Tanium™ API Gateway User Guide

Version 1.2.53

April 08, 2022

The information in this document is subject to change without notice. Further, the information provided in this document is provided "as is" and is believed to be accurate, but is presented without any warranty of any kind, express or implied, except as provided in Tanium's customer sales terms and conditions. Unless so otherwise provided, Tanium assumes no liability whatsoever, and in no event shall Tanium or its suppliers be liable for any indirect, special, consequential, or incidental damages, including without limitation, lost profits or loss or damage to data arising out of the use or inability to use this document, even if Tanium Inc. has been advised of the possibility of such damages.

Any IP addresses used in this document are not intended to be actual addresses. Any examples, command display output, network topology diagrams, and other figures included in this document are shown for illustrative purposes only. Any use of actual IP addresses in illustrative content is unintentional and coincidental.

Please visit https://docs.tanium.com for the most current Tanium product documentation.

This documentation may provide access to or information about content, products (including hardware and software), and services provided by third parties ("Third Party Items"). With respect to such Third Party Items, Tanium Inc. and its affiliates (i) are not responsible for such items, and expressly disclaim all warranties and liability of any kind related to such Third Party Items and (ii) will not be responsible for any loss, costs, or damages incurred due to your access to or use of such Third Party Items unless expressly set forth otherwise in an applicable agreement between you and Tanium.

Further, this documentation does not require or contemplate the use of or combination with Tanium products with any particular Third Party Items and neither Tanium nor its affiliates shall have any responsibility for any infringement of intellectual property rights caused by any such combination. You, and not Tanium, are responsible for determining that any combination of Third Party Items with Tanium products is appropriate and will not cause infringement of any third party intellectual property rights.

Tanium is committed to the highest accessibility standards for our products. To date, Tanium has focused on compliance with U.S. Federal regulations - specifically Section 508 of the Rehabilitation Act of 1998. Tanium has conducted 3rd party accessibility assessments over the course of product development for many years and has most recently completed certification against the WCAG 2.1 / VPAT 2.3 standards for all major product modules in summer 2021. In the recent testing the Tanium Console UI achieved supports or partially supports for all applicable WCAG 2.1 criteria. Tanium can make available any VPAT reports on a module-by-module basis as part of a larger solution planning process for any customer or prospect.

As new products and features are continuously delivered, Tanium will conduct testing to identify potential gaps in compliance with accessibility guidelines. Tanium is committed to making best efforts to address any gaps quickly, as is feasible, given the severity of the issue and scope of the changes. These objectives are factored into the ongoing delivery schedule of features and releases with our existing resources.

Tanium welcomes customer input on making solutions accessible based on your Tanium modules and assistive technology requirements. Accessibility requirements are important to the Tanium customer community and we are committed to prioritizing these compliance efforts as part of our overall product roadmap. Tanium maintains transparency on our progress and milestones and welcomes any further questions or discussion around this work. Contact your sales representative, email Tanium Support at support@tanium.com, or email accessibility@tanium.com to make further inquiries.

Table of contents

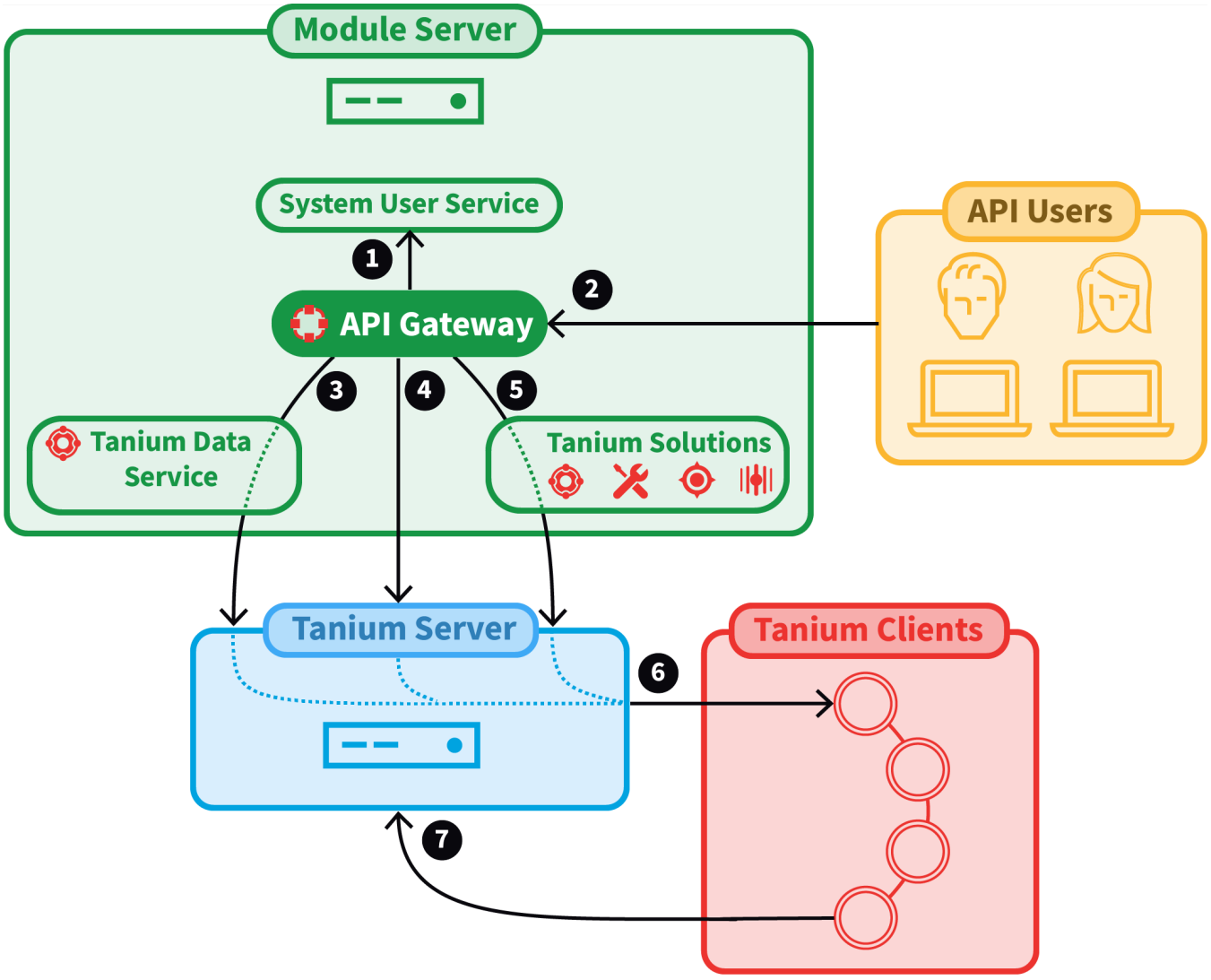
API Gateway overview

Tanium™ API Gateway provides a single and stable API integration point for various Tanium solutions. It is designed for Tanium partners and customers interested in building integrated solutions with the Tanium™ Core Platform.

API Gateway topology

The API Gateway service runs on the Tanium™ Module Server, and requires the System User service to run background processes. The API Gateway service passes user requests to Tanium™ Data Service or the Tanium™ Server; the Tanium Server collects data from Tanium™ Clients or updates data based on the request, and returns the results to the API Gateway service. The API Gateway service passes the results to the user that submitted the request.

API Gateway topology



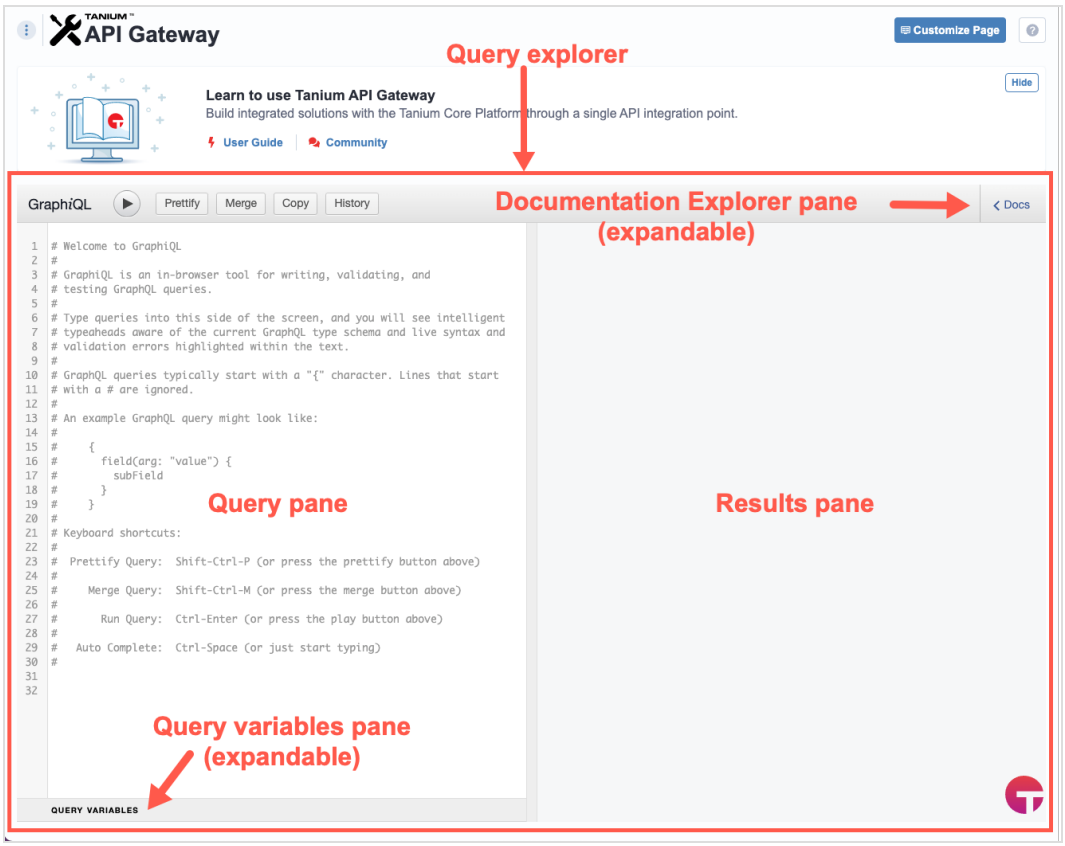
With the API Gateway, information passes between users and the Tanium Server through the following workflow (matching the numbers in API Gateway topology on page 7):

1. The API Gateway requires the System User service to be running to perform background functions.
2. Users send requests to the API Gateway to be performed on endpoints, such as action deployment and queries for question results.
3. By default, the API Gateway relays user queries to the Tanium Data Service, which continuously collects results for all registered sensors through the Tanium Server. Continuous collection from endpoints ensures that results are available in the Tanium Data Service cache for both online and offline endpoints.
4. Users can optionally specify queries that go directly from the API Gateway to the Tanium Server for collecting live results from endpoints that are currently online.
5. Users can specify requests that go from the API Gateway to Tanium solutions to be performed on endpoints. For example, a user might request Tanium™ Direct Connect to establish a direct connection to a specific endpoint.
6. The Tanium Server issues questions and deploys actions (and associated content such as packages) to Tanium Clients.
7. Tanium Clients return the question results or action statuses to the Tanium Server. The server responds to the request with the results and statuses. If the user sent the request through the API Gateway query explorer, the results pane displays the response.

Query explorer

API Gateway includes an interactive query explorer that you can use to write and run queries and mutations in the Tanium Console. Use the query explorer to try new queries and discover what data is available.

You can find the query explorer on the API Gateway **Overview** page:

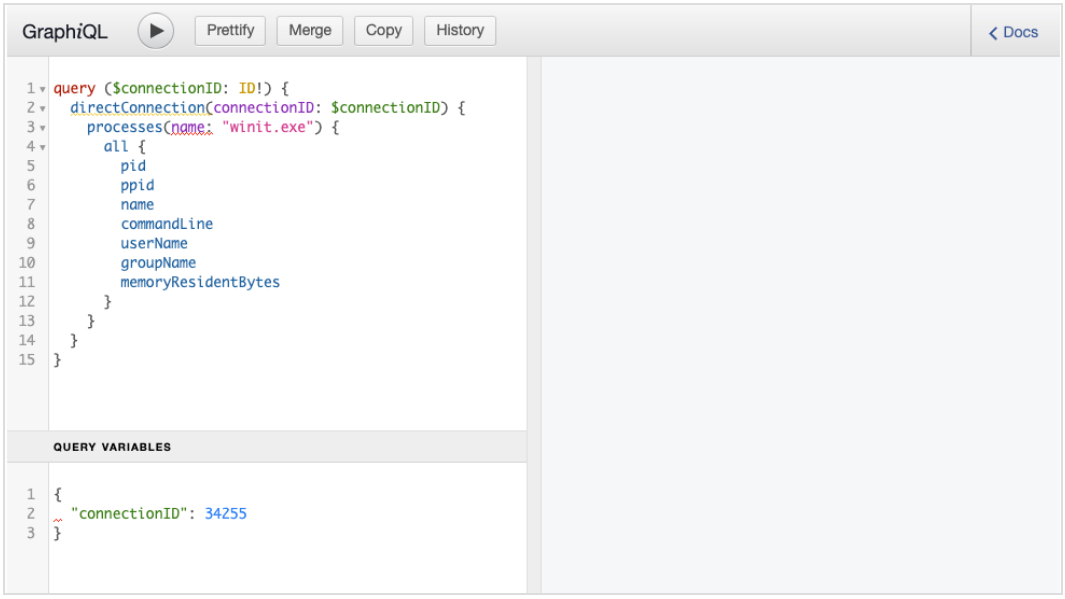


TIP

If the query explorer does not appear on the API Gateway **Overview** page, click **Customize Page** and make sure the **Query Explorer** option is selected.

Query variables

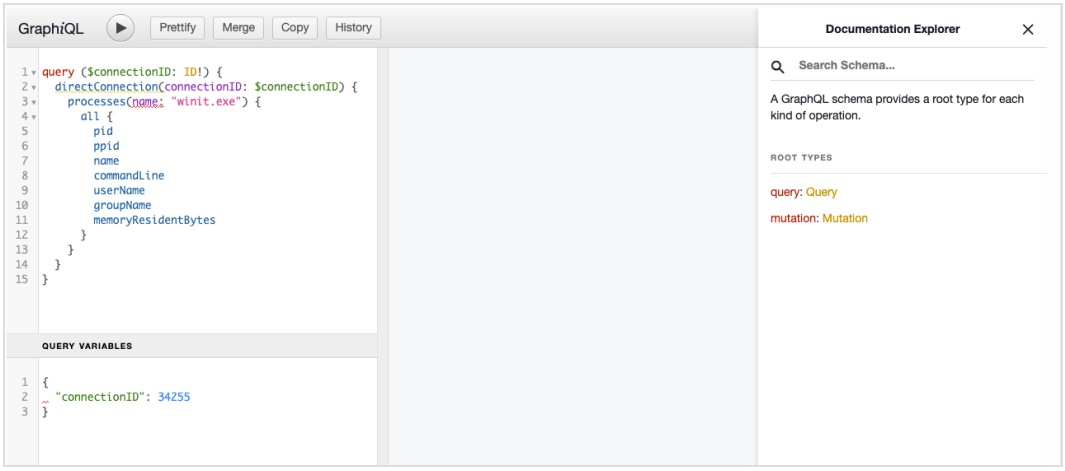
If a query or mutation uses variables, expand the **QUERY VARIABLES** pane and include the variables in the pane that expands.



Schema reference

API Gateway contains a schema reference that documents all queries, mutations, and objects that are available in API Gateway. The schema reference is generated directly from the schema; refer to the schema reference in API Gateway for the most up-to-date documentation.

To view the schema reference in the query explorer, click **Docs** to expand the **Documentation Explorer** pane of the query explorer.



NOTE The query explorer uses the GraphiQL interactive browser to send GraphQL queries and mutations to the Tanium Server. For more information about the options that are available in GraphiQL, see https://graphql.org/learn/.

Authentication

Requests that are sent from the query explorer in the Tanium Console are authenticated and authorized with the session ID of the user who is signed in. The Tanium Server uses the role-based access control (RBAC) permissions of the user account to determine which content you can query and mutate.

Requests that are sent from outside the Tanium Console are authenticated and authorized with either session IDs or API tokens. You must include an API token or session ID in the authorization header of all requests that are sent to API Gateway. API Gateway uses the RBAC permissions of the requesting user to determine content access for all queries and mutations. For an example cURL query that shows the authorization header, see Example cURL syntax on page 11.

BEST PRACTICE Use API tokens to send requests through API Gateway instead of session IDs. While session IDs time out after five minutes of inactivity, you can set a longer timeout for API tokens. You can create API tokens in the Tanium Console or through the Tanium Core Platform REST API. For more information, see Tanium Console User Guide: Managing API tokens.

IMPORTANT When requests are sent outside the Tanium Console, make sure to use the correct URL to send requests. See Root endpoint on page 11 and Example cURL syntax on page 11 for examples.

Rate limits

API Gateway has no specific rate limits.

Root endpoint

To send queries and mutations outside the Tanium Console, use the following address:

`https://<server>/plugin/products/gateway/graphql`

Example cURL syntax

```
curl --request POST \
   --url https://localhost/plugin/products/gateway/graphql \
   --header 'Content-Type: application/json' \
   --header 'session: token-356d5f5bbb3671f28e24f65be3bdd54d9d81001ca823efaabc5fbff251' \
   --data '{"query":"{\n  now\n}\n"}'
```

Pagination

Queries that return many results are paginated to reduce resource utilization. API Gateway uses the standard Relay GraphQL pagination specification to provide users the option to explicitly control pagination.

Paginated queries return a connection type that is prefixed with the name of the data type, such as `EndpointConnection`. Queries accept standard arguments to control the pagination.

Example of a paginated query:

```
{
  endpoints {
    edges {
      node {
        id
        serialNumber
      }
    }
    pageInfo {
      hasNextPage
      endCursor
    }
  }
}
```

Cursors

Use cursors to control relay pagination. Cursors are opaque strings that point to records within queried collections, and can be used to request the records after the cursor. All collections support forward traversal, and some also support backward traversal.

Cursors are valid only for the query for which they were returned. Cursors are generally valid for five minutes after their most recent use. Any queries that deviate from this policy are documented in the query field.

Connection results are stable and consistent when traversed with cursors unless documented in the query field.

Connection and edges

The connection includes an `edge` field that returns a list of typed edges, such as `EndpointEdge`. Each edge contains at least two fields:

- A `node` field with the actual data type, such as `Endpoint`
- A `cursor` field with a cursor for the record

The connection type also includes a `pageInfo` field that contains at least two fields:

- `hasNextPage` indicates if there are more records
- `endCursor` is the cursor of the last record in the returned page, if any

Some connection types feature other metadata, such as `totalRecords`.

Arguments

Paginated queries support at least two arguments: `first` indicates the number of records to return, and `after` is the value of the record cursor that precedes the records in the requested page. When fully paginated, this value is the same as the `endCursor` value from the previous page. Both arguments have sensible defaults.

Paginated queries that support backward traversal allow two corresponding arguments: `last` and `before`.

A single query supports either forward or backward traversal, but not both. The server returns an error response for queries with arguments for both forward and backward traversals.

When a paginated request extends beyond the collection, the query returns only the available results.

Example of a request for a page of data within a collection:

```
{
  endpoints(after: "the-cursor-value", first: 10) {
    edges {
      node {
        id
        serialNumber
      }
    }
    pageInfo {
      hasNextPage
      endCursor
    }
  }
}
```

Filters

Most queries that return multiple results provide support to filter the results. Such queries provide a `filter` argument.

Simple filters

Simple filters are single filters that constrain the values of fields that participate in the query. You can specify simple filters in the `path` property with a period to separate levels in the graph starting at the record type. For example:

```
{
  endpoints(filter: {path: "primaryUser.email", value: "user@example.com"}) {
    edges {
      node {
        id
        primaryUser {
          email
        }
      }
    }
  }
}
```

The query does not need to return the filtered path. Not all field paths are filterable. Refer to the schema to see which paths cannot be filtered.

Simple filters must also contain a string `value` property.

You can specify an operator in the `op` property, which is an enumerated type and defaults to the equality operator. For example:

```
{
  endpoints(filter: {path: "processor.logicalProcessors", value: "4", op: GTE}) {
    edges {
      node {
        id
      }
    }
  }
}
```

Not all operators are valid for all fields.

Compound filters

Compound filters contain multiple simple or compound filters that appear in the `filters` property. By default, all child filters must pass for a record to be included. If the `or` argument is given with a `true` value, then a record is included if any child filter matches.

Example of a simple compound filter:

```
{
  endpoints(filter: {filters: [{path: "serialNumber" value: "x"}, {path: "name", value: "y"}]}) {
    edges {
      node {
        id
      }
    }
  }
}
```

Negated filters

You can negate both simple and compound filters with a `negated` property of `true`. For example, the following query returns endpoints whose serial number does not contain the letter `x`:

```
{
  endpoints(filter: {path: "serialNumber", value: "x", negated: true}) {
    edges {
      node {
        id
      }
    }
  }
}
```

Field filters

Filters apply to the entire record. Some records contain fields that are collections; you can also filter these fields. When you filter a field, the filter applies to both the child collection and to the records. For example, if you search for endpoints with an installed application named `Tanium Client` with a filter on the field, API Gateway returns only those endpoints with such an application, as well as only the matching application:

```
{
  endpoints {
    edges {
      node {
        installedApplications(filter: {path: "name", value: "Tanium Client"}) {
          name
          version
        }
      }
    }
  }
}
```

Field filters can be simple or compound. Compound filters are limited to one level of children that must use the equality operator, and require all child filters. For example:

```
{
  endpoints {
    edges {
      node {
        installedApplications(
          filter: {
            filters: [
              {path: "name", value: "Tanium Client"},
              {path: "version": value: "7.5.0.0"}
            ]
          }
        ) {
          name
          version
        }
      }
    }
  }
}
```

For more information on specific filter syntax, see Reference: Filter syntax on page 38.

Integration with other Tanium products

The following solutions are supported by API Gateway:

- Tanium Core Platform
    - Actions
    - Tanium™ Data Service
    - Tanium™ Direct Connect
    - Packages
- Tanium™ Blob
- Tanium™ Deploy
- Tanium™ Performance

Getting started with API Gateway

Step 1: Review the requirements

Review the system, network, security, and user role requirements: see API Gateway requirements on page 19.

Step 2: Install API Gateway

See Installing API Gateway on page 23.

Step 3: Install any integrated solutions that use the API Gateway

Import any integrated solutions that you want to use. For information on which Tanium solutions use the API Gateway, see Integration with other Tanium products on page 17.

Step 4: Grant API Gateway permissions

Grant permissions to users to use API Gateway. Uses with the **Administrator** reserved role have access by default. See User role requirements on page 21.

Step 5: Test queries through the Tanium™ Console

Use the interactive query explorer to test queries in the Tanium Console. See Test a query in the Tanium Console on page 25.

Step 6: (Optional) Test queries through cURL

Test queries through cURL. See Example cURL syntax on page 11.

Step 7: Explore sample queries and mutations

Explore sample queries and mutations to see what you can do with API Gateway. See Reference: API Gateway examples on page 43.

API Gateway requirements

Review the requirements before you install and use API Gateway.

Core platform dependencies

Make sure that your environment meets the following requirements:

- **Tanium™ Core Platform servers**: 7.4.4 or later
- **Tanium™ Client**: No client requirements.
- **Tanium™ Console**: 2.0 or later
- **Tanium content**: API Gateway uses sensors that are included in the Core Content and Core AD Query content packs.

Solution dependencies

Other Tanium solutions are required for API Gateway to function (required dependencies) or for specific API Gateway features to work (feature-specific dependencies). The installation method that you select determines if the Tanium Server automatically imports dependencies or if you must manually import them.

Some API Gateway dependencies have their own dependencies, which you can see by clicking the links in the lists of Required dependencies on page 19 and Feature-specific dependencies on page 20. Note that the links open the user guides for the latest version of each solution, not necessarily the minimum version that API Gateway requires.

Tanium recommended installation

If you select **Tanium Recommended Installation** when you import API Gateway, the Tanium Server automatically imports all your licensed solutions at the same time. See Tanium Console User Guide: Import all modules and services.

Import specific solutions

If you select only API Gateway to import and are using Tanium Core Platform 7.5.2.3531 with Tanium Console 3.0.72 or later, the Tanium Server automatically imports the latest available versions of any required dependencies that are missing. If some required dependencies are already imported but their versions are earlier than the minimum required for API Gateway, the server automatically updates those dependencies to the latest available versions.

If you select only API Gateway to import and you are using Tanium Core Platform 7.5.2.3503 or earlier with Tanium Console 3.0.64 or earlier, you must manually import or update required dependencies. See Tanium Console User Guide: Import, re-import, or update specific solutions.

Required dependencies

API Gateway has the following required dependencies at the specified minimum versions:

- Tanium Interact 2.9.83 or later
- Tanium System User 1.0.40 or later

Feature-specific dependencies

If you select only API Gateway to import, you must manually import or update its feature-specific dependencies regardless of the Tanium Console or Tanium Core Platform versions. API Gateway has the following feature-specific dependencies at the specified minimum versions:

- Tanium Blob 1.0.6 or later
- Tanium Direct Connect 1.10.39 or later
- Tanium Deploy 2.9.123 or later
- Tanium Performance 1.10.57 or later

Tanium™ Module Server

API Gateway is installed and runs as a service on the Module Server host computer. The impact on the Module Server is minimal and depends on usage.

For information about Module Server sizing in a Windows deployment, see Tanium Core Platform Deployment Guide for Windows: Host system sizing guidelines.

Endpoints

API Gateway does not directly deploy packages to endpoints. However, you can use API Gateway to deploy packages through Tanium Deploy. For Tanium Deploy endpoint requirements, see Tanium Deploy User Guide: Endpoints.

For Tanium Client operating system support, see Tanium Client Management User Guide: Client version and host system requirements.

Host and network security requirements

Specific ports and processes are needed to run API Gateway.

Ports

The following ports are required for API Gateway communication.

Source	Destination	Port	Protocol	Purpose
Module Server	Module Server (loopback)	17600	TCP	Internal purposes, not externally accessible

BEST PRACTICE Configure firewall policies to open ports for Tanium traffic with TCP-based rules instead of application identity-based rules. For example, on a Palo Alto Networks firewall, configure the rules with service objects or service groups instead of application objects or application groups.

Security exclusions

If security software is in use in the environment to monitor and block unknown host system processes, Tanium recommends that a security administrator create exclusions to allow the Tanium processes to run without interference. The configuration of these exclusions varies depending on AV software. For a list of all security exclusions to define across Tanium, see Tanium Core Platform Deployment Reference Guide: Host system security exclusions.

API Gateway security exclusions

Target Device	Notes	Exclusion Type	Exclusion
Module Server		Process	`<Module Server>\services\gateway-service\TaniumGatewayService.exe`

User role requirements

The following tables list the role permissions required to use API Gateway. For more information about role permissions and associated content sets, see Tanium Console User Guide: Managing RBAC.

API Gateway user role permissions

Permission	API Gateway User[1]	Gateway Service Account	Gateway Service Account - All Content Sets
Gateway Api Access API Gateway	✓ EXECUTE	✗	✗
Gateway Service Account Provides access for the API Gateway service.	✗	✓ EXECUTE	✗

[1] This role provides module permissions for Tanium Interact. You can view which Interact permissions are granted to this role in the Tanium Console. For more information, see Tanium Interact User Guide: User role requirements.

Provided API Gateway administration and platform content permissions

Permission	Permission Type	API Gateway User	Gateway Service Account	Gateway Service Account - All Content Sets
Action Group	Administration	✗	✓ READ WRITE	✗
Computer Group	Administration	✗	✓ READ	✗
Global Settings	Administration	✗	✓ READ	✗
Sensor	Platform Content	✗	✗	✓ READ[1]
Token - Use	Administration	✓ SPECIAL	✗	✗
Plugin	Platform Content	✓ EXECUTE[2] READ[2]	✗	✗

[1] This permission applies to all content sets.

[2] This permission applies to the Interact content set.

Installing API Gateway

Use the Tanium Console **Solutions** page to install API Gateway and choose either automatic or manual configuration:

- **Automatic configuration** (Tanium Core Platform 7.4.2 or later only): API Gateway is installed with any required dependencies and other selected products. This option is the best practice for most deployments. For more information about the automatic configuration for API Gateway, see Tanium Console User Guide: Import all modules and services.
- **Manual configuration**: Manually install API Gateway and the required dependencies. For more information, see Import API Gateway on page 23.

Before you begin

- Read the release notes.
- Review the API Gateway requirements on page 19.
- If you are upgrading from a previous version, see Upgrade API Gateway on page 24.
- Assign the correct roles to users for API Gateway. Review the User role requirements on page 21.
    - To import the API Gateway solution, you must be assigned the **Administrator** reserved role.
- If you install API Gateway in an All-in-One deployment, update the **api_token_trusted_ip_address_list** platform setting. For more information, see Update platform setting for All-in-One deployment on page 27.

Import API Gateway

Perform the following steps to install the API Gateway solution on the Tanium Server.

If you have multiple Tanium Servers in an active-active configuration, you only need to perform these steps on one Tanium Server if you have Tanium Core Platform 7.4.3.1204 or later.

1. Sign in to the Tanium Console with an account that has the **Administrator** reserved role.
2. From the Main menu, go to **Administration > Configuration > Solutions**.
3. In the **Content** section, select the checkbox for **API Gateway** and click **Install**.

   If you need to install any prerequisite Tanium solutions or content, select the corresponding checkboxes for those solutions as well.
4. Review the content to import and click **Begin Install**.

Manage solution dependencies

Other Tanium solutions are required for API Gateway to function (required dependencies) or for specific API Gateway features to work (feature-specific dependencies). See Solution dependencies.

Upgrade API Gateway

For the steps to upgrade API Gateway, see Tanium Console User Guide: Import, re-import, or update specific solutions. After the upgrade, verify that the correct version is installed: see Verify API Gateway version on page 24.

Verify API Gateway version

After you import or upgrade API Gateway, verify that the correct version is installed:

1. Refresh your browser.
2. From the Main menu, go to **Administration > Shared Services > API Gateway** to open the API Gateway **Overview** page.
3. To display version information, click Info  .

Troubleshoot issues

If you experience issues with installing API Gateway, see Queries return unexpected results or errors on page 27.

Using API Gateway

Use API Gateway to build API-based integrations with the Tanium Core Platform. This service consolidates information from multiple Tanium modules into a unified view of information on the endpoints in the environment. API Gateway intelligently routes requests to the services and sources that provide the most recent information and the most reliable mutations.

API Gateway uses GraphQL to request data (queries) and to make changes (mutations). With GraphQL, you can compose queries in API Gateway to retrieve the exact data that you want as well as filter the results to a set of endpoints.

Use API Gateway to:

- Query endpoints through the Tanium Server, or access data through Tanium Data Service
- Create, delete, and query actions
- Query packages
- Open a connection to an endpoint through Tanium Direct Connect and retrieve data from the endpoint

For examples of available functions, see Reference: API Gateway examples on page 43.

Test a query in the Tanium Console

To access the query explorer in the Tanium Console and run a query, perform the following steps:

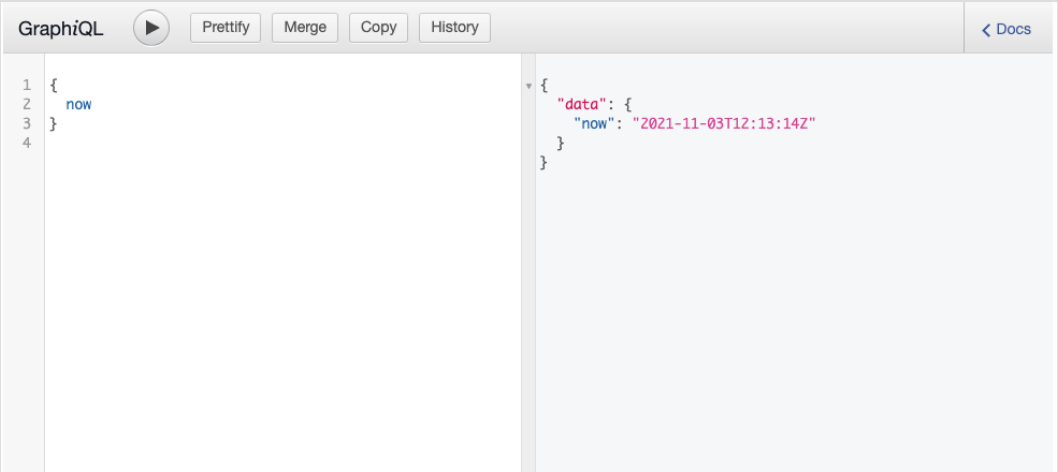
1. From the Main menu, go to **Administration > Shared Services > API Gateway**.
2. Enter a query in the query pane. For example, paste the following query to get the time from the Tanium Server:

```
{
  now
}
```


If the query explorer does not appear on the API Gateway **Overview** page, click **Customize Page** and make sure the **Query Explorer** option is selected.

3. (Optional) If a query or mutation uses variables, expand the **QUERY VARIABLES** pane and include the variables in the pane that expands.
4. Click Execute Query  .

API Gateway sends the query to the server and returns the response in the results pane.



For more information on the query explorer, see Query explorer on page 8.

Troubleshooting API Gateway

If API Gateway is not performing as expected, you might need to troubleshoot issues.

Collect logs

The information is saved as a ZIP file that you can download with your browser.

1. From the API Gateway **Overview** page, click Help, then the **Troubleshooting** tab.
2. Click **Download Support Package**.
   A `tanium-api-gateway-support-[timestamp].zip` file downloads to the local download directory.
3. Contact Tanium Support to determine the best option to send the ZIP file. For more information, see Contact Tanium Support on page 37.

Tanium API Gateway maintains logging information in the `gateway-service.log` file in the `\Program Files\Tanium\Tanium Module Server\services\gateway-files\logs\` directory.

Update platform setting for All-in-One deployment

For All-in-One deployments, before you install System User Service as a required dependency for API Gateway, you must first add the `127.0.0.1` IPv4 address if you enabled IPv4, or the `::1` IPv6 address if you enabled IPv6, to the **api_token_trusted_ip_address_list** platform Server setting. Otherwise, the installation process does not complete.

All-in-One deployments are supported only for proof-of-concept (POC) demonstrations.

1. From the Main menu, go to **Administration > Configuration > Platform Settings**.
2. In the **Name** column, click **api_token_trusted_ip_address_list**.
3. You have the following options:
   - If you enabled IPv4, enter `127.0.0.1` in the **Value** field.
   - If you enabled IPv6, enter `::1` in the **Value** field.
4. Click **Save**.

Queries return unexpected results or errors

- The API Gateway service redirects queries and mutations to other Tanium solutions. If API Gateway returns unexpected results or errors, make sure that all prerequisites are installed at the minimum recommended version. For information, see API Gateway requirements on page 19.

- If all queries and mutations return a **502 Gateway Timeout** error, make sure the Tanium System User service and the Tanium API Gateway service are running on the Tanium Module Server.
- If you recently installed API Gateway or the System User service, restart the Tanium Module Server.
- Queries and mutations that use the **eid** element require Interact 2.9 or later.

Request error handling

When requests cannot be fully satisfied, the gateway service gives the caller accurate, practical error information. The gateway service honors the GraphQL error specification with some custom extensions for greater specificity.

Syntax Validation

Before processing any portion of a request, the service parses its syntax. If the request is not valid GraphQL, or if it does not conform to the schema, the service returns a response with the syntax errors, including line and column positions whenever possible.

Examples of syntax errors include:

- Unbalanced parentheses, braces, brackets, and quotation marks
- References to fields that do not exist

The GraphQL error specification requires reporting these errors to the caller by line and column in the request.

When possible, the Query pane in the Query Explorer identifies syntax errors in the request.

Structural Validation

If the request passes syntax validation, the service evaluates the request for structural validity, which checks for errors not expressible with the GraphQL type system. Structural error examples include:

- Restrictions on type values, such as:
    - A string may not be empty
    - An integer must be positive
- Constraints on input objects, such as:
    - Exactly one of two or more entries must be present
- Existence of values in a known set, such as:
    - A named sensor must exist
- References to fields that cannot be satisfied in the current environment, such as:
    - A sensor does not exist

The GraphQL `extensions` error field for custom fields contains these errors.

Error Extensions

Errors that pertain to a specific field in the request are identified by a `fieldPath` entry in the `extensions` object. Similar to the `path` entry on the GraphQL error type, the `fieldPath` consists of a list of strings and integers which indicate keys in objects and offsets in lists, respectively. The root of the `fieldPath` is the request field that triggered the GraphQL error.

> NOTE
>
> The numbering for list items starts at `0` for the first item and increments for each additional item. For example, if a `fieldPath` contains `5`, this refers to the sixth item in the list.

The `extensions` object may also have these fields:

Extensions field	Description
`code`	an error code attributable to the field itself
`context`	a map of contextual information that refines or specifies the error
`message`	a short error explanation

For example, with the following syntactically valid request:

```
{
  endpoints {
    edges {
      node {
        name
        serialNumber
      }
    }
  }
}
```

If the serial number is not available in the underlying data store, the gateway service returns a response including an extended error that the value does not exist:

```
{
  "errors": [
    {
      "message": "Invalid request",
```

```
      "path": ["endpoints"],
      "extensions": {
        "fieldPath": ["edges", "node", "serialNumber"],
        "code": "validation_exists",
        "message": "must refer to an existing sensor",
        "context": {
          "source": "tds"
        }
      }
    }
  ]
}
```

Invalid argument errors contain an extended `argumentErrors` entry. The value of that entry is a list of objects that may have the following entries:

Extensions field	Description
`code`	an error code attributable to the field itself
`context`	a map of contextual information that refines or specifies the error
`path`	the path to the argument error, relative to the field
`message`	a short error explanation

For example, with the following request that does not define a `name` or `path` value:

```
{
  endpoints(filter: {filters: [{memberOf: {name: ""}}, {path:""}]}) {
    edges {
      node {
        name
      }
    }
  }
}
```

The gateway service returns a response including both errors stating a value is a required:

```
{
  "errors": [
    {
      "message": "Invalid request",
      "path": ["endpoints"],
      "extensions": {
        "argumentErrors": [
          {
            "code": "validation_required",
            "path": ["filter", "filters", 0, "memberOf", "name"]
          },
          {
            "code": "validation_required",
            "path": ["filter", "filters", 1, "path"]
          }
        ]
      }
    }
  ]
}
```

Extended errors may identify argument errors within sub-fields. For example, the following request specifies a non-existent sensor name:

```
{
  endpoints {
    edges {
      node {
        sensorReadings(sensors:[{name: "A Non-Existent Sensor"}]) {
          columns {
            values
          }
        }
      }
```

```
    }
  }
}
```

The gateway service returns a response including the sub-field error that the value does not exist:

```
{
  "errors": [
    {
      "message": "Invalid request",
      "path": ["endpoints"],
      "extensions": {
        "fieldPath": ["edges", "node", "sensorReadings"],
        "argumentErrors": [
          {
            "code": "validation_exists",
            "path": ["sensors", 0, "name"]
          }
        ]
      }
    }
  ]
}
```

Whenever possible, the response includes the valid portions of a query. For example, the following request references a valid sensor (Custom Tags) and a not valid sensor (A Non-existent Sensor):

```
{
  endpoints {
    edges {
      node {
        name
        ipAddress
        os {
```

```
          generation
        }
        sensorReadings(sensors: [{name: "Custom Tags"}, {name: "A Non-existent Sensor"}])
{
          columns {
            sensor {
              name
            }
            name
            values
          }
        }
      }
    }
  }
}
```

The response includes the error from A Non-Existent Sensor not being found, and all information that can otherwise be returned from the Custom Tags sensor:

```
{
  "errors": [
    {
      "message": "sensors: (1: (name: must refer to an existing sensor.).).",
      "path": [
        "endpoints"
      ],
      "extensions": {
        "argumentErrors": [
          {
            "code": "validation_exists",
            "context": {
              "source": "tds"
            },
```

```
            "message": "must refer to an existing sensor",
            "path": [
              "sensors",
              1,
              "name"
            ]
          }
        ],
        "fieldPath": [
          "edges",
          "node",
          "sensorReadings"
        ]
      }
    }
  ],
  "data": {
    "endpoints": {
      "edges": [
        {
          "node": {
            "name": "example-host",
            "ipAddress": "192.0.2.10",
            "os": {
              "generation": "Windows 10"
            },
            "sensorReadings": {
              "columns": [
                {
                  "sensor": {
                    "name": "Custom Tags"
                  },
                  "name": "Custom Tags",
                  "values": [
```

```
                    "custom-tag-1",
                    "custom-tag-2"
                  ]
                }
              ]
            }
          }
        ]
      }
    }
}
```

NOTE

Similar to lists in the `fieldPath` referring to offsets in lists, if an error occurs on an item in a list, the error `message` might refer to the list offset instead of the field name. In the above response, the `message` refers to item `1` in the list; because list numbering starts at `0`, this points to the second sensor (A Non-existent Sensor).

Error codes

General error codes include:

Error code	Description
`502 Bad Gateway`	The gateway service received an invalid response from the server.
`internal_server_error`	The response field cannot be resolved due to a server error.

Syntax error codes include:

Error code	Description
`GRAPHQL_PARSE_FAILED`	The request contains unbalanced parentheses, braces, brackets, or quotation marks. See the `message` for more information.
`GRAPHQL_VALIDATION_FAILED`	The request contains an unrecognized argument, or a required argument is not included. See the `message` for more information.

Structural error codes include:

Error code	Description
`validation_exists`	The field must refer to an existing entity.
`validation_in_invalid`	The field must have a value from a specific set.
`validation_invalid_use`	The filter is not valid in its context.
`validation_key_unexpected`	The field is not allowed.
`validation_min_greater_equal_than_required`	The field must have a value greater than or equal to its documented minimum.
`validation_nil`	The field must be absent.
`validation_nil_or_not_empty_required`	The field must be absent or have a non-empty value.
`validation_not_nil_required`	The field must have a value.
`validation_overdetermined`	The field's object must not have inconsistent entries.
`validation_required`	The field must have a non-empty value.
`validation_timestamp`	The field's value must be a valid RFC 3339 string.
`validation_unique`	The field's value must be unique in its documented container.

Uninstall API Gateway

If you need to uninstall API Gateway, perform the following steps.

Consult with Tanium Support before you uninstall or reinstall API Gateway.

1. Sign in to the Tanium Console as a user with the Administrator role.
2. From the Main menu, go to **Administration > Configuration > Solutions**.
3. In the **Content** section, select the **API Gateway** row and click **Uninstall**.
4. Review the summary and click **Yes** to proceed with the uninstallation.
5. When prompted to confirm, enter your password.

The uninstall does not remove the API Gateway log from the Tanium Module Server. To remove the log after the uninstall completes, manually delete the `\Program Files\Tanium\Tanium Module Server\services\gateway-files\` directory.

Contact Tanium Support

To contact Tanium Support for help, sign in to https://support.tanium.com.

Reference: Filter syntax

Endpoint query filter syntax

Filter endpoint query requests using the following filter syntax.

General Fields

You can filter on field values. Unless otherwise specified, define a filter object using the syntax:

```
{path: "field-path", value: "field-value"}
```

Use dot notation for sub-fields not at the root level.

For example, the following request filters and returns only endpoints whose primary user email address is `user@example.com`:

```
{
  endpoints(filter: {path: "primaryUser.email", value: "user@example.com"}) {
    edges {
      node {
        id
        primaryUser {
          email
        }
      }
    }
  }
}
```

Computer group membership

You can filter on membership in named computer groups. Define a filter object using the syntax:

```
{memberOf: {name: "computer-group-name"}}
```

For example, the following request filters and returns only endpoints that are members of the All Linux computer group:

```
{
  endpoints(filter: {memberOf: {name: "All Linux"}}) {
    edges {
      node {
        id
      }
    }
  }
}
```

Sensors

Endpoint queries can filter on readings of arbitrary named sensors. Define a filter object using the syntax:

```
{sensor: {name: "sensor-name", value: "filter-value"}}
```

For example, the following request filters based on the Total Memory sensor returning a value of `16384 MB` for endpoints, and returns only those endpoints:

```
{
  endpoints(filter: {sensor: {name: "Total Memory"}, value: "16384 MB"}) {
    edges {
      node {
        id
      }
    }
  }
}
```

Parameterized sensors

You can also filter on parameterized sensors. Define a filter object using the syntax:

```
{sensor:
  {name: "sensor-name",
  params:[
    {name: "parameter-1-name", value: "parameter-1-value"},
```

```
    {name: "parameter-2-name", value: "parameter-2-value"},
    {name: "parameter-n-name", value: "parameter-n-value"}
  ]
},
  value: "filter-value"
}
```

Add a `params` `name`/`value` object for every parameter.

For example, the following request filters based on the Custom Tag Exists sensor, returning endpoints tagged with the custom tag `the-tag`, and returns only those endpoints:

```
{
  endpoints(filter: {sensor: {name: "Custom Tag Exists", params:[{name: "tag", value: "the-tag"}]}, value: "true"}) {
    edges {
      node {
        id
      }
    }
  }
}
```

Multi-column sensors

You can filter multi-column sensors by specifying the column name. Define a filter object using the syntax:

```
{sensor: {
  name: "sensor-name",
  column: "sensor-column-name"},
  value: "filter-value"}
}
```

For example, the following request filters based on the CPU Details sensor returning endpoints with an `Intel` CPU, and returns only those endpoints:

```
{
  endpoints(filter: {sensor: {name: "CPU Details", column: "CPU"}, value: "Intel"}) {
```

```
    edges {
      node {
        id
      }
    }
  }
}
```

If you want to select across multiple columns of each row of a multi-column sensor, define a `filter.filters` list, then define a list element for each column filter, using the following syntax:

```
filters: [
  {sensor: {column: "column-1-name"}, value: "column-1-value"},
  {sensor: {column: "column-2-name"}, value: "column-2-value"},
  {sensor: {column: "column-n-name"}, value: "column-n-value"}
]
```

You can only use a single level of child filters, and can only use the default `EQ` operator.

For example, the following request filters based on the CPU Details sensor returning endpoints with `4` total cores and an `Intel` CPU, and returns only those endpoints:

```
{
  endpoints(filter: {
    sensor: {name: "CPU Details"},
    filters: [
      {sensor: {column: "Total Cores"}, value: "4"},
      {sensor: {column: "CPU"}, value: "Intel"}
    ]
  }) {
    edges {
      node {
        id
      }
    }
  }
}
```

In this form, the sensor is identified in the top filter and the columns are identified in the child filters. You can only use a single level of child filters, and can only use the default `EQ` operator.

Sensor readings

After you filter endpoints, you can filter your results to return data from additional sensors. As a child of the node object, define the sensor reading filter using the following syntax:

```
sensorReadings
  (sensors: [{name: "sensor-name"}]) {
    columns {
      column-1
      column-2
      column-n
    }
```

For example, the following request first filters endpoints and only returns endpoints with `Firefox` as an installed application, then for each endpoint, returns the results of the Is Linux sensor:

```
{
  endpoints(
    filter: {sensor: {name: "Installed Applications" column:"Name"}, op: CONTAINS, value:
"firefox"}
  ) {
    edges {
      node {
        name
        ipAddress
        sensorReadings(sensors: [{name: "Is Linux"}]) {
          columns {
            sensor {name}
            name
            values
          }
        }
      }
    }
  }
}
```

Reference: API Gateway examples

Use the following query examples to learn about the functionality and syntax of queries and mutations in API Gateway.

General examples

The following queries retrieve data from the endpoints in your environment.

Get server time

The following query retrieves the local time from the Tanium Server.

```
{
  now
}
```

Example response:

```
{
  "data": {
    "now": "2021-11-08T19:22:03Z"
  }
}
```

Get endpoints

The following query retrieves known endpoints from the Tanium Server.

```
{
  endpoints(source: {ts: {expectedCount: 1, stableWaitTime: 10}}) {
```

```
    edges {
      node {
        computerID
        name
        serialNumber
        ipAddress
      }
    }
  }
}
```

Example response:


```
{
  "data": {
    "endpoints": {
      "edges": [
        {
          "node": {
            "computerID": "937672696",
            "name": "ubuntu-test",
            "serialNumber": "Not Specified",
            "ipAddress": "10.168.20.30"
          }
        },
        {
          "node": {
            "computerID": "1867570226",
            "name": "CentOS-test-1",
            "serialNumber": "Not Specified",
            "ipAddress": "10.168.20.40"
          }
        },
        {
```

```
            "node": {
              "computerID": "2711217959",
              "name": "CentOS-test-2",
              "serialNumber": "Not Specified",
              "ipAddress": "10.168.20.50"
            }
          }
        ]
      }
    }
}
```

Get endpoints IDs from Tanium Data Service

The following query retrieves all endpoint IDs from Tanium Data Service.

```
{
  endpoints {
    edges {
      node {
        id
      }
    }
  }
}
```

Example response:

```
{
  "data": {
    "endpoints": {
      "edges": [
        {
          "node": {
```

```
            "id": "12345"
          }
        },
        {
          "node": {
            "id": "54321"
          }
        },
        {
          "node": {
            "id": "21212"
          }
        }
      ]
    }
  }
}
```

Get endpoints using aliases

The following query retrieves known endpoints using aliases to rename the information returned and prevent data collision due to overlapping field names. The `tds` alias retrieves the `id` and `name` of known endpoints from Tanium Data Service. The `ts` alias retrieves the IPv6 address `value` of known endpoints from the Tanium Server and the sensor `name`.

```
{
  tds:endpoints {
    edges {
      node {
        id
        name
      }
    }
  }
  ts:endpoints(source: {ts: {stableWaitTime: 10}}) {
```

```
    edges {
      node {
        sensorReadings(sensors: [{name: "IPv6 Address"}]) {
          columns {
            name
            values
          }
        }
      }
    }
  }
}
```

Example response:

```
{
  "data": {
    "tds": {
      "edges": [
        {
          "node": {
            "id": "12345",
            "name": "example-name"
          }
        },
        {
          "node": {
            "id": "54321",
            "name": "example-name-2"
          }
        }
      ]
    },
    "ts": {
```

```
      "edges": [
        {
          "node": {
            "sensorReadings": {
              "columns": [
                {
                  "name": "IPv6 Address",
                  "values": [
                    "2001:db8::3"
                  ]
                }
              ]
            }
          }
        },
        {
          "node": {
            "sensorReadings": {
              "columns": [
                {
                  "name": "IPv6 Address",
                  "values": [
                    "2001:db8::2"
                  ]
                }
              ]
            }
          }
        }
      ]
    }
  }
}
```

Get endpoints using multiple sensors

The following query retrieves known endpoints using multiple sensors. The request specifies the sensor name as one of the fields returned in the request, to identify the sensor that reported the information.

```
{
  endpoints {
    edges {
      node {
        computerID
        sensorReadings(sensors: [{name:"Installed Applications"},{name:"Running Applications"}]) {
          columns {
            name
            values
            sensor {name}
          }
        }
      }
    }
  }
}
```

Example response:

```
{
  "data": {
    "endpoints": {
      "edges": [
        {
          "node": {
            "computerID": "987654321",
            "sensorReadings": {
              "columns": [
                {
                  "name": "Name",
```

```
                    "values": [
                      "Tanium Client 7.4.7.1094"
                    ],
                    "sensor": {
                      "name": "Installed Applications"
                    }
                  },
                  {
                    "name": "Version",
                    "values": [
                      "7.4.7.1094"
                    ],
                    "sensor": {
                      "name": "Installed Applications"
                    }
                  },
                  {
                    "name": "Silent Uninstall String",
                    "values": [
                      "\"C:\\Program Files (x86)\\Tanium\\Tanium Client\\uninst.exe\""
                    ],
                    "sensor": {
                      "name": "Installed Applications"
                    }
                  },
                  {
                    "name": "Uninstallable",
                    "values": [
                      "Is Uninstallable"
                    ],
                    "sensor": {
                      "name": "Installed Applications"
                    }
                  },
```

```
                {
                  "name": "Name",
                  "values": [
                    "Tanium Client"
                  ],
                  "sensor": {
                    "name": "Running Applications"
                  }
                },
                {
                  "name": "Version",
                  "values": [
                    "7.4.7.1094"
                  ],
                  "sensor": {
                    "name": "Running Applications"
                  }
                },
                {
                  "name": "Process Name",
                  "values": [
                    "TaniumClient.exe"
                  ],
                  "sensor": {
                    "name": "Running Applications"
                  }
                }
              ]
            }
          }
        ]
      }
    }
}
```

Get rich endpoint data

The following query demonstrates using nested fields to retrieve categorized endpoint data.

> **NOTE** The `first:2` argument retrieves two records; set this value higher to retrieve more records at a time. For more information on pagination arguments, see Pagination on page 12.

```
{
  endpoints (first:2) {
    edges {
      node {
        name
        computerID
        ipAddress
        isVirtual
        chassisType
        systemUUID
        domainName
        os {
          name
          platform
          generation
        }
        processor {
          architecture
          cacheSize
          consumption
          cpu
          family
          manufacturer
          speed
        }
        lastLoggedInUser
      }
    }
    pageInfo {
```

```
      startCursor
      endCursor
      hasNextPage
    }
  }
}
```

Example response:


```
{
  "data": {
    "endpoints": {
      "edges": [
        {
          "node": {
            "name": "Test-01",
            "computerID": "1234567890",
            "ipAddress": "192.0.2.10",
            "isVirtual": false,
            "chassisType": "TSE-Error: Unknown - dmidecode unavailable",
            "systemUUID": "TSE-Error: Unknown - dmidecode unavailable",
            "domainName": "(none)",
            "os": {
              "name": "Red Hat Enterprise Linux Server release 5.11 (Tikanga)",
              "platform": "Linux",
              "generation": "Red Hat Enterprise Linux 5"
            },
            "processor": {
              "architecture": "x86_64",
              "cacheSize": "16384 KB",
              "consumption": "9.9 %",
              "cpu": "Intel Core Processor (Haswell, no TSX, IBRS)",
              "family": "6",
              "manufacturer": "GenuineIntel",
```

```
            "speed": "2600 Mhz"
          },
          "lastLoggedInUser": "reboot"
        }
      },
      {
        "node": {
          "name": "Test-02",
          "computerID": "3216549870",
          "ipAddress": "192.0.2.20",
          "isVirtual": true,
          "chassisType": "Virtual",
          "systemUUID": "[no results]",
          "domainName": "(none)",
          "os": {
            "name": "CentOS Linux release 8.4.2105",
            "platform": "Linux",
            "generation": "CentOS 8"
          },
          "processor": {
            "architecture": "x86_64",
            "cacheSize": "35840 KB",
            "consumption": "18.6 %",
            "cpu": "Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz",
            "family": "6",
            "manufacturer": "GenuineIntel",
            "speed": "2400 Mhz"
          },
          "lastLoggedInUser": "tester-5"
        }
      }
    ],
    "pageInfo": {
      "startCursor": "4267468:0",
```

```
        "endCursor": "4267468:1",
        "hasNextPage": true
      }
    }
  }
}
```

Get a set of endpoints

The following query retrieves a set of endpoints. The query demonstrates the use of the `sensorReadings` field and contains a filter argument to retrieve endpoints whose names contain the letter `a`. The results are paginated to 3 records.

```
{
  endpoints(first: 3, filter: {op: MATCHES, path: "name", value: "a.*"}) {
    edges {
      node {
        name
        ipAddress
        sensorReadings(sensors: [{name: "EID Last Seen"}]) {
          columns {
            name
            values
          }
        }
      }
    }
  }
}
```

Example response:

```
{
  "data": {
    "endpoints": {
```

```
    "edges": [
      {
        "node": {
          "name": "test-1",
          "ipAddress": "192.0.2.10",
          "sensorReadings": {
            "columns": [
              {
                "name": "EID Last Seen",
                "values": [
                  "Mon, 08 Nov 2021 21:29:28 +0000"
                ]
              }
            ]
          }
        }
      },
      {
        "node": {
          "name": "test-2",
          "ipAddress": "192.0.2.20",
          "sensorReadings": {
            "columns": [
              {
                "name": "EID Last Seen",
                "values": [
                  "Mon, 08 Nov 2021 21:29:28 +0000"
                ]
              }
            ]
          }
        }
      },
      {
```

```
            "node": {
              "name": "test-3",
              "ipAddress": "192.0.2.30",
              "sensorReadings": {
                "columns": [
                  {
                    "name": "EID Last Seen",
                    "values": [
                      "Mon, 08 Nov 2021 21:17:17 +0000"
                    ]
                  }
                ]
              }
            }
          }
        ]
      }
    }
}
```

Unregistered sensor query

The following query retrieves the operating system platform from all endpoints.

> **NOTE** In API Gateway, a sensor is unregistered if the sensor is not represented by a named field in the API Gateway schema. This has no correlation to registering sensors in Tanium Data Service.

```
{
  endpoints {
    edges {
      node {
        id
        name
        sensorReadings(sensors: [{name: "OS Platform"}]) {
```

```
            columns {
              name
              values
            }
          }
        }
      }
    }
}
```

Example response:

```
{
  "data": {
    "endpoints": {
      "edges": [
        {
          "node": {
            "id": "12345",
            "name": "Test-01",
            "sensorReadings": {
              "columns": [
                {
                  "name": "OS Platform",
                  "values": [
                    "Linux"
                  ]
                }
              ]
            }
          }
        },
        {
          "node": {
```

```
            "id": "54321",
            "name": "Test-03",
            "sensorReadings": {
              "columns": [
                {
                  "name": "OS Platform",
                  "values": [
                    "Linux"
                  ]
                }
              ]
            }
          }
        ]
      }
    }
}
```

Unregistered parameterized sensor query

The following query checks to see if each endpoint contains the `C:\Windows\py.exe` file.

> **NOTE** In API Gateway, a sensor is unregistered if the sensor is not represented by a named field in the API Gateway schema. This has no correlation to registering sensors in Tanium Data Service.

```
{
  endpoints(source: {ts: {}}) {
    edges {
      node {
        name
        id
        sensorReadings(
```

```
          sensors: [{name: "File Exists", params: [{name: "file", value:
"C:\\Windows\\py.exe"}]}]
        ) {
          columns {
            sensor {
              name
              params {
                name
                value
              }
            }
            values
          }
        }
      }
    }
  }
}
```

Example response:

```
{
  "data": {
    "endpoints": {
      "edges": [
        {
          "node": {
            "name": "Test-01",
            "id": "12345",
            "sensorReadings": {
              "columns": [
                {
                  "sensor": {
                    "name": "File Exists",
```

```
                    "params": [
                      {
                        "name": "file",
                        "value": "C:\\Windows\\py.exe"
                      }
                    ]
                  },
                  "values": [
                    "File does not exist"
                  ]
                }
              ]
            }
          }
        },
        {
          "node": {
            "name": "[no results]",
            "id": "54225",
            "sensorReadings": {
              "columns": [
                {
                  "sensor": {
                    "name": "File Exists",
                    "params": [
                      {
                        "name": "file",
                        "value": "C:\\Windows\\py.exe"
                      }
                    ]
                  },
                  "values": [
                    "[no results]"
                  ]
```

```
                }
              ]
            }
          }
        },
        {
          "node": {
            "name": "[no results]",
            "id": "65456",
            "sensorReadings": {
              "columns": [
                {
                  "sensor": {
                    "name": "File Exists",
                    "params": [
                      {
                        "name": "file",
                        "value": "C:\\Windows\\py.exe"
                      }
                    ]
                  },
                  "values": [
                    "[no results]"
                  ]
                }
              ]
            }
          }
        }
      ]
    }
  }
}
```

Endpoint cursor query

The following query retrieves cursors for endpoint records. You can use these cursors to retrieve specific endpoint records on a page. For more information on cursors, see Pagination on page 12.

```
{
  endpoints {
    pageInfo {
      hasPreviousPage
      hasNextPage
      startCursor
      endCursor
    }
  }
}
```

Example response:

```
{
  "data": {
    "endpoints": {
      "pageInfo": {
        "hasPreviousPage": false,
        "hasNextPage": true,
        "startCursor": "4277520:0",
        "endCursor": "4277520:19"
      }
    }
  }
}
```

Paginated query

The following query retrieves the first five endpoint records after the given cursor. For more information on cursors, see Pagination on page 12.

```
{
  endpoints(after: "4277520:4", first: 5) {
    edges {
      node {
        name
        id
        ipAddress
      }
    }
    pageInfo {
      hasNextPage
      startCursor
      endCursor
    }
  }
}
```

Example response:

```
{
  "data": {
    "endpoints": {
      "edges": [
        {
          "node": {
            "name": "Test-06",
            "id": "6172",
            "ipAddress": "192.0.2.10"
          }
        },
        {
          "node": {
            "name": "Test-07",
            "id": "87654",
```

```
            "ipAddress": "192.0.2.20"
          }
        },
        {
          "node": {
            "name": "Test-14",
            "id": "43584",
            "ipAddress": "192.0.2.30"
          }
        },
        {
          "node": {
            "name": "Test-03",
            "id": "37233",
            "ipAddress": "[no results]"
          }
        },
        {
          "node": {
            "name": "Test-55",
            "id": "12139",
            "ipAddress": "[no results]"
          }
        }
      ],
      "pageInfo": {
        "hasNextPage": true,
        "startCursor": "4277520:5",
        "endCursor": "4277520:9"
      }
    }
  }
}
```

Software characteristics query with filter

The following query retrieves endpoints that contain software installed by Deploy, where the package ID is `123`.

```
{
  endpoints {
    edges {
      node {
        ipAddress
        isVirtual
        domainName
        os {
          generation
        }
        lastLoggedInUser
        deployedSoftwarePackages(
          filter: {filters: [{op: EQ, path: "id", value: "123"}, {op: EQ, path:
"applicability", value: "Installed"}]}
        ) {
          id
        }
      }
    }
  }
}
```

Example response:

```
{
  "data": {
    "endpoints": {
      "edges": [
        {
          "node": {
            "ipAddress": "192.0.2.10",
```

```
            "isVirtual": true,
            "domainName": "(none)",
            "os": {
              "generation": "Debian 9"
            },
            "lastLoggedInUser": "[no results]",
            "deployedSoftwarePackages": [
              {
                "id": "123"
              }
            ]
          }
        },
        {
          "node": {
            "ipAddress": "192.0.2.20",
            "isVirtual": true,
            "domainName": "(none)",
            "os": {
              "generation": "CentOS 7"
            },
            "lastLoggedInUser": "admin",
            "deployedSoftwarePackages": [
              {
                "id": "123"
              }
            ]
          }
        },
        {
          "node": {
            "ipAddress": "192.0.2.30",
            "isVirtual": true,
            "domainName": "(none)",
```

```
            "os": {
              "generation": "Ubuntu 20.04"
            },
            "lastLoggedInUser": "admin",
            "deployedSoftwarePackages": [
              {
                "id": "123"
              }
            ]
          }
        }
      ]
    }
  }
}
```

Action examples

The following query and mutation retrieve action details and create an action.

Create action (subset of endpoints)

The following mutation deploys an action to increase the verbosity of log levels on Debian endpoints.

```
mutation {
  createAction(
    action: {description: "Increasing log verbosity level on all debian endpoints for troubleshooting", target: {targetGroup: "All Debian", platforms: [Linux]}, changeClientSetting: {name: LOG_VERBOSITY_LEVEL, value: "41"}}
  ) {
    id
  }
}
```

Example response:

```
{
  "data": {
    "createAction": {
      "id": "82"
    }
  }
}
```

Get action details

The following parameterized query retrieves details and any results for an action.

```
query ($id: ID!) {
  lastActionDetails(id: $id) {
    id
    name
    comment
    expireSeconds
    creationTime
    startTime
    expirationTime
    distributeSeconds
    status
    stoppedFlag
  }
  lastActionResults(id: $id) {
    id
    waiting
    downloading
    running
    waitingToRetry
    completed
    expired
    failed
```

```
    pendingVerification
    verified
    failedVerification
  }
}
```

Include the endpoint ID in the **QUERY VARIABLES** panel:

```
{
  "id": 12323
}
```

Example response:


```
{
  "data": {
    "lastActionDetails": {
      "id": "219",
      "name": "Distribute Tanium Standard Utilities",
      "comment": "Distribute Tanium Standard Utilities",
      "expireSeconds": 3900,
      "creationTime": "2022-03-14T18:05:41Z",
      "startTime": "2022-03-14T18:05:35Z",
      "expirationTime": "2022-03-14T19:10:35Z",
      "distributeSeconds": 3600,
      "status": "OPEN",
      "stoppedFlag": false
    },
    "lastActionResults": {
      "id": "219",
      "waiting": 0,
      "downloading": 0,
```

```
      "running": 0,
      "waitingToRetry": 0,
      "completed": 1,
      "expired": 0,
      "failed": 0,
      "pendingVerification": 0,
      "verified": 0,
      "failedVerification": 0
    }
  }
}
```

Deploy examples

The following queries and mutation require Deploy, retrieve information about software packages deployed in your environment, and allow you to deploy a software package to endpoints.

Deploy a package to all endpoints

The following mutation deploys a package to `All Computers`.

```
mutation {
  manageSoftware(
    operation: INSTALL
    softwarePackageID: 2
    start: "2021-10-27T00:00:00Z"
    end: "2021-11-03T00:00:00Z"
    target: {targetGroup: "All Computers"}
  ) {
    ID
    name
  }
}
```

Example response:

```
{
  "data": {
    "manageSoftware": {
      "ID": "2",
      "name": "Install Tanium Standard Utilities (Linux)"
    }
  }
}
```


Get package details

The following query retrieves multiple fields for all packages.

```
query PackagesQuery {
  packages {
    items {
      id
      name
      displayName
      command
      commandTimeout
      expireSeconds
      contentSet {
        id
        name
      }
      processGroupFlag
      skipLockFlag
      metadata {
        adminFlag
        name
        value
      }
      sourceHash
```

```
      sourceHashChangedFlag
      sourceID
      sourceName
      parameters {
        key
        value
      }
      rawParameterDefinition
      parameterDefinition {
        parameterType
        model
        parameters {
          model
          parameterType
          key
          label
          helpString
          defaultValue
          validationExpressions {
            model
            parameterType
            expression
            helpString
          }
          promptText
          heightInLines
          maxChars
          values
          restrict
          allowEmptyList
          minimum
          maximum
          stepSize
          snapInterval
```

```
            dropdownOptions {
              model
              parameterType
              name
              value
            }
            componentType
            startDateRestriction {
              model
              parameterType
              type
              interval
              intervalCount
              unixTimeStamp
            }
            endDateRestriction {
              model
              parameterType
              type
              interval
              intervalCount
              unixTimeStamp
            }
            startTimeRestriction {
              model
              parameterType
              type
              interval
              intervalCount
              unixTimeStamp
            }
            endTimeRestriction {
              model
              parameterType
```

```
            type
            interval
            intervalCount
            unixTimeStamp
          }
          allowDisableEnd
          defaultRangeStart {
            model
            parameterType
            type
            interval
            intervalCount
            unixTimeStamp
          }
          defaultRangeEnd {
            model
            parameterType
            type
            interval
            intervalCount
            unixTimeStamp
          }
          separatorText
        }
      }
      verifyExpireSeconds
    }
  }
}
```

Example response:

```
{
  "data": {
```

```
    "packages": {
      "items": [
        {
          "id": "1",
          "name": "Distribute Tanium Standard Utilities",
          "displayName": "Distribute Tanium Standard Utilities",
          "command": "cmd.exe /c cscript.exe //E:VBScript install-standard-utils.vbs \"Tools\\StdUtils\"",
          "commandTimeout": 2700,
          "expireSeconds": 3300,
          "contentSet": {
            "id": "5",
            "name": "Client Management"
          },
          "processGroupFlag": true,
          "skipLockFlag": false,
          "metadata": [],
          "sourceHash": "60b3e906f92929da67341792db9675d5cd91686546f01b57857686c8c6d84fa8",
          "sourceHashChangedFlag": false,
          "sourceID": 0,
          "sourceName": "",
          "parameters": [],
          "rawParameterDefinition": null,
          "parameterDefinition": null,
          "verifyExpireSeconds": 600
        },
        {
          "id": "2",
          "name": "Distribute Tanium Standard Utilities (Linux)",
          "displayName": "Distribute Tanium Standard Utilities (Linux)",
          "command": "/bin/bash distribute-tools.sh STRICT",
          "commandTimeout": 120,
          "expireSeconds": 720,
          "contentSet": {
```

```
            "id": "5",
            "name": "Client Management"
          },
          "processGroupFlag": true,
          "skipLockFlag": false,
          "metadata": [],
          "sourceHash":
"4ed7a30a1ca6c81be5a71b892dfadcdb489122cc641fa4b644f53255134215c9",
          "sourceHashChangedFlag": false,
          "sourceID": 0,
          "sourceName": "",
          "parameters": [],
          "rawParameterDefinition": null,
          "parameterDefinition": null,
          "verifyExpireSeconds": 600
        }
      ]
    }
  }
}
```


Get Deploy packages

The following query retrieves all Deploy packages.

```
{
  softwarePackages {
    edges {
      node {
        id
        productName
        productVendor
        productVersion
      }
```

```
    }
  }
}
```

Example response:

```
{
  "data": {
    "softwarePackages": {
      "edges": [
        {
          "node": {
            "id": "19",
            "productName": "Firefox (x64 en-US)",
            "productVendor": "Mozilla",
            "productVersion": "98.0"
          }
        },
        {
          "node": {
            "id": "30",
            "productName": "Power BI Desktop (x64)",
            "productVendor": "Microsoft",
            "productVersion": "2.102.845.0"
          }
        },
        {
          "node": {
            "id": "43",
            "productName": "VLC media player (64-bit)",
            "productVendor": "VideoLAN",
            "productVersion": "3.0.16.0"
          }
        },
```

```
        {
          "node": {
            "id": "46",
            "productName": "Visual Studio Code (x64 en-us)",
            "productVendor": "Microsoft",
            "productVersion": "1.65.2"
          }
        }
      ]
    }
  }
}
```


Get software deployment status

The following query retrieves the deployment status of all Deploy packages.

```
{
  softwareDeployment {
    ID
    name
    status {
      completeCount
      downloadCompleteWaitingCount
      downloadingCount
      failedCount
      notApplicableCount
      runningCount
      waitingCount
    }
    errors {
      error
      count
    }
```

```
  }
}
```

Example response:

```
{
  "data": {
    "softwareDeployment": [
      {
        "ID": "5",
        "name": "Install Tanium Standard Utilities",
        "status": {
          "completeCount": 1,
          "downloadCompleteWaitingCount": 0,
          "downloadingCount": 0,
          "failedCount": 0,
          "notApplicableCount": 0,
          "runningCount": 0,
          "waitingCount": 0
        },
        "errors": null
      },
      {
        "ID": "6",
        "name": "Install Tanium Standard Utilities (Linux)",
        "status": {
          "completeCount": 0,
          "downloadCompleteWaitingCount": 0,
          "downloadingCount": 1,
          "failedCount": 0,
          "notApplicableCount": 0,
          "runningCount": 0,
          "waitingCount": 0
        },
```

```
        "errors": null
      }
    ]
  }
}
```

Direct Connect examples

The following queries and mutations use Direct Connect to connect to a single endpoint, retrieve data, stop a process, and then close the connection. Queries that retrieve information from endpoints require Performance.

Open a connection to an endpoint

The following mutation uses Direct Connect to establish a connection to the endpoint with an ID of `12323`. You can retrieve IDs through the Get endpoints IDs from Tanium Data Service on page 45 query.

Direct Connect connections close after two minutes of inactivity.

```
mutation {
  openDirectConnection(input: {endpointID: "12323"}) {
    connectionID
  }
}
```

Example response:

```
{
  "data": {
    "openDirectConnection": {
      "connectionID": "86d9a9ac-0229-481b-9d88-5f1bcb1b177b"
    }
  }
}
```

Ping the connection to an endpoint

The following mutation retrieves the status for a Direct Connect connection. Use this mutation to check connection details or to keep the connection active. You need the `connectionID` that is returned by the mutation to open the connection.

Direct Connect connections close after two minutes of inactivity.

```
mutation ($connectionID: ID!) {
  pingDirectConnection(input: {connectionID: $connectionID}) {
    result
  }
}
```

Include the connection ID in the **QUERY VARIABLES** panel:

```
{
  "connectionID": "5fc564d6-5767-47fc-abb6-25cba65409d8"
}
```

Example response:

```
{
  "data": {
    "pingDirectConnection": {
      "result": true
    }
  }
}
```

Get data from an endpoint

After you establish a connection to an endpoint through Direct Connect, you can query the endpoint for specific information. The following query retrieves the CPU usage on the endpoint:

```
{
```

```
  directEndpoint (input : {endpointID: "12323"}) {
    performance {
      cpuUsagePercent
    }
  }
}
```

Example response:

```
{
  "data": {
    "directEndpoint": {
      "performance": {
        "cpuUsagePercent": 28.751501243887798
      }
    }
  }
}
```

Get process from an endpoint

After you establish a connection to an endpoint through Direct Connect, you can query the endpoint for process information. The following query retrieves the state of all processes running on the endpoint.

```
{
  directEndpoint (input : {endpointID: "12323"}) {
    processes {
      all {
        pid
        ppid
        name
        commandLine
        userName
        groupName
```

```
        memoryResidentBytes
      }
    }
  }
}
```

Example response:

```
{
  "data": {
    "directEndpoint": {
      "processes": {
        "all": [
          {
            "pid": 2092,
            "ppid": 496,
            "name": "TaniumReceiver.exe",
            "commandLine": "\"C:\\Program Files\\Tanium\\Tanium
Server\\TaniumReceiver.exe\" --service",
            "userName": "admin",
            "groupName": "test-group",
            "memoryResidentBytes": 59842560
          },
          {
            "pid": 5760,
            "ppid": 1112,
            "name": "TaniumClient.exe",
            "commandLine": "\"C:\\Program Files (x86)\\Tanium\\Tanium
Client\\TaniumClient.exe\" -c",
            "userName": "SYSTEM",
            "groupName": "NT AUTHORITY",
            "memoryResidentBytes": 17965056
          },
          {
```

```
            "pid": 1036,
            "ppid": 496,
            "name": "TaniumBlobService.exe",
            "commandLine": "\"C:\\Program Files\\Tanium\\Tanium Module Server\\services\\blob-service\\TaniumBlobService.exe\"",
            "userName": "SYSTEM",
            "groupName": "NT AUTHORITY",
            "memoryResidentBytes": 7426048
          }
        ]
      }
    }
  }
}
```


Get alerts from an endpoint

After you establish a connection to an endpoint through Direct Connect, you can query the endpoint for alert information. The following query retrieves alerts from an endpoint.

```
{
  directEndpoint (input : {endpointID: "12323"}) {
    alerts {
      all {
        schema
        key
        type
        ref
        topProcessesExpr
        labels
        pendingAt
        start
        resolvedAt
        leadup
```

```
        value
      }
    }
  }
}
```

Example response:

```
{
  "data": {
    "directEndpoint": {
      "alerts": {
        "all": [
          {
            "schema": 1,
            "key": "available-mem{heuristic=\"available-mem\"}",
            "type": "available-mem",
            "ref": null,
            "topProcessesExpr": null,
            "labels": {
              "heuristic": "available-mem"
            },
            "pendingAt": "2022-03-15T15:54:38.574990164Z",
            "start": "2022-03-15T15:54:38.574990164Z",
            "resolvedAt": null,
            "leadup": 300000000000,
            "value": 168.48828125
          }
        ]
      }
    }
  }
}
```

Stop a process on an endpoint

After you establish a connection to an endpoint through Direct Connect, you can stop running processes on the endpoint. The following mutation stops a process named `notepad.exe` on an endpoint. You need the `connectionID` that is returned by the mutation to open the connection.

```
mutation {
  killProcess(
    input: {connectionID: "7212763a-20aa-4cdd-a8b2-6b20b3968f2a", name: "notepad.exe", pid: 7056, signal: SIGKILL}
  ) {
    result
  }
}
```

Example response:

```
{
  "data": {
    "killProcess": {
      "result": true
    }
  }
}
```

Close connection to an endpoint

The following mutation closes a Direct Connect connection to an endpoint. You need the `connectionID` that is returned by the mutation to open the connection.

Direct Connect connections close after two minutes of inactivity.

```
mutation ($connectionID: ID!) {
  closeDirectConnection(input: {connectionID: $connectionID}) {
    result
  }
```

```
}
```

Include the connection ID in the **QUERY VARIABLES** panel:

```
{
  "connectionID": "5fc564d6-5767-47fc-abb6-25cba65409d8"
}
```

Example response:

```
{
  "data": {
    "closeDirectConnection": {
      "result": true
    }
  }
}
```