HPC and Cloud 2023

GraphQL Web API Guide

You are reading the Altair® HPC and Cloud 2023

GraphQL Web API Guide (GG)

For the following HPC and Cloud products:

- Altair Grid Engine®

Updated 6/29/23

Contact Us

Altair

Altair Engineering, Inc., 1820 E. Big Beaver Road, Troy, MI 48083-2031 USA www.altair.com

Sales

hpcsales@altair.com 248.614.2400

Please send any questions or suggestions for improvements to agu@altair.com.

Technical Support

Need technical support? We are available from 8am to 5pm local times:

Location	Telephone	e-mail
Australia	+1 800 174 396	anz-pbssupport@india.altair.com
China	+86 (0)21 6117 1666	pbs@altair.com.cn
France	+33 (0)1 4133 0992	pbssupport@europe.altair.com
Germany	+49 (0)7031 6208 22	pbssupport@europe.altair.com
India	+91 80 66 29 4500 +1 800 208 9234 (Toll Free)	pbs-support@india.altair.com
Italy	+39 800 905595	pbssupport@europe.altair.com
Japan	+81 3 6225 5821	pbs@altairjp.co.jp
Korea	+82 70 4050 9200	support@altair.co.kr
Malaysia	+91 80 66 29 4500 +1 800 425 0234 (Toll Free)	pbs-support@india.altair.com
North America	+1 248 614 2425	pbssupport@altair.com
Russia	+49 7031 6208 22	pbssupport@europe.altair.com
Scandinavia	+46 (0)46 460 2828	pbssupport@europe.altair.com
Singapore	+91 80 66 29 4500 +1 800 425 0234 (Toll Free)	pbs-support@india.altair.com
South Africa	+27 21 831 1500	pbssupport@europe.altair.com
South America	+55 11 3884 0414	br_support@altair.com
UK	+44 (0)1926 468 600	pbssupport@europe.altair.com

Contents

Contents

About This Guide

Altair HPC and Cloud Products Covered in This Guide

In this guide, we cover using GraphQL with the following Altair HPC and cloud products:

- Altair Grid Engine

Document Conventions

`Abbr`eviation

The shortest acceptable abbreviation of a command or subcommand is underlined

Attribute

Attributes, parameters, objects, variable names, resources, types

`Command`

Commands such as `qmgr` and `scp`

Definition

Terms being defined

`File name`

File and path names

`Input`

Command-line instructions

Method

Method or member of a class

`Output`

Output, example code, or file contents

Syntax

Syntax, template, synopsis

`Utility`

Name of utility, such as a program

Value

Keywords, instances, states, values, labels

Notation

Optional Arguments

Optional arguments are enclosed in square brackets. For example, in the `qstat` man page, the `-E` option is shown this way:

qstat [-E]

To use this option, you would type:

```
qstat -E
```

Variable Arguments

Variable arguments (where you fill in the variable with the actual value) such as a job ID or vnode name are enclosed in angle brackets. Here's an example from the `pbsnodes` man page:

pbsnodes -v <vnode>

To use this command on a vnode named "my_vnode", you'd type:

```
pbsnodes -v my_vnode
```

Optional Variables

Optional variables are enclosed in angle brackets inside square brackets. In this example from the `qstat` man page, the job ID is optional:

qstat [<job ID>]

To query the job named "1234@my_server", you would type this:

```
qstat 1234@my_server
```

Literal Terms

Literal terms appear exactly as they should be used. For example, to get the version for a command, you type the command, then "--version". Here's the syntax:

qstat --version

And here's how you would use it:

```
qstat --version
```

Multiple Alternative Choices

When there are multiple options and you should choose one, the options are enclosed in curly braces. For example, if you can use either "-n" or "--name":

```
{-n | --name}
```

Where to Find More Information, Documentation, and Support

- Software packages or additional software licenses: contact Altair Sales: https://altair.com/contact-us
- Technical support: contact Altair Support at https://community.altair.com/community
- Documentation: click *Documentation* at https://community.altair.com/community
- For supported platforms and system requirements for Grid Engine, see the *Grid Engine Release Notes*
- For installation information for Grid Engine, see the *Grid Engine Installation Guide*

1 Introduction to Using GraphQL with Altair Grid Engine

1.1 Integrating Your Interface via the GraphQL Web API

You can integrate your site tools with Altair Grid Engine using the GraphQL web API. The API provides an interface for job submission, modification, monitoring, and deletion, as well as queries and operations on other Grid Engine objects such as queues and machines. We describe how to use the interface in Chapter 3, "Using the GraphQL Web API with Grid Engine Objects", on page 17.

1.2 Providing the GraphQL Web API

The Grid Engine administrator can configure Grid Engine to provide a GraphQL web API, by enabling the API at installation, or by modifying the bootstrap file for a running instance and then restarting the API thread. We describe these steps in Chapter 2, "Configuring Altair Grid Engine to Use GraphQL", on page 3.

We describe how the administrator provides the API endpoint in section 2.6, “Providing the API Endpoint”, on page 10, and we describe using the endpoint in section 3.2, “Getting Started with GraphQL”, on page 18.

1.3 User Authentication

The GraphQL web API requires a user authentication token. This token is provided by the administrator or by the web API user; we describe how to do this in section 2.7, “Providing Authentication”, on page 11.

Altair Grid Engine allows users to authenticate requests via the GraphQL web API using OpenID (OIDC)/OAuth2 authentication and JSON Web Token (JWT) authentication.

Authentication using OIDC requires connecting to an externally-provided OIDC service. This service is not part of the Altair Grid Engine distribution. We describe how to connect to an external OIDC service in section 2.7.2, “Using an OIDC Authentication Service”, on page 11.

Authentication using JWT does not require an external service; Grid Engine includes a tool called `hpcgentoken` for creating JWT tokens. We describe how to use the `hpcgentoken` tool in Section 2.7.3, "Setting up JWT-based Authentication".

2 Configuring Altair Grid Engine to Use GraphQL

Altair Grid Engine allows the administrator to provide a GraphQL web API that can be used by application developers (integrators). The administrator can enable the API at installation during either the interactive or automated installation, or after installation by modifying the bootstrap file and then restarting the API thread. In all cases, the administrator needs to do some additional configuration after enabling the web API.

2.1 Choosing Authentication Method

The GraphQL web API requires an authentication token when it contacts the `qmaster`. Each user requires their own token. Altair Grid Engine allows users to authenticate requests via the GraphQL web API using OpenID (OIDC)/ OAuth2 authentication and JSON Web Token (JWT) authentication.

Authentication using OIDC requires connecting to an externally-provided OIDC service. This service is not part of the Altair Grid Engine distribution. We describe how to connect to an external OIDC service in section 2.7.2, “Using an OIDC Authentication Service”, on page 11.

Authentication using JWT does not require an external service; Grid Engine includes a tool called `hpcgentoken` for creating JWT tokens. We describe how to use the `hpcgentoken` tool in section 2.7.3, “Setting up JWT-based Authentication”, on page 11.

Tokens generated via `hpcgentoken` have a default lifetime of 10 days. We recommend taking special care in the handling of compromised tokens (lost tokens, or tokens of users who should no longer have access to the Grid Engine cluster). We describe how to handle these tokens in section 2.7.5, “Revoking Tokens and User Access”, on page 15.

Both authentication methods can be used in parallel. Sites starting initially with token-based authentication can add OIDC authentication later and disable the JWT token method once all users are migrated. We describe how to disable JWT in section 2.7.6, “Disabling JWT Authentication Method”, on page 15.

You can use an identity service such as Dex to integrate with an LDAP or AD service, so that the *sub* field in each token matches the user's UNIX user ID at the workload manager.

Regardless of the authentication method users accessing Altair Grid Engine through the GraphQL API need to be added to the appropriate access lists; we describe how in section 2.5.1, “Adding GraphQL Users to Access Lists”, on page 9.

2.2 Enabling the GraphQL Web API At Installation

2.2.1 Enabling the GraphQL API During Interactive Installation

To enable the GraphQL web API during an interactive installation:

1. If you are using `hpcgentoken` to generate tokens or you are providing an https API endpoint, and have not already created your certificates, create the required certificates; see section 2.7.3.1, “Creating Certificates”, on page 11
2. Run the interactive installation:
   a. Start `install_qmaster` with the `-gqlapi` option:

      ```
      ./install_qmaster ... -gqlapi ...
      ```

   b. The installation script then asks for the values of bootstrap file parameters:
      - For the API endpoint, set api_threads, api_port, tls_key_pair_path

        See section 2.6, “Providing the API Endpoint”, on page 10
      - If you are using an external OIDC service provider, set api_oidc_provider_url, api_oidc_client_id, and tls_key_pair_path
      - If you are using the `hpcgentoken` tool, set api_gentoken_auth_issuers; we strongly recommend setting tls_key_pair_path (see section 2.6.2, “Choosing http or https Mode for API Endpoint”, on page 10)
      - Optionally set otel_endpoint

      We describe the parameters for the GraphQL web API in section 2.4, “Bootstrap and Auto-install Template Parameters for GraphQL Web API”, on page 6.
3. Finish configuring Grid Engine for the GraphQL web API; see section 2.5, “Additional Configuration for GraphQL Web API”, on page 9
4. Each web API user requires an authentication token; see section 2.7, “Providing Authentication”, on page 11

For complete interactive installation instructions, see the *Grid Engine Installation Guide*.

2.2.2 Enabling the GraphQL Web API During Automated Installation

To enable the GraphQL web API during an automated installation, you need to create a template file that contains the parameters you will use for the GraphQL web API. The Grid Engine installation includes an automated installation template file. You can copy this, make the necessary edits, and then use your copy for the automated installation. The (unedited) template file is in the distribution here:

```
$SGE_ROOT/util/install_modules/inst_template.conf
```

1. If you are using `hpcgentoken` to generate tokens or you are providing an https API endpoint, and have not already created your certificates, create the required certificates; see section 2.7.3.1, "Creating Certificates", on page 11
2. Create a template file that contains all the parameters you need, including the web API parameters. See section 2.4, "Bootstrap and Auto-install Template Parameters for GraphQL Web API", on page 6.
   - Set parameters for the API endpoint: set SGE_API_PORT and SGE_API_THREADS; if you are providing an https API endpoint, set AGE_TLS_KEY_PAIR_PATH. See section 2.6, "Providing the API Endpoint", on page 10
   - If you are using an external OIDC service, make sure you set AGE_API_OIDC_PROVIDER_URL and AGE_API_OIDC_CLIENT_ID
   - If you are using JWT tokens:
     - make sure you set AGE_API_GENTOKEN_AUTH_ISSUERS
     - we strongly recommend setting AGE_TLS_KEY_PAIR_PATH
   - Optionally set AGE_OTEL_ENDPOINT
3. Specify parameters for the `inst_sge` script using the command-line flags; you can combine these flags. We list the `inst_sge` flags in the following table:

Table 2-1: Command-line Flags for the inst_sge Script

Flag	Description
`-auto <path to template>`	Enables automated installation when it points to full path to template
`-m`	Installs master host; same as `install_qmaster`
`-x`	Installs execution host; same as `install_execd`
`-sm`	Installs shadow master host
`-s`	Installs submit host
`-csp`	Enables enhanced security features (CSP)
`-gqlapi`	Enables GraphQL web API

4. Run the automatic installation. If your copy of the template file is named age_autoinstall.txt:

   ```
   ./install_qmaster -auto <path>/age_autoinstall.txt <options>
   ```

   See the *Grid Engine Installation Guide* for instructions.
5. Finish configuring Grid Engine for the GraphQL web API; see section 2.5, "Additional Configuration for GraphQL Web API", on page 9
6. Each web API user requires an authentication token; see section 2.7, "Providing Authentication", on page 11

2.3 Enabling the GraphQL Web API After Installation

If you are already running Grid Engine, and you want to be able to use the GraphQL web API with your existing instance of Grid Engine, you can do that without having to restart `qmaster`. To enable the GraphQL web API after installation:

1. If you are using `hpcgentoken` to generate tokens or you are providing an https API endpoint, and have not already created your certificates, create the required certificates; see section 2.7.3.1, "Creating Certificates", on page 11
2. Specify the GraphQL web API bootstrap configuration options in the bootstrap file.
   - Set parameters for the API endpoint: set api_port and api_threads; if you are providing an https API endpoint, set tls_key_pair_path. See section 2.6, "Providing the API Endpoint", on page 10
   - If you are using an external OIDC service service, set api_oidc_provider_url, api_oidc_client_id, and tls_key_pair_path
   - If you are using JWT tokens:
     - make sure you set api_gentoken_auth_issuers
     - we strongly recommend setting tls_key_pair_path
   - Optionally set otel_endpoint

   See section 2.4, "Bootstrap and Auto-install Template Parameters for GraphQL Web API", on page 6 for a description of these options.

   For more about the API endpoint, see section 2.6, "Providing the API Endpoint", on page 10.
3. Kill and then restart the API thread:

   ```
   qconf -kt api; qconf -at api
   ```

   To show the list of threads:

   ```
   qconf -stl
   ```
4. Finish configuring Grid Engine for the GraphQL web API; see section 2.5, "Additional Configuration for GraphQL Web API", on page 9
5. Each web API user requires an authentication token; see section 2.7, "Providing Authentication", on page 11

2.4 Bootstrap and Auto-install Template Parameters for GraphQL Web API

- For an interactive installation, Grid Engine gets its bootstrap parameters from your answers to questions.
- For an automated installation, Grid Engine uses parameters from a template file that you create.
- When you enable the GraphQL web API after installation, you set the bootstrap parameters in the bootstrap file.

The bootstrap file is in `$SGE_ROOT/$SGE_CELL/common/bootstrap`. For a complete description of the bootstrap file, see `sge_bootstrap(5)`.

The following table shows the bootstrap and template parameters used for the GraphQL web API:

Table 2-2: Bootstrap Options for the GraphQL Web API

Bootstrap Parameter	Auto-install Template Parameter	Description
api_gentoken_auth_issuers	AGE_API_GENTOKEN_AUTH_ISSUERS	Comma-separated list of accepted JWT token issuers. Used only when using JWT tokens generated via `hpcgentoken`. Default: “none” (no issuers are accepted)
api_oidc_client_id	AGE_API_OIDC_CLIENT_ID	Used with external OIDC service. The client identifier generated when registering the application with the OIDC provider. For validation of OIDC ID tokens passed in http(s) requests.

Table 2-2: Bootstrap Options for the GraphQL Web API

Bootstrap Parameter	Auto-install Template Parameter	Description
api_oidc_provider_url	AGE_API_OIDC_PROVIDER_URL	Used with external OIDC service. URL of the OIDC provider, for example https://accounts.google.com For validation of OIDC ID tokens passed in http(s) requests.
api_port	SGE_API_PORT	Port number on which the API thread listens for incoming http requests. The API endpoint is {`http`|`https`}`://<qmaster host>:`*<api_port>* Valid port numbers are between 1 and 65535 The default port is 8080. Required for both OIDC and JWT.
api_threads	SGE_API_THREADS	Number of API threads created initially in qmaster. See section 2.6.1, "The API Thread", on page 10. Valid values: *0*, *1* Default: *0* Required for both OIDC and JWT.
otel_endpoint	AGE_OTEL_ENDPOINT	Enables the API thread to send tracing information to an `opentelemetry` service. When otel_endpoint contains a hostname and port, the API thread provides tracing data to the `opentelemetry` service. When not specified or set to *none* , tracing is disabled. Format: *<hostname>:<port>* Default: *none* (tracing is disabled) The otel_endpoint parameter is optional.
tls_key_pair_path	AGE_TLS_KEY_PAIR_PATH	Full path to TLS key pair directory; this directory contains `key.pem` and `key.cert` files. By default, the API thread starts in `http` mode, where the protocol is *`http://<qmaster host>:<API port>`* . To start the API thread in `https` mode, set the tls_key_pair_path parameter to a directory containing valid certificates. When the tls_key_pair_path parameter is configured to point to valid certificates, the API endpoint protocol changes from *`http://<qmaster host>:<API port>`* to *`https://<qmaster host>:<API port>`* The tls_key_pair_path parameter is optional. If it is not specified or if its value is "*none*", the API endpoint stays in `http` mode. Default value: *none* (`http` mode) Required for OIDC when providing `https` API endpoint. Strongly recommended when using JWT tokens.

2.4.1 Example Bootstrap File

```
admin_user              none
default_domain          none
ignore_fqdn             true
spooling_method         classic
spooling_lib            libspoolc
spooling_params         /opt/ge/default/common;/opt/ge/default/spool/qmaster
binary_path             /opt/ge/bin
qmaster_spool_dir       /opt/ge/default/spool/qmaster
security_mode           none
communication_params    none
listener_threads        2
pworker_threads         2
worker_threads          4
reader_threads          2
otel_endpoint           none
scheduler_threads       1
jvm_threads             0
lo_threads              0
api_threads             1
api_port                8080
tls_key_pair_path       /opt/ge/default/common/certs
api_gentoken_auth_issuers gentoken
api_oidc_client_id      110921278006-
    a08frb90rlva5e12aft0r7icmfsmsop9.apps.googleusercontent.com
api_oidc_provider_url   https://accounts.google.com
```

2.5 Additional Configuration for GraphQL Web API

2.5.1 Adding GraphQL Users to Access Lists

Anyone who will use the GraphQL API needs to be in the correct access list:

- Add the users of the GraphQL API to the `sudoers` user list. For example:
  `qconf -au <username> sudoers`
- Add the administrator username to the `sudomasters` user list. For example, if we are adding root to the list:
  `qconf -au root sudomasters`

2.5.2 Allowing Web API to Process Slot Requests

1. Add a custom resource to allow the web API to process slot requests. Open the editor:
   ```
   qconf -mc
   ```
2. Add the following line to the table:
   ```
   ncpus ncpus INT <= YES YES 1 0 NO 0.000000 YES YES
   ```
3. Save the changes to the table.

2.6 Providing the API Endpoint

The API endpoint used by the GraphQL web API has this syntax:

{http/https}://<qmaster host>:<API port>

The endpoint for cURL users is the GraphQL endpoint with "/graphql" appended:

{http/https}://<qmaster host>:<API port>/graphql

The *API port* is specified in the api_port bootstrap parameter; default is *8080*. See section 2.4, "Bootstrap and Auto-install Template Parameters for GraphQL Web API", on page 6.

The *qmaster host* must be specified as a fully-qualified domain name.

2.6.1 The API Thread

The API thread provides a GraphQL API to the qmaster. You can have up to one API thread. The default number of threads is zero. The number of API threads is specified in the api_threads bootstrap parameter; see section 2.4, "Bootstrap and Auto-install Template Parameters for GraphQL Web API", on page 6.

To kill and then restart the API thread:

```
qconf -kt api; qconf -at api
```

To show the list of threads:

```
qconf -stl
```

2.6.2 Choosing http or https Mode for API Endpoint

By default, the API thread starts in `http` mode. If you will use JWT tokens, we strongly recommend `https` mode. To start the API thread in `https` mode:

1. Create a self-signed root CA; see section 2.7.3.1, "Creating Certificates", on page 11
2. Set the tls_key_pair_path bootstrap parameter to a directory containing valid certificates. See section 2.4, "Bootstrap and Auto-install Template Parameters for GraphQL Web API", on page 6

So for example if tls_key_pair_path is set to a directory containing valid certificates, the `qmaster` host is host1.mydomain.com, and the API port is set to 8080, the API endpoint is:

```
https://host1.mydomain.com:8080
```

and the cURL endpoint is:

```
https://host1.mydomain.com:8080/graphql
```

2.7 Providing Authentication

2.7.1 Requirements for Tokens

Each token must have the following characteristics:

- The *sub* field in the token must be the same as the UNIX user ID for that user, because the workload manager uses the *sub* field in the token as the UNIX user ID of the user who is invoking the API
- If the token is a JWT token, the token must reside in the user's home directory in a file named `.wlm_token`
- If the token is a JWT token, the token file must be readable by the user

2.7.2 Using an OIDC Authentication Service

If you use an external OIDC authentication service, make sure that the generated tokens meet the requirements of the workload manager, described above in Section 2.7.1, "Requirements for Tokens". In particular, the workload manager uses the *sub* field in a token as the UNIX user ID of the user who is invoking the API, so the *sub* field **must** match the UNIX user ID used in the back end of the workload manager. If an external authentication service provides its own unique ID in the *sub* field that does not match the person's UNIX user ID at the workload manager, this will not work.

We cover the required steps for using an external OIDC authentication service in the sections describing how to configure Grid Engine.

2.7.3 Setting up JWT-based Authentication

2.7.3.1 Creating Certificates

Tokens need to be signed in order to ensure token integrity and to reliably avoid username spoofing. In addition, using an https API endpoint requires a self-signed root CA. You can create a self-signed CA and use it for both JWT token authentication and the https API endpoint. We recommend creating the self-signed CA as the Grid Engine administrative user (`admin_user` in the bootstrap file).

1. Log into the `qmaster` host as administrator
2. Make sure that the necessary environment variables for Grid Engine are set; typically you can do this by sourcing the `$SGE_ROOT/$SGE_CELL/common/settings.[c]sh` file
3. Create the `certs` directory:

   ```
   umask 022
   mkdir $SGE_ROOT/$SGE_CELL/common/certs
   ```

4. Create a self-signed certificate. For example:

   If you are using OpenSSL 1.0:

   ```
   umask 277
   openssl req -x509 -nodes -newkey rsa:4096 -keyout $SGE_ROOT/$SGE_CELL/common/certs/key.pem \
   -out $SGE_ROOT/$SGE_CELL/common/certs/cert.pem -sha256 -days 365 -subj "/CN=$HOSTNAME" \
   -extensions san -config \
   <(echo '[req]'; echo 'distinguished_name=req';echo '[san]'; echo "subjectAltName=DNS:$HOSTNAME")
   ```

If you are not using OpenSSL 1.0:

```
umask 277
openssl req -x509 -newkey rsa:4096 -keyout $SGE_ROOT/$SGE_CELL/common/certs/key.pem \
-out $SGE_ROOT/$SGE_CELL/common/certs/cert.pem -sha256 -days 365 -subj \
"/CN=$HOSTNAME" -addext "subjectAltName=DNS:$HOSTNAME" -noenc
```

5. Set the permissions for the certificates:

```
chmod 644 $SGE_ROOT/$SGE_CELL/common/certs/cert.pem
```

2.7.3.2 The hpcgentoken Authentication Utility

You can use the `hpcgentoken` tool to create web API user tokens. The `hpcgentoken` command is found in `$SGE_ROOT/utilbin/<arch>/hpcgentoken`. The `hpcgentoken` command is designed to be used either in restricted-access mode, where only the administrator has permissions to run it and generate tokens, or in user mode, where web API users can run the command to generate their own tokens. By default, `hpcgentoken` runs in restricted-access mode, where only the administrator can generate tokens.

We recommend that administrators, rather than API users, create any JWT tokens. The administrator can then provide the tokens to the users. If the administrator creates the tokens, the administrator can track which users have which tokens, which makes management of compromised tokens much easier.

We strongly recommend that administrators store all created tokens in a secure place so that they have a straightforward means for revoking tokens. We describe how to revoke tokens in section 2.7.5.1, “Revoking Tokens”, on page 15.

Tokens generated via `hpcgentoken` have a default lifetime of 10 days.

2.7.3.2.i Prerequisites for Using hpcgentoken

The `hpcgentoken` tool uses authentication certificates that reside on the qmaster host. If you will use the `hpcgentoken` tool and you do not have certificates already, create them.

When validating GraphQL requests via JWT, Grid Engine checks whether `hpcgentoken` is in the list of accepted authorization issuers. To allow JWT-based validation of GraphQL requests, *hpcgentoken* must be listed in the api_gentoken_auth_issuers bootstrap parameter. (To disable JWT-based authentication, remove *hpcgentoken* from the api_gentoken_auth_issuers bootstrap parameter and restart qmaster; see Section 2.7.6, "Disabling JWT Authentication Method".)

2.7.3.2.ii Restricted-access Mode for hpcgentoken

When the administrator runs `hpcgentoken` in restricted-access mode:

- Only the administrator can run the command
- The administrator uses the -u option to specify the username for whom the token is to be generated
- The administrator generates all tokens on behalf of users
- The setuid bit is not required on the `hpcgentoken` binary
- The permissions for the `hpcgentoken` binary are 0700 (the default)

2.7.3.2.iii Web API User Mode for hpcgentoken

When non-root web API users run `hpcgentoken` to generate their own tokens:

- Each web API user generates their own token(s)
- The person running `hpcgentoken` does not use the -u option
- The setuid bit is required on the `hpcgentoken` binary, to allow it access to the /var/spool/pbs/common/certs/key.pem file generated by the administrator
- The permissions for the `hpcgentoken` binary need to be 4755

By default, `hpcgentoken` runs in restricted-access mode and has permissions of 0700.

If you want web API users to be able to run `hpcgentoken`:

- Set the permissions for `hpcgentoken` to 4755
- Make sure that the `gentoken` binary has the SUID bit set. As root:

  `chmod u+s ${SGE_ROOT}/utilbin/<arch>/gentoken`

  This allows the `hpcgentoken` tool to read the `/var/spool/pbs/common/certs/key.pem` file generated by the administrator.

2.7.3.2.iv Syntax of hpcgentoken Command

hpcgentoken [-e <lifetime>] [-h] -k <absolute path to TLS key pair> [-u <user ID>] [-V] [-v <valid start>] [-version] [-h]

Options:

-e <seconds>

Sets token lifetime. Default lifetime is 10 days (864,000 seconds).

Optional.

-h, -help, --help

Shows a help menu. Optional.

-k

Absolute path to the TLS key pair to be used to sign the token. Required.

-u

Used with the default restricted-access mode, where only the administrator can run `hpcgentoken`. Specifies user on whose behalf the administrator is generating a token.

When used in this mode, the `hpcgentoken` tool verifies that the invoking user is actually root, and that the username specified via the -u option actually exists on the host where `hpcgentoken` is invoked.

-V

Prints username along with token in the format *<username>:<token>*

Optional.

-v <seconds>

Token becomes valid after the specified number of seconds from now.

Optional.

-version, --version

Shows the version of the web API being used.

Optional.

2.7.3.3 Administrator Steps for Generating Tokens

2.7.3.3.i Managing User Tokens

We recommend that you create an easy way to track JWT tokens. Before you give JWT tokens to web API users, create a directory where each created token is stored and associated with its intended user. This directory can be anywhere. For example:

```
umask 077
mkdir $SGE_ROOT/$SGE_CELL/common/certs/usertokens
```

2.7.3.3.ii Generating User Tokens

These instructions are for the administrator who generates tokens so that non-root users can use the GraphQL web API. To generate a token for a user:

1. Log into the qmaster host as root
2. Run the `hpcgentoken` command:
   - Specify the full path to the certificates
   - Specify the UNIX user ID of the user
   - The token is printed on stdout and must be redirected to a file. Capture the output in a token file that is associated with the user name, in your token-managing directory:

     ```
     $SGE_ROOT/utilbin/$ARCH/hpcgentoken -k $SGE_ROOT/$SGE_CELL/common/certs  \
     -u <username> > $SGE_ROOT/$SGE_CELL/common/certs/usertokens/<username>-wlm_token
     ```

   - The default lifetime of 10 days (864,000 seconds) can be extended or shortened via the -e flag. For example, the following creates a token with a lifetime of one year:

     ```
     $SGE_ROOT/utilbin/$ARCH/hpcgentoken -e 31536000
     ```

3. Give a copy of the token file to the user; make sure it is in their home directory, named .wlm_token, and they have read permissions for it. The web API user needs to copy the token file into their home directory with read-only permission for the owner:

   ```
   umask 277
   cp <token file> ~/.wlm_token
   ```

2.7.3.4 API User Steps for Generating Tokens

2.7.3.4.i Administrator Steps to Allow Users to Generate Tokens

The `hpcgentoken` tool needs access to the private key file, so `hpcgentoken` should have the setuid bit set, and should be owned by the administrator. The administrator does the following:

```
chown <administrator username> $SGE_ROOT/utilbin/$ARCH/hpcgentoken
chmod 4755 $SGE_ROOT/utilbin/$ARCH/hpcgentoken
```

2.7.3.4.ii Web API User Steps to Generate Tokens

We strongly recommend that any web API user who generates a token send a copy to the administrator, so that the administrator can revoke it if the token is lost or stolen.

These instructions are for non-root users who need a token for the GraphQL web API. To generate a token you can use for the web API:

1. Log into the qmaster host as the UNIX user ID you use in the workload manager
2. Set your umask

   ```
   umask 277
   ```

3. Call `hpcgentoken` and specify the full path to the certificates. You can put the token in your home directory:

   ```
   $SGE_ROOT/utilbin/$ARCH/hpcgentoken -k $SGE_ROOT/$SGE_CELL/common/certs  > ~/.wlm_token
   ```

   The token looks like this:

   ```
   eyJhbGciOiJQUzI1NiIsInR5cCI6IkpXVCJ9.eyJjdXN0b21Ub2tlbiI6dHJ1ZSwiZXhwIjoxNjgzODg1NzQzLCJpYXQiOjE2OD
       MwMjE3NDMsImlzcyI6ImdlbnRva2VuIiwianRpIjoiMDc3NzgzNzEtM2U1Mi00YTE3LWE5ZDktNGYwMDYzYzdiNmFiIiwib
       mJmIjoxNjgzMDIxNzQzLCJzdWIiOiJqb2dhIn0.n0ZpUXS3J2mqXKB6rOZqNOSl2Mx-
       fo_MkgLnNEcgP9Pzni2jBAVbl1svIk1NGQZRyIhSqHn8oFAMMFsBvYmL7zIWzfo9bXNP8rIsFw8RQmXc8vSVW9q2uaugbdi
       j-5aItVkEZ_uVItvFXnQmfhsstfOUQ-
       vEiz92qaaiv2L_JEZaQzuLKG58yLT8xwqd5i04RElypHGDsER2jRKtImTRXSXpova3j_Xhe97qDxm9gnrn8qS23FTlyuqZB
   ```

```
Avtb0NdTrpiSdge4P696uPKXBthRzk1cMStKXFxFo4_ImCwwiClbGju1qYsFoudUnWsN4CHkgHwQFIdcowcQOEBMvQEPWCJ
yG4hu2qTHkk1Q3FXIiEAWegAE4IDwWrzMhRZi9b4MjQ_6EgzJfNY9cjrMGW21ZuvDFMGh371HFkewjZ0atxLaHzsa-ySp-
sccrwldcxooASMmY-DgruArZjhnsi4aDj4icNHfHIdf4yV8OEV_Hk4H6v_gooVYizSLSH3zt4jsGnIPgX60-5Naz14-
cN0MHyTWkf7brz9acTML-lzmROQqkME7ZmH7fdh0-
uCYkbfZyYCVsK_yJRh6E1RycuMZGR2ZdUtjGEGl7T2dQaL5c5tgnsjaJzMEikqwHod0eml5Nc9xjlXHVByqG6GOSjv4BGgC
CCCWBsPpzTEIJdJAuZk
```

4. Make sure that the token is in your home directory, in a file called .wlm_token, and you have read permissions for it
5. Give a copy of the token to your administrator

2.7.4 Decoding a JWT Token

You can decode a JWT token. There are several online services which allow you to decode the content of a JWT token. Be careful when sending sensitive information to a website on the Internet.

For example, there is https://jwt.io/

2.7.5 Revoking Tokens and User Access

If a token is compromised by being lost or acquired by the wrong user, you can revoke it in order to prevent incoming requests with that token from being validated. If a misbehaving user has access to tokens, you can revoke their access.

2.7.5.1 Revoking Tokens

Token revocation works by putting bad tokens into a specific token revocation directory, and checking incoming requests against these tokens.

Whenever a new token is added to the revoked token directory, this is automatically detected and the token is automatically revoked.

However, if an existing token in the directory is modified, the administrator must do “touch” on the directory in order to trigger revocation of the modified token.

As the Grid Engine administrative user, create the token revocation directory:

```
mkdir -p $SGE_ROOT/$SGE_CELL/common/server_priv/revoked_tokens
```

Whenever a bad token is identified, copy it into the token revocation directory:

```
cp $SGE_ROOT/$SGE_CELL/common/certs/usertokens/<user>-wlmtoken \
$SGE_ROOT/$SGE_CELL/common/server_priv/revoked_tokens
```

2.7.5.2 Revoking User Access

To revoke user access to Grid Engine via the GraphQL web API, remove the user from the `sudoers` list:

```
qconf -du <username> sudoers
```

See section 2.5.1, “Adding GraphQL Users to Access Lists”, on page 9.

2.7.6 Disabling JWT Authentication Method

If you were using both OIDC and JWT authentication, and you want to disable the JWT method after migrating all your web API users to OIDC:

1. Remove *hpcgentoken* from the api_gentoken_auth_issuers bootstrap parameter and the AGE_API_GENTOKEN_AUTH_ISSUERS template parameter
2. Restart `qmaster`

2.8 Where to Find More Information for Grid Engine

- Where to find documentation: click *Documentation* at https://community.altair.com/community
  - For supported platforms and system requirements for Grid Engine, see the *Grid Engine Release Notes*
  - For installation information for Grid Engine, see the *Grid Engine Installation Guide*

3 Using the GraphQL Web API with Grid Engine Objects

In this section we describe how to use the GraphQL web API to submit, modify, and delete jobs, and query Grid Engine objects such as queues and machines.

Chapter Contents

3.1 Prerequisites for Using the GraphQL Web API

- The GraphQL web API must be enabled; check this with your administrator
- You must be in the list of `sudoers`; check this with your administrator

3.2 Getting Started with GraphQL

3.2.1 Get Authentication Token

In order to use the GraphQL web API to contact `qmaster`, you need an authentication token provided by your administrator.

1. Get an authentication token from your administrator
2. Put this token in a file in your home directory called `.wlm-token`
3. Make sure you have read permission for the token

3.2.2 The API Endpoint

A GraphQL server uses a single API endpoint to resolve queries. All requests to the server including fetching and mutating data must be directed to this endpoint. To connect to the GraphQL server, point your browser to the URL of the API endpoint.

The API endpoint used by the GraphQL web API has this syntax:

{http|https}://<qmaster host>:<API port>

For example, if the mode is *https* and the `qmaster` host is host1.mydomain.com and the API port is 8080, the API endpoint is:

```
https://host1.mydomain.com:8080
```

Check with your administrator for the location of the API endpoint.

3.2.3 Using cURL

You can connect to the server using cURL commands. If you are using cURL, the endpoint is the same GraphQL endpoint with "/graphql" appended:

{http|https}://<qmaster host>:<API port>/graphql

For example, if the mode is *https* and the `qmaster` host is host1.mydomain.com and the API port is 8080, the cURL endpoint is:

```
https://host1.mydomain.com:8080/graphql
```

cURL commands are in the following format, where you include the Authorization field, and use the Bearer schema with your generated JWT token. Copy the token into the command:

```
curl '<server URL>' -H 'Authorization: Bearer <token>'
```

3.2.4 User Authentication for the GraphQL Web API

The GraphQL web API requires a user authentication token. This token is provided by the administrator. This token needs to be in a file in your home directory called `.wlm-token` and you need read permission for it. Include the token in the header for every request.

3.2.5 Using a GraphQL IDE

We recommend using a GraphQL IDE; this gives you an easy-to-use interface for writing and testing queries. Some IDEs provide helpful debugging and exploration tools. The IDE we use for our examples is the GraphQL Playground.

3.2.5.1 User Authentication in the GraphQL Playground

Before you can use the IDE, you must provide a user authentication token. Go to the *REQUEST HEADERS* tab at the bottom left of the screen. Include the Authorization field, and use the Bearer schema with your authorization token. Copy the token into the line:

```
{
"Authorization" : "Bearer <token>"
}
```

Be sure to refresh the page so that you can register your changes and access the Docs.

3.2.5.2 Writing Queries in the GraphQL Playground

After you are authenticated, you can write queries by typing into the query area and then pressing the triangular *play* button to inspect the output of the query.

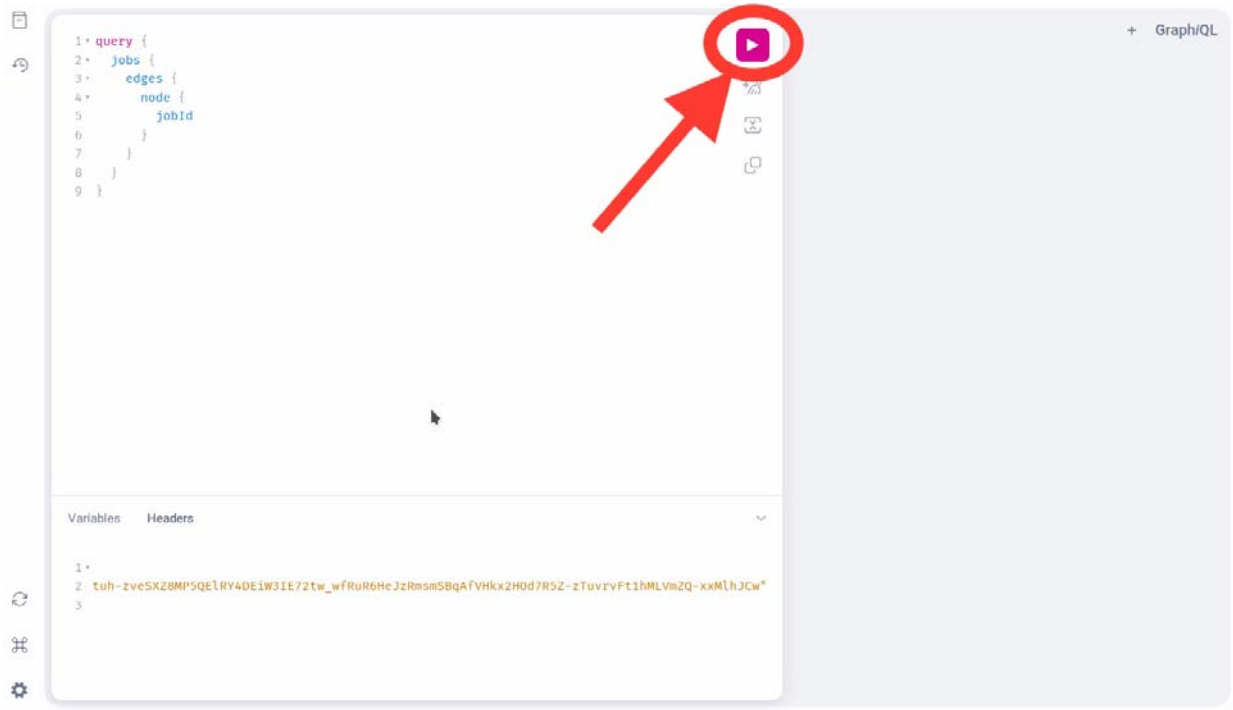

Figure 3-1: Writing queries

Query output appears to the right:

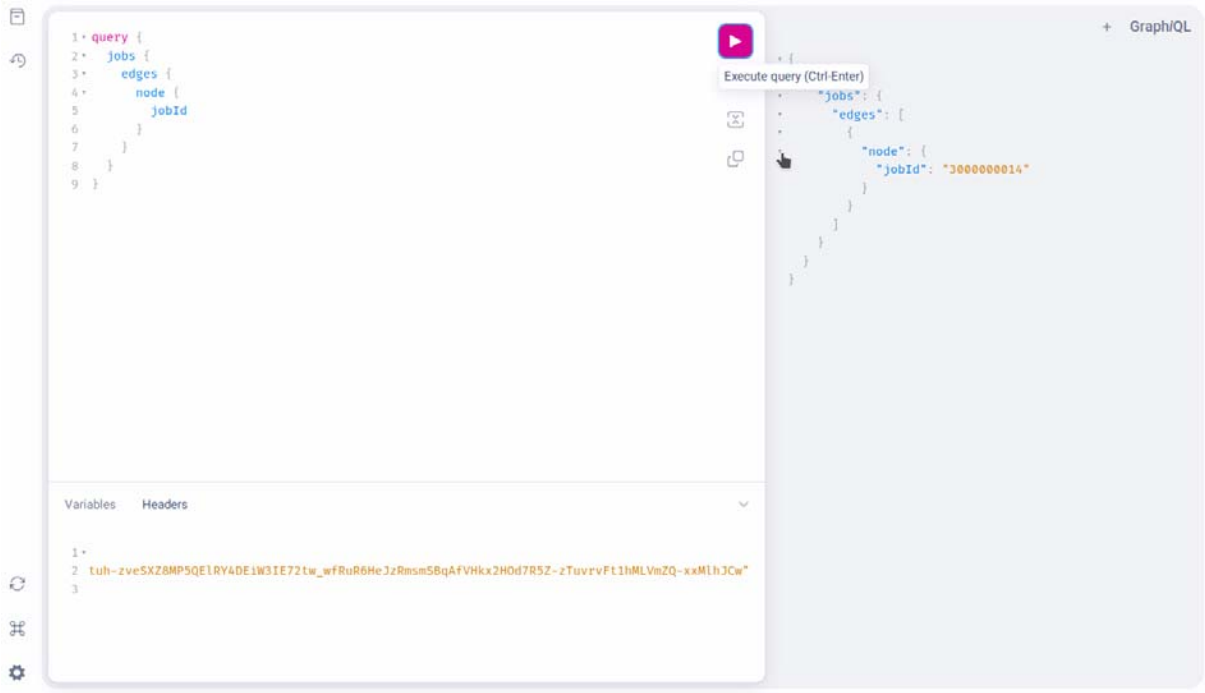

Figure 3-2: Query output

3.2.5.3 Using Tabs to Manage Requests

You can track each request (query, mutation, etc.) in its own tab. Use the "+" button to add a tab. Observe that in the example below, the current tab is unnamed because the query is unnamed:

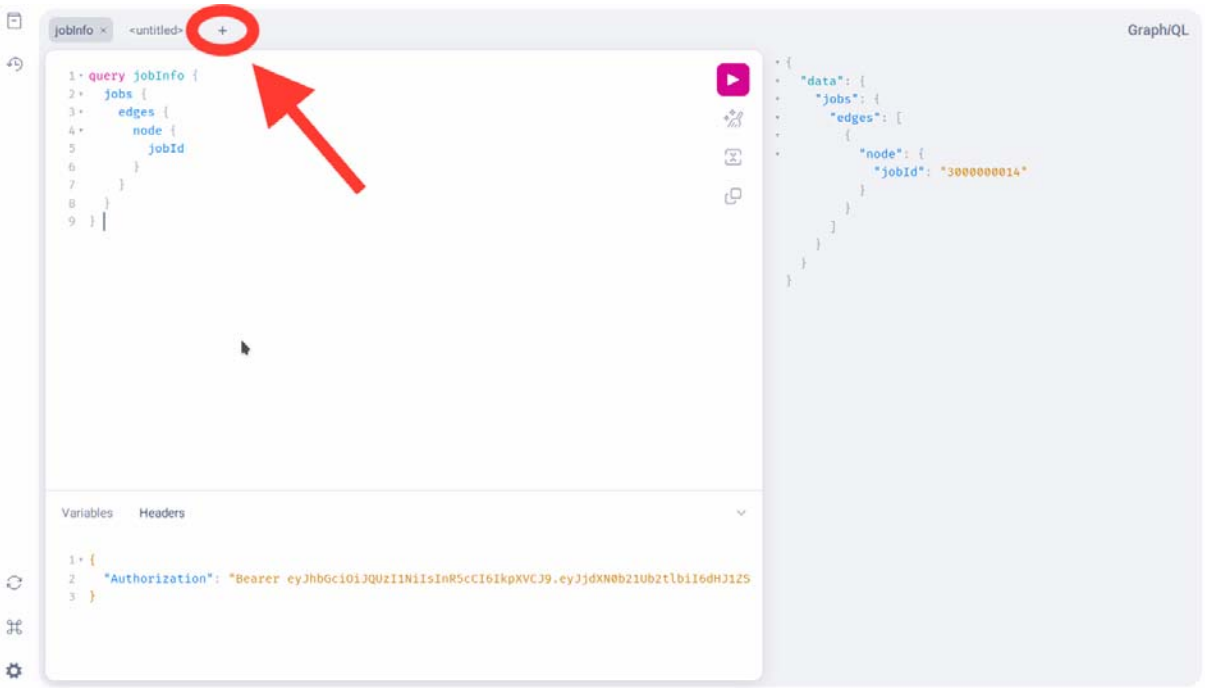

Figure 3-3: New tab

3.2.5.4 Naming Requests

We recommend naming each request; each tab is named for its request. Observe that in the example below, the tab has the same name as the request: :

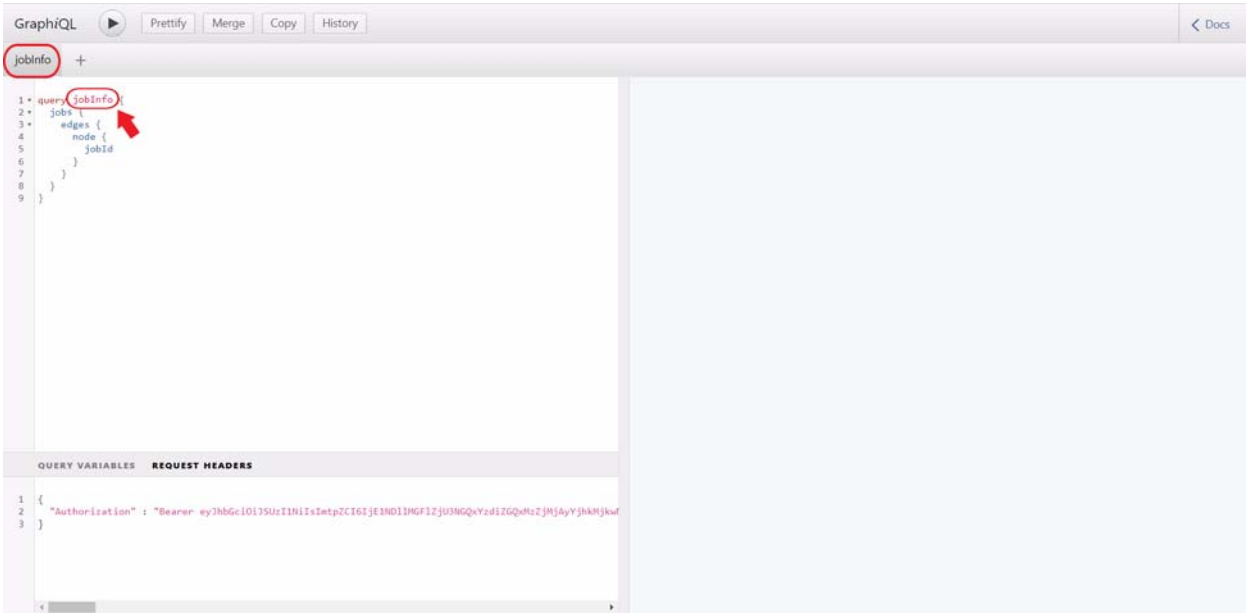

Figure 3-4: Naming a tab

3.2.5.5 Navigating and Using the Docs

To navigate to the Docs, click on the Docs tab in the top right corner.

This displays all query types and the allowed types of mutations that can be performed. To inspect the details of a specific query or mutation, click on it to display its definition and description, its arguments, type details and output types. You can also click on any of the types listed in the arguments or output to get more details and descriptions of their usage. You can also search the Docs using specific keywords. Type your search terms into the *Search Schema* bar and click on the relevant results that you want to learn more about.

3.2.6 Terminology in Requests

Some attributes in the GraphQL schema have the same name as attributes in our workload managers. For example, you will see the terms “node” and “edge” in queries; these are nodes and edges of standard graph theory. These nodes are not to be confused with compute nodes.

3.3 Submitting Jobs

3.3.1 Recommendations for Job Submission

We recommend crafting your queries to return information about newly-created jobs so that you can confirm successful submission and track jobs. For example, you will want to know the job ID of each newly submitted job.

We also recommend returning possible error information to help you figure out what went wrong with an erroneous job submission.

3.3.2 Submitting a Simple Job

3.3.2.1 Basic Syntax for Submitting a Job

You use mutations to create and submit jobs. We show an example mutation named "myJobSubmission" containing basic syntax for submitting a simple job via the GraphQL web API:

```
mutation myJobSubmission {
    createJob (input: {remoteCommand: "<job command/script> <job command/script arguments>"}) {
        node {
            jobId
        }
    }
}
```

For example:

```
mutation myJobSubmission {
    createJob (input: {remoteCommand: "/bin/sleep 3600"}) {
        node {
            jobId
        }
    }
}
```

Submitting the same job via `qsub`:

```
qsub -b y <job command/script> <job command/script arguments>
```

For example:

```
qsub -b y sleep 3600
```

3.3.3 Returning Job ID, Status, and Errors

We recommend returning information about the created job so that you can confirm successful submission and track the job ID. We also recommend returning possible error information.

3.3.3.1 Returning Job ID and Error Information

Our example mutation myJobSubmission2 returns the job ID, a possible error code, and a possible error message:

```
mutation myJobSubmission2 {
    createJob(input: {remoteCommand: "/bin/sleep 36000"}) {
        node {
            jobId
        }
        error {
            errorCode
            errorMessage
        }
    }
}
```

3.3.3.1.i Example of Data Returned on Successful Job Submission

If the submission via myJobSubmission2 is successful, the error information returned is null, as shown here:

```
{
    "data": {
        "createJob": {
            "node": {
                "jobId": "3000000003"
            },
            "error": null
        }
    }
}
```

3.3.3.1.ii Example of Data and Error Information Returned on Unsuccessful Job Submission

If the submission is not successful, for example because we requested a queue which does not exist, we get an error message. Our mutation myErroneousSubmission shows an example of a submission that will fail. It requests a non-existent queue:

```
mutation myErroneousSubmission {
    createJob (input: {
        remoteCommand: "/bin/sleep 36000"
        queue: {name: "no such queue"}
    }) {
        node {
            jobId
        }
        error {
            errorCode
            errorMessage
        }
    }
}
```

Result of using myErroneousSubmission:

```
{
    "data": {
        "createJob": {
            "node": null,
            "error": {
                "errorCode": -1,
                "errorMessage": "Job was rejected because job requests unknown queue \"no such
    queue\"\n"
            }
        }
    }
}
```

3.3.3.2 Example of Submitting Job via Both GraphQL Playground and cURL

Here is an example of a mutation that sends a request to the server to create and submit a job that executes the sleep command for 1000 seconds. The request also returns the job ID of the submitted job:

```
mutation myJobSubmission3 {
    createJob (
        input: {
            remoteCommand: "/bin/sleep 1000"
            name: "sleep"
        }
    ) {
        node {
            jobId
        }
    }
}
```

The same job submission via a cURL command:

```
curl '<server URL>' -H 'Content-Type: application/json' -H \
'Authorization: Bearer <token>' --data-binary '{"query":"mutation \
{ createJob ( input: { remoteCommand: \"/bin/sleep 1000\" name: \"sleep\" } ) \
{ node { jobId } }}"}'
```

The output:

```
{
    "data": {
        "createJob": {
            "node": {
                "jobId": "125"
            }
        }
    }
}
```

3.3.4 Submitting Complex Jobs

3.3.4.1 Submitting an Array Job and Returning Job ID and Status

Our example myArraySubmission shows basic syntax for submitting an array job and returning the job ID and status:

```
mutation myArraySubmission {
    createJob (
        input: {
            remoteCommand: "<job command/script> <job command/script arguments>",
                array : {
                    beginIndex:<starting index>
                    endIndex:<ending index>
                }
        }
    ) {
        node {
            jobId
            status{state}
        }
    }
}
```

For example:

```
mutation myArraySubmission {
    createJob (
        input: {
            remoteCommand: "sleep 3600",
                array : {
                    beginIndex:1
                    endIndex:11
                }
        }
    ) {
        node {
            jobId
            status{state}
        }
    }
}
```

Submitting the same array job via `qsub`:

```
-bash-4.2$ qsub -t <starting index>-<ending index> -b y <job command/script> <job command/script
    arguments>
```

For example:

```
-bash-4.2$ qsub -t 1-11 -b y sleep 3600
```

3.3.4.2 Submitting a Parallel Job

Our example myParallelSubmission shows basic syntax for submitting a parallel job and returning the job ID and status:

```
mutation myParallelSubmission {
    createJob (
        input: {
            remoteCommand: "/bin/sleep 100",
            resourcesRequested: {
                jobResources: {
                    index: ""
                    slots: 2
                    physicalMemory: 1000
                },
                peName:"mytestpe"
                taskCount: {min: 1, max: 3}
            }
        }
    ) {
        node {
            jobId
            status {
                state
            }
        }
        error {
            errorCode
            errorMessage
        }
    }
}
```

Here is the result from myParallelSubmission:

```
{
    "data": {
        "createJob": {
            "node": {
                "jobId": "3000000024",
                "status": {
                    "state": 0
                }
            },
            "error": null
        }
    }
}
```

Here is the equivalent qsub command:

```
qsub -pe mytestpe 1-3 -l ncpus=2,h_rss=1000K /bin/sleep 100
```

3.3.4.3 Submitting Sequential Job Requesting Memory Limit

Our example mySequentialMemLimitSub shows basic syntax for submitting a sequential job requesting a memory limit and returning the job ID and status:

```
mutation mySequentialMemLimitSub {
    createJob (input: {
        remoteCommand: "/bin/sleep 36000"
        resourcesRequested: {
            jobResources: {
                index:""
                physicalMemory: 4000
            }
        }
    }) {
        node {
            jobId
        }
        error {
            errorCode
            errorMessage
        }
    }
}
```

Here is the result from mySequentialMemLimitSub:

```
{
    "data": {
        "createJob": {
            "node": {
                "jobId": "3000000029"
            },
            "error": null
        }
    }
}
```

Here is the equivalent qsub command:

```
qsub -l h_rss=4000K /bin/sleep 36000
```

3.3.4.4 Submitting Dependent Jobs

When using the GraphQL web API, you can submit dependent jobs, but a job can depend on just one other job. Our mutation myDependentJob shows an example:

```
mutation myDependentJob {
    createJob(input: {
        remoteCommand: "/bin/sleep 36000"
        dependOnJob: {jobId: "3000000035"}
    }) {
        node {
            jobId
            dependOnJobs {jobId}
        }
        error {
            errorCode
            errorMessage
        }
    }
}
```

Output of myDependentJob:

```
{
    "data": {
        "createJob": {
            "node": {
                "jobId": "3000000036",
                "dependOnJobs": [
                    {
                        "jobId": "3000000035"
                    }
                ]
            },
            "error": null
        }
    }
}
```

Submitting dependent jobs via the web API is more restricted than via Grid Engine commands. You can use `qsub -hold_jid` to make a job depend on multiple other jobs, and you can specify job names and wildcards.

3.3.5 Nested Queries

You can create nested queries that return information about nested objects. You can create mutations that return nested attributes. For example, you can use a mutation to create a job that also queries the nested attributes of an object such as a queue:

```
mutation myNestedMutation {
    createJob (
        input: {
            remoteCommand: "/bin/bash hello.sh"
            name: "hello"
        }
    ) {
        node {
            jobId
            status {
                state
            }
            queue {
                name
                type
            }
        }
    }
}
```

And its output:

```
{
    "data": {
        "createJob": {
            "node": {
                "jobId": "126",
                "status": {
                    "state": 0
                },
                "queue": {
                    "name": "workq",
                    "type": "BatchInteractive"
                }
            }
        }
    }
}
```

3.3.5.1 Limit on Query Nest Depth

The GraphQL web API allows queries with nesting up to two deep. For example, our query "jobsByMachine" requests information about machines and the jobs running on them:

```
query jobsByMachine {
    machines {
        edges {
            node {
                name
                hostname
                state
                runningJobs {
                    jobId
                }
            }
        }
    }
}
```

Here is the equivalent cURL command:

```
curl '<server URL>' -H 'Content-Type: application/json' -H 'Authorization: Bearer <token>' --
    data-binary '{"query":"query { machines { edges { node { name hostname state runningJobs {
    jobId } } } }}"}'
```

Here is example output from jobsByMachine:

```
{
    "data": {
        "machines": {
            "edges": [
                {
                    "node": {
                        "name": "hostname",
                        "hostname": "hostname",
                        "state": 1,
                         "runningJobs": [
                            {
                                "jobId": "123"
                            },
                            {
                                "jobId": "124"
                            }
                        ]
                    }
                }
            ]
        }
    }
}
```

3.4 Modifying Jobs

You use mutations to modify jobs. We show an example mutation named “myJobModification” containing basic syntax for modifying a job and returning basic job information via the GraphQL web API:

```
mutation myJobModification {
    updateJob (jobId: “<target job ID>”, input:  {
        <target attribute>: “<new value>”
    } ) {
        node {
            jobId
            name
        }
    }
}
```

For example, to update the job name:

```
mutation myJobModification {
    updateJob (jobId: “16”, input:  {
        name: “TEST”
    } ) {
        node {
            jobId
            name
        }
    }
}
```

The mutation above roughly corresponds to the following `qalter` command:

```
-bash-4.2$ qalter -<flag for target attribute> <target value> <target job ID>
```

For example:

```
-bash-4.2$ qalter -N TEST 16
```

3.4.1 Example of Changing Multiple Job Attributes

This mutation updates two job attributes. Here is a request to change both the name and rerunnable attributes of the job:

```
mutation {
    updateJob (
        jobId: "125"
        input: {
            name: "hello world"
            rerunnable: true
        }
    )   {
        node {
            name
            jobId
            rerunnable
        }
    }
}
```

And its output:

```
{
    "data": {
        "updateJob": {
            "node": {
                "name": "hello world",
                "jobId": "125",
                "rerunnable": true
            }
        }
    }
}
```

3.5 Deleting Jobs

You use mutations to delete jobs. We recommend creating job deletion mutations that return the job ID and/or name, and any error information. We show an example mutation named "myJobDeletion" containing basic syntax for deleting a job and returning basic job information via the GraphQL web API:

```
mutation myJobDeletion {
    deleteJob (jobId: "3000000037" input: {}) {
        node {
            jobId
            name
        }
        error {
            jobId
            errorCode
            errorMessage
        }
    }
}
```

Output for an existing job:

```
{
    "data": {
        "deleteJob": {
            "node": {
                "jobId": "3000000037",
                "name": "sleep 100"
            },
            "error": null
        }
    }
}
```

Repeating the mutation gives an error, because the job does no longer exists:

```
{
    "data": {
        "deleteJob": {
            "node": null,
            "error": {
                "jobId": "3000000037",
                "errorCode": -1,
                "errorMessage": "Not found"
            }
        }
    }
}
```

Here is the syntax of the equivalent `qdel` command:

```
-bash-4.2$ qdel <job ID>
```

For example:

```
-bash-4.2$ qdel 3000000037
```

3.5.1 Job Deletion with Force Option

Our next example mutation myJobDeletion3 deletes the job with job ID 3000000029 using the `-f` option, and returns job ID and error information:

```
mutation myJobDeletion2 {
     deleteJob (jobId: “3000000029” input: { force:true }) {
           node {
               jobId
           }
           error {
                jobId
                errorCode
                errorMessage
           }
     }
}
```

Result of using myJobDeletion3:

```
{
    “data”: {
        “deleteJob”: {
            “node”: {
                “jobId”: “3000000029”,
            },
            “error”: null
        }
    }
}
```

Equivalent `qdel` command:

```
qdel -f 3000000029
```

3.6 Querying Objects: Getting Information

3.6.1 Simple Queries

Assume that we have one job currently running on the server. To query the server for the jobId of all the current jobs we have running (which is just one), we can use the following query format in the GraphQL Playground:

```
query {
    jobs {
        edges {
            node {
                jobId
            }
        }
    }
}
```

Or equivalently, use the following cURL command in your CLI:

```
curl '<server URL>' -H 'Content-Type: application/json' -H 'Authorization: Bearer <token>' --
    data-binary '{"query":"query { jobs { edges { node { jobId }}}}"}'
```

This query will then output the following JSON packet, giving us exactly the amount of information that we requested:

```
{
    "data": {
        "jobs": {
            "edges": [
                {
                    "node": {
                        "jobId": "123"
                    }
                }
            ]
        }
    }
}
```

3.7 Getting Information about Jobs

3.7.1 Listing Jobs

You use queries to get job information. Our example myJobList query shows syntax for querying jobs that meet certain filter criteria:

```
query myJobList {
    jobs (filter: {<filter>: <filter value>} ) {
        edges {
            node {
                jobId
                name
                owner
                status {
                    state
                }
                submitTime
                startTime
                allocatedMachines {
                    name
                }
            }
        }
    }
}
```

For example, to query jobs that are queued and pending (where state = *0*):

```
query {
    jobs (filter: {states: 0} ) {
        edges {
            node {
                jobId
                name
                owner
                status { state }
                submitTime
                startTime
                allocatedMachines {name}
            }
        }
    }
}
```

The GraphQL query above roughly corresponds to the following `qstat` command:

```
-bash-4.2$ qstat -<filter> <filter value>
```

For example:

```
-bash-4.2$ qstat -s p
```

3.7.1.1 Example of Job Query and Result

We show an example query named “myJobList2” which returns jobs requesting at least 3 megabytes of memory:

```
query myJobList2 {
    jobs (filter: {resourcesRequested: {physicalMemory: {value: 3000, comp: GE}}}) {
        edges {
            node {
                jobId
                priority
                name
                owner
                status {
                    state
                }
                submitTime
                startTime
                queue {
                    name
                }
            }
        }
        totalCount
    }
}
```

Here is what the result might look like:


```
{
    "data": {
        "jobs": {
            "edges": [
                {
                    "node": {
                        "jobId": "3000000027",
                        "priority": 0,
                        "name": "sleep 36000",
                        "owner": "user2",
                        "status": {
                            "state": 7
                        },
                        "submitTime": 1683818790549000,
                        "startTime": 1683818791001000,
                        "queue": {
                            "name": "all.q"
                        }
                    }
                },
                {
                    "node": {
                        "jobId": "3000000029",
                        "priority": 0,
                        "name": "sleep 36000",
                        "owner": "user2",
                        "status": {
                            "state": 7
                        },
                        "submitTime": 1683818791424000,
                        "startTime": 1683818792006000,
                        "queue": {
                            "name": "all.q"
                        }
                    }
                },
                {
                    "node": {
                        "jobId": "3000000031",
                        "priority": 0,
                        "name": "sleep 36000",
                        "owner": "user2",
                        "status": {
                            "state": 7
                        },
```

```
                        "submitTime": 1683823238660000,
                        "startTime": 1683823239001000,
                        "queue": {
                            "name": "all.q"
                        }
                    }
                }
            ],
            "totalCount": 3
        }
    }
}
```

You can use the `qstat` command to make similar queries, but not exactly the same one.

3.8 Getting Information about Queues

3.8.1 Showing the Queue List

You use queries to get queue information. We show an example query named "myQueueListing" containing basic syntax for showing a list of queues:

```
query myQueueListing {
    queues (orderBy:Q_NAME_ASC) {
        edges {
            node {
                name
                type
                stateCount {
                    total
                    queued
                    queuedHeld
                    held
                    stagingIn
                    stagingOut
                    running
                    suspended
                    exiting
                    done
                    failed
                    moved
                }
            }
        }
    }
}
```

Here is a command with a similar result:

```
qstat -g c
```

3.8.1.1 Example of Listing Queue Information

Here we use myQueueListing2 to get more detailed queue information:

```
query myQueueListing2 {
    queues () {
        edges {
            node {
                    name
                    resourcesAvail {
                        slots
                        physicalMemory
                        virtualMemory
                        wallClockTime
                        cpuTime
                    }
                    resourcesAssigned {
                        slots
                        physicalMemory
                        virtualMemory
                        wallClockTime
                        cpuTime
                    }
                    stateCount {
                        total
                        queued
                        queuedHeld
                        held
                        running
                        suspended
                    }
                }
            }
        }
    }
}
```

If we have a queue named “all.q”, the result might look like this:


```
{
    “data”: {
        “queues”: {
            “edges”: [
                {
                    “node”: {
                        “name”: “all.q”,
                        “resourcesAvail”: {
                            “slots”: 210,
                            “physicalMemory”: 4194304,
                            “virtualMemory”: 16777216,
                            “wallClockTime”: 7820,
                            “cpuTime”: 10800
                        },
                        “resourcesAssigned”: {
                            “slots”: 3,
                            “physicalMemory”: 5604,
                            “virtualMemory”: 55520,
                            “wallClockTime”: 2551,
                            “cpuTime”: 0
                        },
                        “stateCount”: {
                            “total”: 3,
                            “queued”: 0,
                            “queuedHeld”: 0,
                            “held”: 0,
                            “running”: 3,
                            “suspended”: 0
                        }
                    }
                }
            ]
        }
    }
}
```


There is no direct command-line equivalent for this.

3.9 Getting Information about Machines

3.9.1 Showing the List of Machines

You use queries to get machine information. We show an example query named "myMachineListing" containing basic syntax for querying for a list of machines and returning machine information via the GraphQL web API:

```
query myMachineListing {
    machines (orderBy:M_NAME_ASC) {
        edges {
            node {
                name
                state
                load
                machineOS
                machineOSVersion
                machineArchitecture
            }
        }
    }
}
```

Here is a command that gives similar results:

```
qhost
```

3.9.1.1 Example of Getting Machine Information

Our example query myMachineDetails shows how we can return details such as slots, memory, etc.:

```
query myMachineDetails {
     machines (orderBy:M_NAME_ASC) {
        edges {
            node {
                name
                machineArchitecture
                resourcesAvail {
                    slots
                    physicalMemory
                }
                resourcesAssigned {
                    physicalMemory
                }
            }
        }
    }
}
```

If we have several machines, the result of using myMachineDetails might look like this:


```
{
    "data": {
        "machines": {
            "edges": [
                {
                    "node": {
                        "name": "host1.mydomain.com",
                        "machineArchitecture": "lx-amd64",
                        "resourcesAvail": {
                            "slots": 10,
                            "physicalMemory": 8009068
                        },
                        "resourcesAssigned": {
                            "physicalMemory": 3344064
                        }
                    }
                },
                {
                    "node": {
                        "name": "global",
                        "machineArchitecture": null,
                        "resourcesAvail": {
                            "slots": 0,
                            "physicalMemory": 0
                        },
                        "resourcesAssigned": {
                            "physicalMemory": 0
                        }
                    }
                },
                {
                    "node": {
                        "name": "host2.mydomain.com",
                        "machineArchitecture": "lx-amd64",
                        "resourcesAvail": {
                            "slots": 10,
                            "physicalMemory": 16424696
                        },
                        "resourcesAssigned": {
                            "physicalMemory": 4516400
                        }
                    }
                },
                {
                    "node": {
```

```
                        "name": "host3.mydomain.com",
                        "machineArchitecture": "lx-amd64",
                        "resourcesAvail": {
                            "slots": 10,
                            "physicalMemory": 65806596
                        },
                        "resourcesAssigned": {
                            "physicalMemory": 2726324
                        }
                    }
                },
                {
                    "node": {
                        "name": "host4.mydomain.com",
                        "machineArchitecture": "lx-amd64",
                        "resourcesAvail": {
                            "slots": 10,
                            "physicalMemory": 1000284
                        },
                        "resourcesAssigned": {
                            "physicalMemory": 404396
                        }
                    }
                },
                {
                    "node": {
                        "name": "template",
                        "machineArchitecture": null,
                        "resourcesAvail": {
                            "slots": 0,
                            "physicalMemory": 0
                        },
                        "resourcesAssigned": {
                            "physicalMemory": 0
                        }
                    }
                }
            ]
        }
    }
}
```


The equivalent command:

```
qhost
```

3.10 Getting Information about Parallel Environment Objects

3.10.1 Showing the PE List

Here is an example of a query for a list of parallel environment objects:

```
query myPEList {
    parallelEnvs () {
        edges {
            node {
                name
            }
        }
    }
}
```

Here is a command with a similar result:

```
qconf -spl
```

3.11 Complex Queries

3.11.1 Querying Multiple Attributes

Here is an example of a more complex query that queries multiple attributes: the totalCount of jobs, the jobId of each job, and its status:

```
query {
    jobs {
        totalCount
        edges {
            node {
                jobId
                status {
                    state
                }
            }
        }
    }
}
```

And its output,

```
{
    "data": {
        "jobs": {
            "totalCount": 2,
            "edges": [
                {
                    "node": {
                        "jobId": "123",
                        "status": {
                            "state": 7
                        }
                    }
                },
                {
                    "node": {
                        "jobId": "124",
                        "status": {
                            "state": 7
                        }
                    }
                }
            ]
        }
    }
}
```

3.12 Refining Queries

3.12.1 Using Filters to Constrain and Specify Output

You can use filters to constrain and specify the amount of data returned by a query or the amount of data you will be mutating. We have defined certain filter types that you may use such as the JobsFilter or MachinesFilter (you can read more about these by searching the Docs). Otherwise, you can filter using the “filter” keyword and specify a certain value for an attribute of the schema type you are interested in querying. Here is an example of a query that requests the job ID of all the jobs whose withSubJobs attribute is *true* (meaning the job is an array job):

```
query {
    jobs (filter: {withSubJobs: true}) {
        edges {
            node {
                jobId
            }
        }
    }
}
```

And its cURL command equivalent:

```
curl '<server URL>' -H 'Content-Type: application/json' -H 'Authorization: Bearer <token>' --
    data-binary '{"query":"query { jobs (filter: {withSubJobs: true}) { edges { node { jobId
    }}}}"}'
```

Then, because we have no jobs with subjobs, the server returns an empty result:

```
{
    "data": {
        "jobs": {
            "edges": []
        }
    }
}
```

3.12.2 Using Filters on Nested Attributes

You can use filters when querying nested attributes in order to request specific information about a nested object, given a specific list of attributes of the main object. For example, to request information about the machines that contain running jobs:

```
query {
    jobs (filter: {states: 7}){
        edges {
            node {
                jobId
                allocatedMachines {
                    name
                    resourcesAssigned {
                        slots
                    }
                }
            }
        }
    }
}
```

And its output:

```
{
    "data": {
        "jobs": {
            "edges": [
                {
                    "node": {
                        "jobId": "123",
                        "allocatedMachines": [
                            {
                                "name": "hostname",
                                "resourcesAssigned": {
                                    "slots": 2
                                }
                            }
                        ]
                    }
                },
                {
                    "node": {
                        "jobId": "124",
                        "allocatedMachines": [
                            {
                                "name": "hostname",
                                "resourcesAssigned": {
                                    "slots": 2
                                }
                            }
                        ]
                    }
                }
            ]
        }
    }
}
```

Similarly, if we want to request information about the running jobs on the machines that have a slot value of 2, we could use the following query:

```
query {
    machines (filter: {
        resourcesAssigned: {slots: {value: 2 }}}
    ){
        edges {
            node {
                name hostname
                state
                runningJobs {
                    jobId
                }
            }
        }
    }
}
```

And its output:

```
{
    "data": {
        "machines": {
            "edges": [
                {
                    "node": {
                        "name": "hostname",
                        "hostname": "hostname",
                        "state": 1,
                        "runningJobs": [
                            {
                                "jobId": "123"
                            },
                            {
                                "jobId": "124"
                            }
                        ]
                    }
                }
            ]
        }
    }
}
```

3.12.3 Limiting Amount of Data Returned

You can also use limits to limit the amount of data that is returned by the server, using the "count" keyword. Here is an example of query that limits the return data of the server to a maximum of 5 job IDs:

```
query {
    jobs (count: 5) {
        edges {
            node {
                jobId
            }
        }
    }
}
```

And its cURL command equivalent:

```
curl '<server URL>' -H 'Content-Type: application/json' -H 'Authorization: Bearer <token>' --
    data-binary '{"query":"query {jobs (count: 5) { edges {node { jobId }}}}"}'
```

3.12.4 Introduction to Paginating Large Requests

When your request will return a large number of elements, you can paginate the output.

You can employ *forward pagination*, where the request returns a specific count of elements after a specific start point, or *backward pagination*, where the request returns a specific count of elements before a specific start point.

Each paginating query specifies a count of elements to return via the "`count: <count>`" statement. If you want to begin at the first element, your first query does not need to specify a start point. Subsequent queries do need to specify a start point via the "`from: <start point>`" statement. Each query does need to return the next start point.

We introduce cursor-based pagination in this section. A cursor is a location in a page. You can use a GraphQL request to query for the location of an element in the current page of data. The startCursor is the first element in the current page, and the endCursor is the last element in the current page; both are fields in the pageInfo object. These are numbers encoded in Base64. In graphical representation of data and cursors, data are graph nodes, and cursors are graph edges. The PageInfo object looks like this:

Table 3-1: PageInfo Object

Field	Type	Description
hasPreviousPage	Boolean	True if there is a previous page
hasNextPage	Boolean	True if there is a next page
startCursor	Base64 string	Cursor (edge) of first node in current page
endCursor	Base64 string	Cursor (edge) of last node in current page

You can find out more about these and other cursors in the Docs.

3.12.4.1 Example of Forward Pagination

Assume we have four jobs currently running, with IDs={0, 1, 2, 3} and we want to get the job IDs of all jobs in batches of two. In this example, the elements to return are jobs, the number of elements per page is two, and we query for the location (the endCursor) where the subsequent query will start:

```
query myFirstJobs {
    jobs (count: 2) {
        edges {
            node {
                jobId
            }
        }
        pageInfo {
            endCursor
        }
    }
}
```

This query outputs the first two job IDs, and the endCursor:

```
{
    "data": {
        "jobs": {
            "edges": [
                {
                    "node": {
                        "jobId": "123"
                    }
                },
                {
                    "node": {
                        "jobId": "124"
                    }
                }
            ],
            "pageInfo": {
                "endCursor": "MQA="
            }
        }
    }
}
```

In the subsequent query, we specify the start location to be the value returned in endCursor ("MQA="):

```
query myNextJobs {
    jobs (count: 2, from: "MQA=") {
        edges {
            node {
                jobId
            }
        }
    }
}
```

This outputs the next batch of job IDs:

```
{
    "data": {
        "jobs": {
            "edges": [
                {
                    "node": {
                        "jobId": "125"
                    }
                },
                {
                    "node": {
                        "jobId": "126"
                    }
                }
            ]
        }
    }
}
```

If we wanted to continue paginating, we would have expanded the above query to request the endCursor, and we would keep using the value of the endCursor as the “from” value for the next query.

Equivalently, if you want to perform the whole sequence of queries using cURL commands, here are the above queries as cURL commands:

```
curl '<server URL>' -H 'Content-Type: application/json' -H 'Authorization: Bearer <token>' --
    data-binary '{"query":"query { jobs (count: 2) { edges { node { jobId } } pageInfo { endCursor
    }}}"}'
curl '<server URL>' -H 'Content-Type: application/json' -H 'Authorization: Bearer <token>' --
    data-binary '{"query":"query { jobs (count: 2, from: \"MQA=\") { edges { node { jobId }}}}"}'
```

3.12.4.2 Example of Backward Pagination

Using our previous example, we will also perform backward pagination to return the first two jobs. First, we request the startCursor, the location of the first item in the current page (and the current page is the second page):

```
query myCurrentFirstIndex {
    jobs (from: "MQA=") {
        pageInfo {
            startCursor
        }
    }
}
```

This outputs a cursor that points to the job with jobId=2, since that is the first item of the current page:

```
{
    "data": {
        "jobs": {
            "pageInfo": {
                "startCursor": "MgA="
            }
        }
    }
}
```

Then, to perform backward pagination, we use the same query format, use the value of startCursor for the "from" statement, and use a *negative* count value:

```
query myPreviousJobs {
    jobs (count: -2, from: "MgA=") {
        edges {
            node {
                jobId
            }
        }
    }
}
```

This returns the job IDs of the first two jobs:

```
{
    "data": {
        "jobs": {
            "edges": [
                {
                    "node": {
                        "jobId": "124"
                    }
                },
                {
                    "node": {
                        "jobId": "123"
                    }
                }
            ]
        }
    }
}
```

3.13 Caveats for the Grid Engine GraphQL Web API

3.13.1 Limit on Query Nest Depth

The GraphQL web API allows queries with nesting up to two deep. See section 3.3.5.1, “Limit on Query Nest Depth”, on page 30.

3.13.2 Limit on Dependent Jobs

A dependent job can depend on at most one other job. See section 3.3.4.4, “Submitting Dependent Jobs”, on page 28.

Index