About the API

in Basics

Authors list

Published: Jul 18, 2022

The Deskpro API is a *REST* API that runs over HTTP(S). All API requests are made to a URL that begins with `http://example.com/api/v2/`.

The API uses *JSON* for requests and responses. There are a few exceptions to this rule where an API is explicitly designed to return a specific kind of resource. Those will be noted within the documentation.

Here's an example call made to the `helpdesk/discover` endpoint:

```
None


curl http://example.com/api/v2/helpdesk/discover

copy
```

The response will look something like this:

```
None


{
    "data": {
        "is_deskpro": true,
        "helpdesk_url": "https:\/\/example.com\/",
        "base_api_url": "https:\/\/example.com\/api\/v2\/",
        "build": "1477995658"
    },
    "meta": {},
    "linked": {}
}
```

Request Format

in Basics

Authors list

Published: Jul 18, 2022|Last updated: Mar 14, 2024

The Deskpro API is completely JSON.

- You SHOULD send a `Accept: application/json` to confirm that your app is expecting JSON back from the API.
- You MUST send `POST/PUT` requests with a `Content-Type: application/json` header and send a well-formed JSON payload.

HTTP Verbs

The HTTP verb you use is meaningful. Here's a summary of the HTTP verbs you can use with the Deskpro API and what they mean:

Verb	Description	Example
GET	Get a single resource or collection	GET /tickets or GET /tickets/13

DELETE	Delete a single resource	DELETE /ticket/13
POST	Create something new	POST /tickets
PUT	Update an existing resource	PUT /tickets/13

Response Format

When you perform a `GET` request to load a resource or a collection, you'll get a response like this.

```
None
```

```
{
    "data": {},
    "links": {},
    "meta": {}
}
```

copy

- `data` is the JSON representation of the resource requested, or an array of objects if you requested a collection

- `links` optional; when specified, it is an array of useful links. Resource collections always include links for “next”, “prev”, “first”, “last”.
- `meta` optional; misc information that varies based on endpoint. Resource collections should include “count”, “total_count*, “page”, “total_pages”.

Collection Responses

Example request to /tickets which is a collection resource:

None

```
curl -H "Authorization: key 2:YWK2PGCS8CNW62SR8TBTBKWMY"
http://example.com/api/v2/tickets

copy
```

Example response of a collection resource

None

```
{
   "links": {
      "first": "/tickets",
      "next": "/tickets?page=2",
      "prev": null,
```

```
        "last": "/tickets?page=106"
    },
    "data": [
        {
            "id": 1,
            "person_id": 4,
            "subject": "Foo Bar",
            "...": "..."
        },
        {
            "id": 2,
            "person_id": 12,
            "subject": "Fizz Buzz",
            "...": "..."
        }
    ],
    "meta": {
        "total": 5445,
        "page": 1,
        "per_page": 10,
        "total_pages": 545
    }
}
```

copy

Example fetching ticket objects for 3 specific ticket IDs

None

```
curl -H "Authorization: key 2:YWK2PGCS8CNW62SR8TBTBKWMY" \
```

```
    http://example.com/api/v2/tickets?ids=123,456,789
```

copy

During a resource collection request, you can specify the following optional query parameters:

- `count` the number of results to return
- `page`the page you are requesting

Collection responses are always an *array* of objects inside of `data`.

`links` will contain some useful links to help you paginate:

- `next` URL to the next page of results
- `prev` URL to the previous page of results
- `self` URL to the page you requested
- `first` URL to the first page of results
- `last` URL to the last page of results

`meta` will contain some useful data:

- `total` is the total count (all matching resources)
- `page` is current page number
- `per_page` is a count of the number of resources returned to you on this page
- `total_pages` is the total number of pages for this request

Tip: Most collections accept a `ids` parameters

Most collection resources accept an `ids` parameter where you can provide specific IDs for resources you want to retrieve. This makes it easy to load multiple objects at once.

Creating and Updating Content

When you create a resource with `POST`, the API will return a `201 Created` response. When you update a resource with a `PUT` request, the API will simply return a `204 No Content` empty response to acknowledge that the request succeeded.

In both cases, a `Location` header will be supplied with the full URL to the API endpoint where you can `GET` the resource.

TIP: If you would rather your `POST/PUT` request return the full resource automatically (i.e. you want to use it right away), you can submit your request with the `?follow_location=1` query string parameter. This will cause the API to return the resource as if you did the GET yourself.

Common HTTP Status Codes

The HTTP status codes that the API returns is significant. Here is a list of common return codes.

Code	Verbs	Example

200	GET	found and returned response is described above
404	Any	resource not found
201	POST	created resource the created entity is in the response (including “self” and “data” links)
204	PUT	updated resource no body will be in the response
400	Any	Bad Request malformed request or invalid input
401	Any	unauthorized you are not authenticated
403	Any	forbidden we accept your token, but you can’t perform this action, no resource has been updated

405	Any	method not allowed
429	Any	too many requests throttle/rate-limit hit
423	Any	Locked Special code we use when the helpdesk is offline for maintenance

API Key Limits

You can specify API limits on every API key you create. For example, it is not uncommon to set an upper limit on an API key to prevent some kind of mistake in its use (such as a flood or infinite loop situation). You can define these limits in terms of an hourly limit or a daily limit.

For cloud customers, there is also a default global limit to prevent abuse. You can raise this on request by contacting us at support@deskpro.com.

When you hit a rate limit, requests will begin to fail with an HTTP status code `429 Too Many Requests` error.

Error Responses

None

```
{
    "status": 400,
    "code": "invalid_json_body",
    "message": "Invalid JSON body"
}
```

copy

Error responses are also JSON and follow the shape described here.

- The `status` will be the HTTP status code that most closely matches the error.
- The `code` is a Deskpro-specific string to represent the specific error that happened. You may
  wish to use this value in your code in an if/switch etc to base your error handling on.
- The `message` is a human-readable English string that describes what went wrong. This value is typically only useful for developers.

Detailed Errors

None

```
{
    "status": 400,
```

```
    "code": "invalid_input",
    "message": "Request input is invalid.",
    "errors": {
        "errors": [
            {
                "code": "extra_fields",
                "message": "Unexpected field names: name"
            }
        ],
        "fields": {
            "email": {
                "errors": [
                    {
                        "code": "required",
                        "message": "This value should not be blank."
                    }
                ]
            },
            "date": {
                "errors": [
                    {
                        "code": "invalid_date",
                        "message": "This date is invalid"
                    }
                ],
                "fields": {
                    "day": {
                        "errors": [
                            {
                                "code": "out_of_range",
                                "message": "Must be a number between 1 and
31"
                            }
                        ]
                    },
                    "year": {
                        "errors": [
                            {
```

```
                        "code": "required",
                        "message": "This value is required"
                    }
                ]
            }
        }
    }
}


copy
```

In some cases, detailed error messages are available, such as when there are form validation errors.

The root `errors` property is an object that has two properties:

- `errors.errors` is a JSON array of error objects that happened on a global level (not specific to any one input).
- `errors.fields` is a JSON object that has a property for every expected input name that had an error. Inside of each field is a property named “errors” which is an array of error objects (code/message).

Batch Requests

in Basics

Authors list

Published: Jul 18, 2022

Sometimes you want to perform multiple queries all at once. For example, maybe you need to load a few collections to populate a form. Normally you would need to submit multiple HTTP requests which could be slow.

You POST a map to the `/batch` endpoint describing the requests you want to submit. The system will process all of your requests at once, and send you back a single result object. The keys of the result will be the same keys you submitted.

Example request using a key-value map of requests to execute in a batch:

None

```
curl -H "Authorization: key 1:CVRRGQ58QDX8H5B4W4RAJ978Q" \
     -H "Content-Type: application/json" \
     -X POST \
     -d '
 {
     "t": {
          "method": "POST",
          "url": "/api/tickets",
          "payload": {
               "title": "new-ticket"
          }
     },
     "john": "/api/people/1",
     "img": "/api/people/1/image",
     "parts": "/api/tickets/5/participants"
 }
' http://example.com/api/v2/batch

copy
```

Example response where each response uses the same key you sent in the request:

None

```
{
```

```
    "t": {
      "headers": {
          "response_code": 201,

      },
      "data": {
          "title": "new-ticket"
          "...": "..."
      }
    },
    "john": {
        "headers": {
            "response_code": 404
        },
        "data": {
            "...": "..."
        }
    },
    "...": "..."
}
```

copy

POST /batch

Post an object/map

Property	Description	Example
Note: `KEY`	Any arbitrary string you want to use to identify the request.	`req_1`, `department_info`, `foobar`, `12`

`KEY.url`	The API endpoint you want to call. You can also specify query-string params here.	`/api/v2/me`, `/api/v2/tickets?page=2`
`KEY.method`	(optional) `GET`, `POST`, `DELETE`, `PUT`	Defaults to `GET`
`KEY.payload`	(optional) When using POST/PUT, this is the JSON object you want to submit as the payload itself.	`{"foo": "bar"}`

GET /batch -- Short GET Syntax

You can also use the short GET syntax to process multiple GET requests all at once by specifying multiple `get[KEY]=endpoint_path` parameters.

Property	Description	Example
Note: `KEY`	Any arbitrary string you want to use to identify the request.	`req_1`, `department_info`, `foobar`, `12`
`get[KEY]`	The API endpoint to fetch	`/api/v2/me`, `/api/v2/tickets`

If you need to specify parameters for any of the `get[KEY]` calls (for example, to specify a page number), you can use the extended syntax.

Example GET call:

```
None

curl -H "Authorization: key 1:CVRRGQ58QDX8H5B4W4RAJ978Q" \
```

```
'http://example.com/api/v2/batch/api/v2/batch?get[stars]=/api/v2/tic
ket_stars&get[departments]=/api/v2/ticket_departments'
copy
```

Here's an example where the endpoint contains parameters. You need to encode '?' and '&' so they don't get read by the main batch request itself.

None

```
curl -H "Authorization: key 1:CVRRGQ58QDX8H5B4W4RAJ978Q" \

'http://example.com/api/v2/batch/api/v2/batch?get[stars]=/api/v2/tic
ket_stars&get[tickets]=/api/v2/tickets%3Fpage%3D2%26ids%3D1%2C2%2C3%
2C4'

# the last line decodes to a URL like:
# /api/v2/tickets?page=2&ids=1,2,3,4
```

Sideloading

in Basics

Authors list

Published: Jul 18, 2022

Most objects in Deskpro have other objects that are related to them. For example, a ticket has various users associated with it such as the creator, the assigned agent and any followers or CC'd users.

By default, the API will only return the IDs of these related objects and if you needed more information, you would need to use the API again to fetch the objects you wanted (for example, to get the actual title of the department a ticket was in).

However, this is tedious if you are doing it a lot. For this reason, the Deskpro API has the idea of *side-loading* which instructs the API to return related objects of certain types. You do this by specifying the type of object you want to side-load in the request as a query parameter: `include=type1,type2,type3`.

Examples:

- `/tickets/123?include=department`
- `/tickets/123?include=department,person,category,product,workflow`
- `/people/456?include=usergroup`

When you specify an `include` parameter, the related objects are returned in the response under the `linked` object. You'll get properties like `linked.TYPENAME.ID`.

(Note that the actual data response stays the same. Your own app logic will need to connect the ID from the `data` to the object returned in the `linked` section.)

Example ticket response without side loading

None

```
curl -H "Authorization: key 1:CVRRGQ58QDX8H5B4W4RAJ978Q" \
    http://example.com/api/v2/tickets/123
```

```
copy
```

Example response. Notice how we have IDs for various relationships such as the person, agent, or department.

```
None
```

```
{
    "data": {
        "id": 123,
        "ref": "XXXX-0028-IOCC",
        "department": 20,
        "person": 59080,
        "agent": 2,
        "status": "awaiting_user",
        "subject": "Example Ticket"
    },
    "meta": [],
    "linked": []
}
```

```
copy
```

Example WITH sideloading on department and person

```
None
```

```
curl -H "Authorization: key 1:CVRRGQ58QDX8H5B4W4RAJ978Q" \
    http://example.com/api/v2/tickets/123?include=department,person
```

copy

Example response with sideloaded objects

None


```
{
    "data": {
        "id": 123,
        "ref": "XXXX-0028-IOCC",
        "department": 20,
        "person": 59080,
        "agent": 2,
        "status": "awaiting_user",
        "subject": "Example Ticket"
    },
    "meta": [],
    "linked": {
        "department": {
            "20": {
                "id": 20,
                "title": "Support"
            }
        },
        "person": {
            "2": {
                "id": 2,
                "name": "John Doe",
```

```
                "primary_email": "agent@example.com"
            },
            "59080": {
                "id": 59080,
                "name": "Fionna Apple",
                "primary_email": "user@example.com"
            }
        }
    }
}


copy
```

Inline Sideloading

Above you saw that sideloading loads related resources into the `linked` key in the response. That means it's still up to you, the developer, to map an ID from the `data` to the correct value in `linked`.

To make this even easier, you can choose to *inline* the data. This replaces the IDs with the actual object itself, making consuming requests very very easy.

To enable inlining, you specify a query parameter `inline_sideloads=1`.

Same example above but with inlining enabled:

None

```
curl -H "Authorization: key 1:CVRRGQ58QDX8H5B4W4RAJ978Q" \

http://example.com/api/v2/tickets/123?include=department,person&inli

ne_sideloads=1

copy
```

Example response with sideloaded objects

None

```
{
    "data": {
        "id": 123,
        "ref": "XXXX-0028-IOCC",
        "department": {
            "id": 20,
            "title": "Support"
        },
        "person": {
            "id": 59080,
            "name": "Fionna Apple",
            "primary_email": "user@example.com"
        },
        "agent": {
            "id": 2,
```

```
            "name": "John Doe",
            "primary_email": "agent@example.com"
        },
        "status": "awaiting_user",
        "subject": "Example Ticket"
    },
    "meta": [],
    "linked": {}
}


copy
```

Inlining can make your life easier, but it comes at the cost of *data duplication*. For example, if you loading a list of 100 tickets, then the same agent might be assigned to many different tickets. Inlining would copy the agent data in multiple places which would increase the size of the response considerably.

So it's often beneficial to keep the default behavior and link IDs to `linked` objects in your application logic.