with this method, since you need to encode the content of the binary file client wise to `base64` format, and when sending the request, the request will be very large and the `base64` encoded string should be decoded server wise. This is clearly not an optimal solution for uploading binary files with graphql.

A better and efficient solution is to create a **multipart/form request** and send the binary file within it. according to the GraphQL multipart request specification there should be three form-fields, which are ordered like the following:

- *operations*
- *map*
- File field

The form-field *operation* should contain a JSON encoded object that represents the normal graphql POST request body:

```
{
  "query": "...",
  "variables": { "var_1": "value_1", ... }
}
```

Every variable value of a file in the *operations* form-field should be **replaced with** `null` value. The form-field *map* should be a JSON encoded map that describes where the files in *operations* are. An entry of this map is corresponding to a file, where the key is the file multipart form-field name and the value is an array of operations paths. The final form-field is the binary file field, where the actual binary file name is mapped to the file key from the above-mentioned map. There could be one or many file fields, it depends on the operations that should be executed.

The following example shows how to upload a single binary file with graphql using cURL:

upload a single file

```
curl -v --basic -u admin:admin http://localhost:8080/nscalealinst1/graphql ^
     -F operations="{\"query\":\"mutation upload($file: Upload, $folder: String) {
                                uploadFile(content: $file, folder: $folder )}\",
                    \"variables\":{\"file\":null,\"folder\":\"folderId\"}}" ^
     -F map="{\"0\":[\"variables.file\"]}" ^
     -F 0=@small_file.pdf
```



note how the file variable value was set to `null` inside the *operations* form-field.

Multiple files could be also uploaded with this method. The following example shows how to upload multiple files with graphql using cURL:

upload multiple files

```
curl -v --basic -u admin:admin http://localhost:8080/nscalealinst1/graphql ^
     -F operations="{\"query\":\"mutation upload($files: [Upload], $folder: String) {
                                uploadFiles(contentList: $files, folder:$folder )}\",
                    \"variables\":{\"files\":[null,null],\"folder\":\"folderId\"}}" ^
     -F map="{\"0\":[\"variables.files.0\"], \"1\":[\"variables.files.1\"]}" ^
     -F 0=@small_file.pdf ^
     -F 1=@big_file.docx
```



the operations `uploadFile` and `uploadFiles` do not exist in the ApplicationLayer, the above commands were for demonstration purpose only!

After sending these requests to the server, the server should know how to handle them properly. So the server should meet the specification to be able to handle the request. ApplicationLayer knows already how to handle them.

For clients there are already many libraries that meet the specification and produce such requests. For instance, for `JavaScript` the library `apollo-upload-client` could be used in combination with `apollo client`.

5.6.1. File Upload with ApplicationLayer

There are new graphql operations where uploading binary files is possible. The new operations are mutations and belong to the RepositoryService and are:

new graphql operations in nscale ApplicationLayer with file upload functionality

```
createDocument (content: [Upload]!,
                parentKey: String,
                documentObjectclassName: ObjectclassNameInput,
                properties:[PropertyInput],
                archived: Boolean,
                initialLock: Boolean)

createExtension (content: [Upload]!,
                 ownerKey: String,
                 type: ExtensionType,
                 properties: [PropertyInput])

updateContent (content: [Upload]!, resourceKey: String)

update (content: [Upload]!, resourceKey: String, props: [PropertyInput])

proceedContent (content: [Upload]!, documentKey: String)
```

```
proceed (content: [Upload]!, documentKey: String, props: [PropertyInput])
```



new operations

the above mutations are only available since the ApplicationLayer version 8.3.

The following example demonstrates the correct usage of one of the new ApplicationLayer mutations:

```
curl -v --basic -u admin:admin http://localhost:8080/nscalealinst1/graphql ^
     -F operations="{\"query\":\"mutation updateContent($file: [Upload]!,
                                                        $docKey: String!) {
                    RepositoryService_updateContent(content: $file,
                                                    resourceKey: $docKey)}\",
                     \"variables\":{\"file\":[null], \"docKey\":\"da$1$9285$2$1\"}}" ^
     -F map="{\"0\":[\"variables.file.0\"]}" ^
     -F 0=@one_giga.zip
```

The above example shows how to update the resource's content with a **single-content-item document** using the new mutation `updateContent`. Only one file should be uploaded to create single-content-item document.



When creating a single-content-item document with the new ApplicationLayer mutations, the variable `file` should be filled with single-element list `\"file\":[null]` **or** with single value `\"file\":null`. Consider adjusting the form-field *map* value when using the second variant, just like the following: `map="{\"0\":[\"variables.file\"]}"`.

Uploading multiple files to create **multi-content-item document** is also possible with the new ApplicationLayer mutations. The following example shows how to upload files to build a nscale multi-content-item document:

```
curl -u admin http://localhost:8080/nscalealinst1/graphql ^
     -F operations="{\"query\":\"mutation multiItemDocument($file: [Upload]!,
                                $parent: String!, $objclass: ObjectclassNameInput!,
                                $props: [PropertyInput]){
                                RepositoryService_createDocument(content: $file,
                                parentKey: $parent,
                                documentObjectclassName:$objclass,
                                initialLock: true,
                                properties:$props)}\",
                    \"variables\":{\"file\":[null,null,null],
                                  \"parent\":\"da$NOTSET$2$1$NOTSET\",
                                  \"props\":[{\"propertyName\":
                                  {\"IndexingPropertyName\": {\"name\":
                                  \"displayname\", \"areaName\": \"da\"}},
                                  \"value\":\"multiContentItemDoc\"}],
                                  \"objclass\":{\"name\":\"D1\",
```

```
                                  \"areaName\":\"da\"}}}" ^
     -F map="{\"0\":[\"variables.file.0\"], \"1\":[\"variables.file.1\"],
             \"2\":[\"variables.file.2\"]}" ^
     -F 0=@top_secret.pdf ^
     -F 1=@classified.docx ^
     -F 2=@dont_readme.txt
```

With the above example a multi-content-item document with the name `multiContentItemDoc` is created. This document has the content-type `multipart/mixed`.



If all files have the same content-type, then the multi-content-item document won't have the content-type `multipart/mixed`. Instead, it will have just the common content-type of these files,

Every mutation that accepts multiple files `[Upload]` can create nscale single-content-item documents when uploading only one file or multi-content-item documents when uploading multiple files.

Uploading multiple files to create **multiple single-content-item documents** is also possible with the following brand-new ApplicationLayer Graphql mutation:

```
createMultipleDocuments (files: [Upload]!,
                         parentKey: String!,
                         objectclassName: ObjectclassNameInput,
                         properties: [PropertyInput])
```

The property `displayname` of every created document will be automatically set to the name of the corresponding uploaded file. Other properties could be manually specified **for all** uploaded files, when wished. The parameter `objectclassName` is also optional, and if it is not specified, the default object class will be set. This mutation is useful if we want to upload a bunch of files with only one request. This could be useful for implementing client functionalities like `drag and drop files into nscale` or something similar.



It is not possible to specify the property values for each newly created document with the above mutation. Specified property values will apply to **all** newly created document **equally**, so be aware!

5.7. Downloading Files

Like file upload, file download with graphql is not straight forward since graphql supports only JSON representation format. To make graphql supports binary file download, a workaround is implemented where the servlet context is accessed and the binary data is directly written within the servlet http response. With this approach we can perform a pure binary file download that has great performance, and this is how file download is implemented in the Application Layer graphql api.

There are two brand-new operations in the Application Layer graphql api which support file download, and are part of the repository service:

new graphql operations file download functionality

```
downloadDocument (resourceKey: String, itemIndex: Int): Boolean

downloadMultipleDocuments (resourceKeys: [String]): Boolean
```



note that the operation parameter `itemIndex` is optional and will be needed only for multi content item documents.

With the operation `downloadDocument` we can download any document inside nscale system by delivering the resource key of that resource.

If the resource is a multi content item document then we will get as a result of the download process a zip file that contains all content items of that resource. If we want only a specific content item from the multi content item document, then we should specify the index of that content item with the operation parameter `itemIndex`.

The following example shows how to download a single content item document:

```
curl -v --basic -u admin:admin -O -J http://localhost:8080/nscalealinst1/graphql ^
     -F operations="{\"query\":\"query downloadADocument($resKey: String!){
                                repositoryService {
                                   downloadDocument(resourceKey: $resKey)
                                }
                               }\",
                    \"variables\":{\"resKey\":\"da$NOTSET$154$2$NOTSET\"}}"
```



to download file using `curl`, use the parameters `-O` and `-J`. The parameter `-J` let curl extract the file name from the Content-Disposition's `filename`.

Downloading a multi content item document is performed in the same way as above.

With the operation `downloadMultipleDocuments` we can download a list of resources at once with one request. The resources will be then zipped and delivered as expected.

The following `curl` command demonstrates how to download multiple resources with Application Layer graphql api:

```
curl -v --basic -u admin:admin -O -J http://localhost:8080/nscalealinst1/graphql ^
     -F operations="{\"query\":\"query downloadDocuments($resKeys: [String]!){
                                repositoryService {
                                   downloadMultipleDocuments(resourceKeys: $resKeys)
                                }
                               }\",
                    \"variables\":{\"resKeys\":[\"da$NOTSET$154$2$NOTSET\",
                                                \"da$NOTSET$153$2$NOTSET\"]}}"
```