DETA Docs - English

Table of contents

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

export let Bubble = ({ item }) => { return (

<div style={{ display: 'flex', fontFamily: 'monospace', borderRadius: '3px', backgroundColor: '#ddd', display: 'inline', padding: '5px'}}> {item}
); }

General & Auth

> You can get your **Project Key** and your **Project ID** from your [Deta dashboard](https://web.deta.sh). You need these to talk with the Deta API.

> Base currently supports **maximum 16 digit numbers** (integers and floating points), please store larger numbers as a string.

Root URL

This URL is the base for all your HTTP requests:

`https://database.deta.sh/v1/{project_id}/{base_name}`

> The `base_name` is the name given to your database. If you already have a **Base**, then you can go ahead and provide it's name here. Additionally, you could provide any name here when doing any `PUT` or `POST` request and our backend will automatically create a new base for you if it does not exist. There is no limit on how many Bases you can create.

Auth

A **Project Key** *must* to be provided in the request **headers** as a value for the `X-API-Key` key for authentication. This is how we authorize your requests.

Example `'X-API-Key: a0abcyxz_aSecretValue'`.

Content Type

We only accept JSON payloads. Make sure you set the headers correctly: `'Content-Type: application/json'`

Endpoints

Put Items

`PUT /items`

Stores multiple items in a single request. This request overwrites an item if the key already exists.

<Tabs defaultValue=request values={[{ label: 'Request', value: 'request', }, { label: 'Response', value: 're-sponse', },] }>

JSON Payload	Required	Type	Description
`items`	Yes	`array`	An array of items `object` to be stored.

Example

```
{
   // array of items to put
   items: [
        {
            key: {key}, // optional, a random key is generated if not provided
            field1: value1,
            // rest of item
        },
        // rest of items
    ]
}
```

`207 Multi Status`

```
{
    processed: {
        items: [
            // items which were stored
        ]
    },
    failed: {
       items: [
          // items failed because of internal processing
       ]
    }
}
```

Client errors

In case of client errors, **no items** in the request are stored.

`400 Bad Request`

```
{
    errors : [
       // error messages
    ]
}
```

Bad requests occur in the following cases:

- if an item has a non-string key
- if the number of items in the requests exceeds 25
- if total request size exceeds 16 MB

- if any individual item exceeds 400KB
- if there are two items with identical keys

> Empty keys in objects/dictionaries/structs, like `{: value}` are invalid and will fail to be added during the backend processing stage.

Get Item

`GET /items/{key}`

Get a stored item.

> If the *key* contains url unsafe or reserved characters, make sure to url-encode the *key*. Otherwise, it will lead to unexpected behavior.

<Tabs defaultValue=request values={[{ label: 'Request', value: 'request', }, { label: 'Response', value: 'response', },] }>

URL Parameter	Required	Type	Description
`key`	Yes	`string`	The key (aka. ID) of the item you want to retrieve

`200 OK`

```
{
  key: {key},
  // the rest of the item
}
```

`404 Not Found`

```
{
  key: {key}
}
```

Delete Item

`DELETE /items/{key}`

Delete a stored item.

> If the *key* contains url unsafe or reserved characters, make sure to url-encode the *key*. Otherwise, it will lead to unexpected behavior.

<Tabs defaultValue=request values={[{ label: 'Request', value: 'request', }, { label: 'Response', value: 'response', },] }>

URL Parameter	Required	Type	Description
`key`	Yes	`string`	The key (aka. ID) of the item you want to delete.

The server will always return `200` regardless if an item with that `key` existed or not.

`200 OK`

```
{
  key: {key}
}
```

Insert Item

`POST /items`

Creates a new item only if no item with the same `key` exists.

<Tabs defaultValue=request values={[{ label: 'Request', value: 'request', }, { label: 'Response', value: 'response', },] }>

JSON Payload	Required	Type	Description
`item`	Yes	`object`	The item to be stored.

Example

```
{
    item: {
        key: {key}, // optional
        // rest of item
    }
}
```

`201 Created`

```
{
  key: {key}, // auto generated key if key was not provided in the request
  field1: value1,
  // the rest of the item
}
```

Client errors

`409 Conflict` (if key already exists)

```
{
  errors: [Key already exists]
}
```

`400 Bad Request`

```
{
  errors: [
     // error messages
  ]
}
```

Bad requests occur in the following cases:

- if the item has a non-string key
- if size of the item exceeds 400KB

Update Item

`PATCH /items/{key}`

Updates an item only if an item with `key` exists.

> If the *key* contains url unsafe or reserved characters, make sure to url-encode the *key*. Otherwise, it will lead to unexpected behavior.

<Tabs defaultValue=request values={[{ label: 'Request', value: 'request', }, { label: 'Response', value: 'response', },] }>

JSON Payload	Required	Type	Description
`set`	no	`object`	The attributes to be updated or created.
`increment`	no	`object`	The attributes to be incremented. Increment value can be negative.
`append`	no	`object`	The attributes to append a value to. Appended value must be a list.
`prepend`	no	`object`	The attributes to prepend a value to. Prepended value must be a list.
`delete`	no	`string` `array`	The attributes to be deleted.

Example

If the following item exists in the database

```
{
  key: user-a,
  username: jimmy,
  profile: {
    age: 32,
    active: false,
    hometown: pittsburgh
  },
  on_mobile: true,
  likes: [anime],
  purchases: 1
}
```

Then the request

```
{
   set : {
     // change ages to 33
     profile.age: 33,
     // change active to true
     profile.active: true,
     // add a new attribute `profile.email`
     profile.email: jimmy@deta.sh
   },

   increment :{
     // increment purchases by 2
     purchases: 2
   },

   append: {
     // append to 'likes'
     likes: [ramen]
   },

   // remove attributes 'profile.hometown' and 'on_mobile'
   delete: [profile.hometown, on_mobile]
}
```

results in the following item in the database:

```
{
  key: user-a,
  username: jimmy,
  profile: {
    age: 33,
    active: true,
    email: jimmy@deta.sh
  },
  likes: [anime, ramen],
  purchases: 3
}
```

`200 OK`

```
{
   key: {key},
   set: {
     // identical to the request
   },
   delete: [field1, ..] // identical to the request
}
```

Client errors

`404 Not Found` **(if key does not exist)**

```
{
  errors: [Key not found]
}
```

`400 Bad Request`

```
{
  errors: [
     // error messages
  ]
}
```

Bad requests occur in the following cases:

- if you're updating or deleting the `key`
- if `set` and `delete` have conflicting attributes
- if you're setting a hierarchical attribute but an upper level attribute does not exist, for eg. `{set: {user.age: 22}}` but `user` is not an attribute of the item.

Query Items

`POST /query`

List items that match a query.

<Tabs defaultValue=request values={[{ label: 'Request', value: 'request', }, { label: 'Response', value: 're-sponse', },] }>

JSON Payload	Required	Type	Description
`query`	No	`list`	list of a query
`limit`	No	`int`	no of items to return. min value 1 if used
`last`	No	`string`	last key seen in a previous paginated response

Example

```
{
   query: [
        // separate objects in the list are ORed
        // query evaluates to list all users whose hometown is Berlin and is
active OR all users who age less than 40
        {user.hometown: Berlin, user.active: true},
        {user.age?lt: 40}
   ],
   limit: 5,
   last: afsefasd // last key if applicable
}
```

The response is paginated if data process size exceeds 1 MB (before the query is applied) or the total number of items matching the `query` exceeds the `limit` provided in the request.

For paginated responses, `last` will return the last key seen in the response. You must use this `key` in the following request to continue retreival of items. If the response does not have the `last` key, then no further items are to be retreived.

> Upto 1 MB of data is retrieved before filtering with the query. Thus, in some cases you might get an empty list of items but still the `last` key evaluated in the response.
>
> To apply the query through all the items in your base, you have to call fetch until `last` is empty.

`200 OK`

```
{
    paging: {
        size: 5, // size of items returned
        last: adfjie // last key seen if paginated, provide this key in the
following request
    },
    items: [
       {
         key: {key},
         // rest of the item
       },
       // rest of the items
     ]
}
```

Client Errors

`400 Bad Request`

```
{
  errors: [
    // error messages
  ]
}
```

Bad requests occur in the following cases:

- if a query is made on the `key`
- if a query is not of the right format
- if `limit` is provided in the request and is less than 1

Issues

If you run into any issues, consider reporting them in our Github Discussions.

Go to TOC

Deta Base is a fully-managed, fast, scalable and secure NoSQL database with a focus on end-user simplicity. It offers a UI through which you can easily see, query, update and delete records in the database.

What can Deta Base be used for?

Deta Base is great for projects where configuring and maintaining a database is overkill.

Examples:

- Serverless Applications
- Internal Tools
- Stateful Integrations
- Hackathons and Side Projects
- Prototyping

Is my data secure?

Your data is encrypted and stored safely on AWS. Encryption keys are managed by AWS; AWS manages Exabytes of the world's most sensitive data.

How do I start?

1. Log in to Deta.
2. Grab your **Project Key** and your **Project ID** and start writing code in Python, Node.js, or over HTTP wherever you need persistent data storage.

Go to TOC

The Deta Base Python Async SDK can be used for reading and writing data asynchronously with Deta Base in Python.

> These are docs for an alpha version of the Python Deta Base Async SDK. The SDK API may change in a stable release.

Installing

```
pip install deta[async]==1.1.0a2
```

Instantiating

```
from deta import Deta

# initialize with a project key
# you can also init without specfying the project key explicitly
# the sdk looks for the DETA_PROJECT_KEY env var in that case
deta = Deta(project_key)

# create an async base client
async_db = deta.AsyncBase(base_name)
```

Methods

The `AsyncBase` class offers the same API to interact with your Base as the `Base` class:

Put

```
put(
    data: typing.Union[dict, list, str, int, float, bool],
    key: str = None,
    *,
    expire_in: int = None,
    expire_at: typing.Union[int, float, datetime.datetime]
)
```

- **data** (required): The data to be stored.
- **key** (optional): The key to store the data under. It will be auto-generated if not provided.
- **expire_in** (optional) - Accepts: `int` and `None`
  - Description: seconds after which the item will expire in, see also expiring items
- **expire_at** (optional) - Accepts: `int`, `float`, `datetime.datetime` and `None`
  - Description: time at which the item will expire in, can provide the timestamp directly(`int` or `float`) or a datetime.datetime object, see also expiring items

Example

```
import asyncio
from deta import Deta

async_db = Deta(project_key).AsyncBase(base_name)

```

```
async def put_item():
    item = {msg: hello}
    await async_db.put(item, test)
    print(put item:, item)

    # put expiring items
    # expire item in 300 seconds
    await async_db.put(item, expire_in=300)

    # with expire at
    expire_at = datetime.datetime.fromisoformat(2023-01-01T00:00:00)
    await async_db.put({name: max, age: 28}, max28, expire_at=expire_at)

    # close connection
    await async_db.close()

loop = asyncio.get_event_loop()
loop.run_until_complete(put_item())
```

Get

```
get(key: str)
```

- **key** (required): The key of the item to be retrieved.

Example

```
import asyncio
from deta import Deta

async_db = Deta(project_key).AsyncBase(base_name)

async def get_item():
    item = await async_db.get(my_key)
    print(got item:, item)

    # close connection
    await async_db.close()

loop = asyncio.get_event_loop()
loop.run_until_complete(get_item())
```

Delete

```
delete(key: str)
```

- **key** (required): The key of the item to delete.

Example

```
import asyncio
from deta import Deta

async_db = Deta(project_key).AsyncBase(base_name)

async def del_item():
    await async_db.delete(my-key)
    print(Deleted item)
    # close connection
    await async_db.close()

loop = asyncio.get_event_loop()
loop.run_until_complete(del_item())
```

Insert

`insert` is unique from `put` in that it will raise an error if the `key` already exists in the database, whereas `put` overwrites the item.

```
insert(
    data: typing.Union[dict, list, str, int, float, bool],
    key: str = None,
    *,
    expire_in: int = None,
    expire_at: typing.Union[int, float, datetime.datetime] = None
)
```

- **data** (required): The data to be stored.
- **key** (optional): The key to store the data under, will be auto generated if not provided.
- **expire_in** (optional) - Accepts: `int` and `None`
  - Description: seconds after which the item will expire in, see also expiring items
- **expire_at** (optional) - Accepts: `int`, `float`, `datetime.datetime` and `None`
  - Description: time at which the item will expire in, can provide the timestamp directly(`int` or `float`) or a datetime.datetime object, see also expiring items

Example

```
import asyncio
from deta import Deta

async_db = Deta(project_key).AsyncBase(base_name)

async def insert_item():
    item = {msg: hello}
    await async_db.insert(item, test)
    print(inserted item:, item)

    # put expiring items
    # expire item in 300 seconds
    await async_db.insert(item, expire_in=300)

    # with expire at
    expire_at = datetime.datetime.fromisoformat(2023-01-01T00:00:00)
    await async_db.insert({name: max, age: 28}, max28, expire_at=expire_at)

    # close connection
    await async_db.close()

loop = asyncio.get_event_loop()
loop.run_until_complete(insert_item())
```

Put Many

```
put_many(
    items: typing.List[typing.Union[dict, list, str, int, bool]],
    *,
    expire_in: int = None,
    expire_at: typing.Union[int, float, datetime.datetime] = None
)
```

- **items** (required): list of items to be stored.
- **expire_in** (optional) - Accepts: `int` and `None`
  - Description: seconds after which the item will expire in, see also expiring items
- **expire_at** (optional) - Accepts: `int`, `float`, `datetime.datetime` and `None`
  - Description: time at which the item will expire in, can provide the timestamp directly(`int` or `float`) or a datetime.datetime object, see also expiring items

Example

```
import asyncio
from deta import Deta

async_db = Deta(project_key).AsyncBase(base_name)

async def put_items():
    items = [{key: one, value: 1}, {key: two, value: 2}]
    await async_db.put_many(items)
    print(put items:, items)

    # put with expiring items
    # expire in 300 seconds
    await async_db.put_many(items, expire_in=300)

    # with expire at
     expire_at = datetime.datetime.fromisoformat(2023-01-01T00:00:00)
     await async_db.put_many(items, expire_at=expire_at)

    # close connection
    await async_db.close()

loop = asyncio.get_event_loop()
loop.run_until_complete(put_items())
```

Update

```
update(
    updates:dict,
    key:str,
    *,
    expire_in: int = None,
    expire_at: typing.Union[int, float, datetime.datetime] = None
)
```

- **updates** (required): A dict describing the updates on the item, refer to updates for more details.
- **key** (required): The key of the item to update.

- **expire_in** (optional) - Accepts: `int` and `None`
  - Description: seconds after which the item will expire in, see also expiring items
- **expire_at** (optional) - Accepts: `int`, `float`, `datetime.datetime` and `None`
  - Description: time at which the item will expire in, can provide the timestamp directly(`int` or `float`) or a datetime.datetime object, see also expiring items

Example

```
import asyncio
from deta import Deta

async_db = Deta(project_key).AsyncBase(base_name)

async def update_item():
    updates = {profile.age: 20, likes: async_db.util.append(ramen)}
    await async_db.update(updates, jimmy)
    print(updated user jimmy)

    # close connection
    await async_db.close()

loop = asyncio.get_event_loop()
loop.run_until_complete(update_item())
```

Fetch

`fetch(query=None, limit=1000, last=None)`

- **query** : a query or a list of queries
- **limit** : the limit of the number of items you want to recieve, min value `1` if used.
- **last**: the last key seen in a previous paginated response.

Example

```
import asyncio
from deta import Deta

async_db = Deta(project_key).AsyncBase(base_name)

async def fetch_items():
    query = {profile.age?gt: 20}
    res = await async_db.fetch(query)
    print(fetched items: res.items)

    # close connection
    await async_db.close()

loop = asyncio.get_event_loop()
loop.run_until_complete(fetch_items())
```

Issues

If you run into any issues, consider reporting them in our Github Discussions. We also appreciate any feedback.

deta

Go to TOC

Base UI let's you inspect, add, modify and delete records from a Base via a GUI.

Opening Base UI

You can open an individual Base's UI within any project by clicking on the Base name in the project sidebar.

base_ui_1

All the data from your Base should load into the table view when clicked. You can edit individual cells, if they do not contain an array or an object.

base_ui_2

Advanced Editing

You can expand any cell if you want to do advanced editing (like the editing of objects and arrays or type changes).

base_ui_3

Queries

If you don't want to deal with all of your data at once, you can use Deta Base's queries to get a filtered view. Click the **Query** button, enter your query, and hit enter or click **Fetch**.

base_ui_4

base_ui_5

Adding Items

You can add new items by clicking **+ Add**.

base_ui_6

New rows and edited rows will appear in yellow.

You can permanently save these modifications by clicking **Save edits**.

Deleting Items

To delete items, click on the checkbox(es) for any item(s) and then click the **Delete** button.

base_ui_7

Undoing Changes

You can revert your local changes, restoring the BaseUI state to the last fetch by clicking the **Undo** button.

base_ui_8

Final Notes

We hope you enjoy Base UI!

Base UI is still in Beta; it has been internally tested but may have some uncaught bugs or issues.

If you run into any issues, consider reporting them in our Github Discussions. We also take suggestions!

Go to TOC

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

Deta Base supports storing items with an expiration timestamp. Items with an expired timestamp will be automatically deleted from your Base.

Storing

Items specify the expiration timestamp value in a field name `__expires` in the item itself. The value is a Unix time, a number.

For e.g.

```
{
  key: item_key,
  msg: this will be deleted,
  __expires: 1672531200
}
```

The item above will be deleted automatically on `2023-01-01 00:00:00 GMT` (the equivalent date of the timestamp above).

You can use the Base SDK to `put`, `put_many` or `insert` items with an expiration timestamp (or the HTTP API directly).

> Storing an item with an already expired timestamp will not fail but the item will be immediately deleted.

> Base SDKs might offer higher level methods with easier APIs to specify the expiration timestamp. If they do so, they still store the timestamp in the item itself as mentioned above.

Examples

<Tabs groupId=lang defaultValue=js values={[{ label: 'JavaScript', value: 'js', }, { label: 'Python', value: 'py', }, { label: 'Go', value: 'go', }, { label: 'HTTP', value: 'http', },]}

```
const Deta = require('deta');
const db = Deta(project_key).Base('examples');

const item = {'value': 'temp'};

// expire in 300 seconds
await db.put(item, 'temp_key', {expireIn:300})

// expire at date
await db.put(item, 'temp_key', {expireAt: new Date('2023-01-01T00:00:00')})
```

```
import datetime
from deta import Deta

db = Deta(project_key).Base(examples)

item = {key: temp_key, value: temp}

# expire in 300 seconds
db.put(item, expire_in=300)

# expire at date
expire_at = datetime.datetime.fromisoformat(2023-01-01T00:00:00)
db.put(item, expire_at=expire_at)
```

```
import (
  log
  time

  github.com/deta/deta-go/deta
  github.com/deta/deta-go/service/base
)

type TmpData struct {
  Key string `json: key`
  Value string `json:value`
  // json struct tag `__expires` for expiration timestamp
  // 'omitempty' to prevent default 0 value
  Expires int64 `json:__expires,omitempty`
}

func main() {
  // errors ignored for brevity
  d, _ := deta.New(deta.WithProjectKey(project_key))
  db, _ := base.New(d, examples)

  tmp := &TmpData{
    Key: temp_key,
    Value: temp,
    Expires: time.Date(2023, 1, 1, 0, 0, 0, 0, time.UTC).Unix(),
  }
  _, err := db.Put(tmp)
  if err != nil {
    log.Fatal(failed to put item:, err)
  }
}
```

```
curl -X PUT https://database.deta.sh/v1/test_project_id/examples/items \
    -H Content-Type: application/json \
    -H X-Api-Key: test_project_key \
    -d {items:[{key: temp_key, value: temp, __expires: 1672531200}]}
```

Retrieving

When you retrieve items with an expiration timestamp, the timestamp value will be present in the `__expires` field. The value is a Unix time.

`Get` and `Query` operations will not retrieve already expired items.

Examples

<Tabs groupId=lang defaultValue=js values={[{ label: 'JavaScript', value: 'js', }, { label: 'Python', value: 'py', }, { label: 'Go', value: 'go', }, { label: 'HTTP', value: 'http', },]}

```
const { __expires } = await db.get(temp_key);
```

```
expires = db.get(temp_key).get(__expires)
```

```
dest := struct{
  Key     string `json:key`
  Expires int64  `json:__expires,omitempty`
}{}

if err := db.Get(temp_key, &dest); err != nil {
  log.Fatal(failed to get item:, err)
}

expires := dest.Expires
```

```
curl https://database.deta.sh/v1/test_project_id/examples/items/temp_key \
    -H X-Api-Key: test_project_key

{key: temp_key, value: temp, __expires: 1672531200}
```

Updating

You can update the expiration timestamp with a new timestamp by updating the value of the `__expires` as long as the item has not already expired.

Updating other fields of the item **does not** update (or renew) the expiration timestamp. You **must** update the value of `__expires` field.

You can use the Base SDK to `update` the expiration timestamp (or the HTTP API directly).

Examples

<Tabs groupId=lang defaultValue=js values={[{ label: 'JavaScript', value: 'js', }, { label: 'Python', value: 'py', }, { label: 'Go', value: 'go', }, { label: 'HTTP', value: 'http', },]}

```
// update item to expire in 300 seconds from now
await db.update(null, temp_key, {expireIn: 300})

// update item to expire at date
await db.update(null, temp_key, {expireAt: new Date('2023-01-01T00:00:00')})
```

```
# update item to expire in 300 seconds from now
db.update(None, temp_key, expire_in=300)
```

```
# update item to expire at date
expire_at = datetime.datetime.fromisoformat(2023-01-01T00:00:00)
db.update(None, temp_key, expire_at=expire_at)
```

```
updates := base.Updates{
  __expires: 1672531200,
}
if err := db.Update(temp_key, updates); err != nil {
  log.Fatal(failed to update item:, err)
}
```

```
curl -X PATCH https://database.data.sh/v1/test_project_id/examples/items/temp_key
\
    -H Content-Type: application/json \
    -H X-Api-Key: test_project_key \
    -d {set: {__expires: 1672531200}}
```

Issues

If you run into any issues, consider reporting them in our Github Discussions.

Go to TOC

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

Building a Simple CRUD with Deta Base

Setup

Two dependencies are needed for this project, `deta` and `express`:

```
npm install deta express
```

To configure the app, import the dependencies and instantiate your database.

```
const express = require('express');
const { Deta } = require('deta');

const deta = Deta('myProjectKey'); // configure your Deta project
const db = deta.Base('simpleDB');  // access your DB


const app = express(); // instantiate express

app.use(express.json()) // for parsing application/json bodies
```

Creating Records

For our database we are going to store records of users under a unique `key`. Users can have three properties:

```
{
    name: string,
    age: number,
    hometown: string
}
```

We'll expose a function that creates user records to HTTP `POST` requests on the route `/users`.

```
app.post('/users', async (req, res) => {
    const { name, age, hometown } = req.body;
    const toCreate = { name, age, hometown};
    const insertedUser = await db.put(toCreate); // put() will autogenerate a key
for us
    res.status(201).json(insertedUser);
});
```

Request

`POST` a payload to the endpoint:

```
{
    name: Beverly,
    age: 44,
    hometown: Copernicus City
}
```

Response

Our server should respond with a status of `201` and a body of:

```
{
    key: dl9e6w6859a9,
    name: Beverly,
    age: 44,
    hometown: Copernicus City
}
```

Reading Records

To read records, we can simply use `Base.get(key)`.

If we tie a `GET` request to the `/users` path with a path param giving a user id (i.e. `/users/dl9e6w6859a9`), we can return a record of the user over HTTP.

```
app.get('/users/:id', async (req, res) => {
    const { id } = req.params;
    const user = await db.get(id);
    if (user) {
        res.json(user);
    } else {
        res.status(404).json({message: user not found});
    }
});
```

Another option would to use `Base.fetch(query)` to search for records to return, like so:

<Tabs groupId=js-version defaultValue=new values={[{ label: 'version < 1.0.0', value: 'legacy', }, { label: 'version >= 1.0.0', value: 'new', },] }>

```
app.get('/search-by-age/:age', async (req, res) => {
    const { age } = req.params;
    return (await db.fetch({'age': age}).next()).value;
});
```

```
app.get('/search-by-age/:age', async (req, res) => {
    const { age } = req.params;
    const { items } = await db.fetch({'age': age});
    return items;
});
```

Request

Let's try reading the record we just created.

Make a `GET` to the path `/users/dl9e6w6859a9`.

Response

The server should return the same record:

```
{
    key: dl9e6w6859a9,
    name: Beverly,
    age: 44,
    hometown: Copernicus City
}
```

Updating Records

To update records under a given `key`, we can use `Base.put()`, which will replace the record under a given key.

We can tie a `PUT` request to the path `/users/{id}` to update a given user record over HTTP.

```
app.put('/users/:id', async (req, res) => {
    const { id } = req.params;
    const { name, age, hometown } = req.body;
    const toPut = { key: id, name, age, hometown };
    const newItem = await db.put(toPut);
    return res.json(newItem)
});
```

Request

We can update the record by passing a `PUT` to the path `/users/dl9e6w6859a9` with the following payload:

```
{
    name: Wesley,
    age: 24,
    hometown: San Francisco
}
```

Response

Our server should respond with the new body of:

```
{
    key: dl9e6w6859a9,
    name: Wesley,
    age: 24,
    hometown: San Francisco
}
```

Deleting Records

To delete records under a given `key`, we can use `Base.delete(key)`, which will remove the record under a given key.

We can tie a `DELETE` request to the path `/users/{id}` to delete a given user record over HTTP.

```
app.delete('/users/:id', async (req, res) => {
    const { id } = req.params;
    await db.delete(id);
    res.json({message: deleted})
});
```

Request

We can delete the record by passing a `DELETE` to the path `/users/dl9e6w6859a9`.

```
{
    name: Wesley,
    age: 24,
    hometown: San Francisco
}
```

Response

Our server should respond with:

```
{
    message: deleted
}
```

Issues

If you run into any issues, consider reporting them in our Github Discussions.

Go to TOC

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

Building a Simple CRUD with Deta Base

Setup

Two dependencies are needed for this project, `deta` and `flask`:

```
pip install deta flask
```

To configure the app, import the dependencies and instantiate your database.

```
from flask import Flask, request, jsonify
from deta import Deta


deta = Deta('myProjectKey') # configure your Deta project
db = deta.Base('simpleDB')  # access your DB
app = Flask(__name__)
```

Creating Records

For our database we are going to store records of users under a unique `key`. Users can have three properties:

```
{
    name: str,
    age: int,
    hometown: str
}
```

We'll expose a function that creates user records to HTTP `POST` requests on the route `/users`.

```
@app.route('/users', methods=[POST])
def create_user():
    name = request.json.get(name)
    age = request.json.get(age)
    hometown = request.json.get(hometown)

    user = db.put({
        name: name,
        age: age,
        hometown: hometown
    })

    return jsonify(user, 201)
```

Request

`POST` a payload to the endpoint:

```
{
    name: Beverly,
    age: 44,
    hometown: Copernicus City
}
```

Response

Our server should respond with a status of `201` and a body of:

```
{
    key: dl9e6w6859a9,
    name: Beverly,
    age: 44,
    hometown: Copernicus City
}
```

Reading Records

To read records, we can simply use `db.get(key)`.

If we tie a `GET` request to the `/users` path with a param giving a user id (i.e. `/users/dl9e6w6859a9`), we can return a record of the user over HTTP.

```
@app.route(/users/<key>)
def get_user(key):
    user = db.get(key)
    return user if user else jsonify({error: Not found}, 404)
```

Request

Let's try reading the record we just created.

Make a `GET` to the path (for example) `/users/dl9e6w6859a9`.

Response

The server should return the same record:

```
{
    key: dl9e6w6859a9,
    name: Beverly,
    age: 44,
    hometown: Copernicus City
}
```

Updating Records

To update records under a given `key`, we can use `db.put()`, which will replace the record under a given key.

We can tie a `PUT` request to the path `/users/{id}` to update a given user record over HTTP.

```
@app.route(/users/<key>, methods=[PUT])
def update_user(key):
    user = db.put(request.json, key)
    return user
```

Request

We can update the record by passing a `PUT` to the path `/users/dl9e6w6859a9` with the following payload:

```
{
    name: Wesley,
    age: 24,
    hometown: San Francisco
}
```

Response

Our server should respond with the new body of:

```
{
    key: dl9e6w6859a9,
    name: Wesley,
    age: 24,
    hometown: San Francisco
}
```

Deleting Records

To delete records under a given `key`, we can use `Base.delete(key)`, which will remove the record under a given key.

We can tie a `DELETE` request to the path `/users/{id}` to delete a given user record over HTTP.

```
@app.route(/users/<key>, methods=[DELETE])
def delete_user(key):
    db.delete(key)
    return jsonify({status: ok}, 200)
```

Request

We can delete the record by passing a `DELETE` to the path `/users/dl9e6w6859a9`.

```
{
    name: Wesley,
    age: 24,
    hometown: San Francisco
}
```

Response

Our server should respond with:

```
None
```

Issues

If you run into any issues, consider reporting them in our Github Discussions.

Go to TOC

Deta Base supports queries for fetching data that match certain conditions. Queries are regular objects/dicts/maps with conventions for different operators for specifying the conditions.

Operators

Queries support the following operators:

Equal

```
{age: 22, name: Beverly}

// hierarchical
{user.profile.age: 22, user.profile.name: Beverly}
```

```
{fav_numbers: [2, 4, 8]}
```

```
{time: {day: Tuesday, hour: 08:00}}
```

Not Equal

```
{user.profile.age?ne: 22}
```

Less Than

```
{user.profile.age?lt: 22}
```

Greater Than

```
{user.profile.age?gt: 22}
```

Less Than or Equal

```
{user.profile.age?lte: 22}
```

Greater Than or Equal

```
{user.profile.age?gte: 22}
```

Prefix

```
{user.id?pfx: afdk}
```

Range

```
{user.age?r: [22, 30]}
```

Contains

```
{
  // if user email contains the substring @deta.sh
  user.email?contains: @deta.sh
}
```

```
{
  // if berlin is in a list of places lived
  user.places_lived_list?contains: berlin
}
```

Not Contains

```
{
  // if user email does not contain @deta.sh
  user.email?not_contains: @deta.sh // 'user.email?!contains' also valid
}
```

```
{
  // if berlin is not in a list of places lived
  user.places_lived_list?not_contains: berlin //
'user.places_lived_list?!contains' also valid
}
```

`?contains` and `?not_contains` only works for a list of strings if checking for membership in a list; it does not apply to list of other data types. You can store your lists always as a list of strings if you want to check for membership.

Logical Operators

AND

The entries in a single query object are `AND` ed together. For e.g. the query:

```
{
    active: true,
    age?gte: 22
}
```

will retrieve items where `active` is `true` **and** `age` is greater than or equal to `22`.

The query above would translate to `SQL` as:

```
SELECT * FROM base WHERE active=1 AND age>=22;
```

OR

Multiple query objects in a list are `OR` ed together. For eg. the queries:

```
[{age?lte: 30}, {age?gte: 40}]
```

will retrieve items where `age` is less than equal to `30` **or** `age` is greater than equal to `40`.

The query above would translate to `SQL` as:

```
SELECT * FROM base WHERE age<=30 OR age>=40;
```

Hierarchy

You can use the period character `.` to query for hierarchical fields within the data. For instance if you have the following item in the base:

```
{
    key: user-key,
    profile: {
        age: 22,
        active: true
    }
}
```

Then you can query for the `active` and `age` within `profile` directly:

```
{
    profile.age: 22,
    profile.active: true
}
```

Querying Keys

You need to consider the following when querying on keys:

- The keys must be strings hence the operation values **must** also be strings.
- The contains and not-contains operators **are not supported**.
- The `AND` and `OR` operations for different query values **are not supported**. For e.g. **the following queries are invalid**:

```
{
    // different AND key queries (invalid query)
    key: a,
    key?pfx: b
}
```

```
{
    // different OR key queries (invalid query)
    [{key?pfx:a}, {key?pfx: b}]
}
```

Issues

If you run into any issues, consider reporting them in our Github Discussions.

deta

Go to TOC

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

The Deta library is the easiest way to store and retrieve data from your Deta Base. Currently, we support JavaScript (Node + Browser), Python 3 and Go. Drop us a line if you want us to support your favorite language.

A Deta Base instance is a collection of data, not unlike a Key-Value store, a MongoDB collection or a PostgreSQL/MySQL table. It will grow with your app's needs.

We have an alpha version of the Python Base Async SDK. [Check the documentation here](/docs/base/async_sdk).

Installing

Using NPM:

```
npm install deta
```

Using Yarn:

```
yarn add deta
```

```
pip install deta
```

```
go get github.com/deta/deta-go
```

If you are using the Deta SDK within a [Deta Micro](/docs/micros/about), you must include `deta` in your dependencies file (`package.json` or `requirements.txt`) to install the latest sdk version.

Instantiating

To start working with your Base, you need to import the `Deta` class and initialize it with a **Project Key**. Then instantiate a subclass called `Base` with a database name of your choosing.

Deta Bases are created for you automatically when you start using them.

```
const { Deta } = require('deta'); // import Deta

// Initialize with a Project Key
const deta = Deta('project key');

// This how to connect to or create a database.
const db = deta.Base('simple_db');
```

```
// You can create as many as you want without additional charges.
const books = deta.Base('books');
```

If you are using Deta Base within a [Deta Micro](/docs/micros/about), the **Deta SDK** comes pre-installed and a valid project key is pre-set in the Micro's environment. There is no need to install the SDK or pass a key in the initialization step.

```
const { Deta } = require('deta');

const deta = Deta();
```

If you are using the `deta` npm package of `0.0.6` or below, `Deta` is the single default export and should be imported as such.

```
const Deta = require('deta');
```

```
from deta import Deta  # Import Deta

# Initialize with a Project Key
deta = Deta(project key)

# This how to connect to or create a database.
db = deta.Base(simple_db)

# You can create as many as you want without additional charges.
books = deta.Base(books)
```

If you are using Deta Base within a [Deta Micro](/docs/micros/about), the **Deta SDK** comes pre-installed and a valid project key is pre-set in the Micro's environment. There is no need to install the SDK or pass a key in the the initialization step.

```
from deta import Deta

deta = Deta()
```

= 1.0.0', value: 'new', },] }>

```
import (
  fmt
  github.com/deta/deta-go
)

func main(){
  // initialize with project key
  // returns ErrBadProjectKey if project key is invalid
  d, err := deta.New(project_key)
  if err != nil {
    fmt.Println(failed to init new Deta instance:, err)
    return
  }
```

```
    // initialize with base name
    // returns ErrBadBaseName if base name is invalid
    db, err := d.NewBase(base_name)
    if err != nil {
      fmt.Println(failed to init new Base instance:, err)
      return
    }
}
```

```
import (
    fmt

    github.com/deta/deta-go/deta
    github.com/deta/deta-go/service/base
)

func main() {

    // initialize with project key
    // returns ErrBadProjectKey if project key is invalid
    d, err := deta.New(deta.WithProjectKey(project_key))
    if err != nil {
        fmt.Println(failed to init new Deta instance:, err)
        return
    }

    // initialize with base name
    // returns ErrBadBaseName if base name is invalid
    db, err := base.New(d, base_name)
    if err != nil {
        fmt.Println(failed to init new Base instance:, err)
        return
    }
}
```

Your project key is confidential and meant to be used by you. Anyone who has your project key can access your database. Please, do not share it or commit it in your code.

Using

Deta's `Base` class offers the following methods to interact with your Deta Base:

`put` – Stores an item in the database. It will update an item if the key already exists.

`insert` – Stores an item in the database but raises an error if the key already exists. (2x slower than `put`).

`get` – Retrieves an item from the database by its key.

`fetch` – Retrieves multiple items from the database based on the provided (optional) filters.

`delete` – Deletes an item from the database.

`update` – Updates an item in the database.

Storing Numbers

> Base currently supports **maximum 16 digit numbers** (integers and floating points), please store larger numbers as a string.

Put

`put` is the fastest way to store an item in the database.

If an item already exists under a given key, put will replace this item.

In the case you do not provide us with a key, we will auto generate a 12 char long string as a key.

<Tabs groupId=preferred-language defaultValue=js values={[{ label: 'JavaScript', value: 'js', }, { label: 'Python', value: 'py', }, { label: 'Go', value: 'go', },] }>

`async put(data, key = null, options = null)`

Parameters

- **data** (required) – Accepts: `object` (serializable), `string`, `number`, `boolean` and `array`.
  - Description: The data to be stored.
- **key** (optional) – Accepts: `string`, `null` or `undefined`
  - Description: the key (aka ID) to store the data under. Will be auto generated if not provided.
- **options** (optional) - Accepts: `object`, `null` or `undefined`

  ```
  {
    expireIn: number,
    expireAt: Date
  }
  ```

  - Description: Optional parameters.
    - **expireIn** : item will expire in `expireIn` seconds after a successfull put operation, see also expiring items.
    - **expireAt** : item will expire at `expireAt` date, see also expiring items.

Code Example

```
const Deta = require('deta');

const deta = Deta(project key);
const db = deta.Base(simple_db);

// store objects
// a key will be automatically generated
await db.put({name: alex, age: 77})
// we will use one as a key
await db.put({name: alex, age: 77}, one)
// the key could also be included in the object itself
await db.put({name: alex, age: 77, key:one})

// store simple types
```

```
await db.put(hello, worlds)
await db.put(7)
// success is the value and smart_work is the key.
await db.put(success, smart_work)
await db.put([a, b, c], my_abc)

// put expiring items
// expire item in 300 seconds
await db.put({name: alex, age: 21}, alex21, {expireIn: 300})
// expire item at expire date
await db.put({name: max, age:28}, max28, {expireAt: new Date('2023-01-
01T00:00:00')})
```

Returns

`put` returns a promise which resolves to the item on a successful put, otherwise it throws an Error.

```
put(
  data: typing.Union[dict, list, str, int, float, bool],
  key: str = None,
  *,
  expire_in: int = None,
  expire_at: typing.Union[int, float, datetime.datetime] = None
)
```

Parameters

- **data** (required) – Accepts: `dict`, `str`, `int`, `float`, `bool` and `list`.
  - Description: The data to be stored.
- **key** (optional) – Accepts: `str` and `None`
  - Description: the key (aka ID) to store the data under. Will be auto generated if not provided.
- **expire_in** (optional) - Accepts: `int` and `None`
  - Description: seconds after which the item will expire in, see also expiring items
- **expire_at** (optional) - Accepts: `int`, `float`, `datetime.datetime` and `None`
  - Description: time at which the item will expire in, can provide the timestamp directly(`int` or `float`) or a datetime.datetime object, see also expiring items

Code Example

```
from deta import Deta
deta = Deta(project key)
db = deta.Base(simple_db)

# store objects
# a key will be automatically generated
db.put({name: alex, age: 77})
# we will use one as a key
db.put({name: alex, age: 77}, one)
# the key could also be included in the object itself
db.put({name: alex, age: 77, key: one})

# simple types
db.put(hello, worlds)
db.put(7)
# success is the value and smart_work is the key.
db.put(success, smart_work)
db.put([a, b, c], my_abc)

# expiring items
# expire item in 300 seconds
db.put({name: alex, age: 23}, alex23, expire_in=300)
# expire item at date
expire_at = datetime.datetime.fromisoformat(2023-01-01T00:00:00)
db.put({name: max, age: 28}, max28, expire_at=expire_at)
```

Returns

`put` returns the item on a successful put, otherwise it raises an error.

= 1.0.0', value: 'new', },] }>}>

`Put(item interface{}) (string, error)`

Parameters

- **item** : The item to be stored, should be a `struct` or a `map`. If the item is a `struct` provide the field keys for the data with json struct tags. The key of the item must have a json struct tag of `key`. For storing expiring items, the field name `__expires` should be used with a Unix Time value, see also expiring items.

Note for storing numbers

Code Example

```
import (
    log
    time
    github.com/deta/deta-go
)

// User represents a user
type User struct{
    // json struct tag 'key' used to denote the key

    Key      string `json:key`
    Username string `json:username`
    Active   bool `json:active`
    Age      int `json:age`
    Likes    []string `json:likes`
    // json struct tag '__expires' for expiration timestamp,
    // tag has 'omitempty' for ommission of default 0 values
    Expires  int64 `json:__expires,omitempty`
}

func main(){
    // error ignored for brevity
    d, _ := deta.New(project key)
    db, _ := d.NewBase(users)

    // a user
    u := &User{
        Key: kasdlj1,
        Username: jimmy,
        Active: true,
        Age: 20,
        Likes: []string{ramen},
    }

    // put item in the database
    key, err := db.Put(u)
    if err != nil {
        log.Fatal(failed to put item:, err)
    }
    log.Println(put item with key:, key)

    // can also use a map
    um := map[string]interface{}{
      key: kasdlj1,
      username: jimmy,
      active: true,
      age: 20,
      likes: []string{ramen},
    }

    key, err = db.Put(um)
    if err != nil {
        log.Fatal(failed to put item:, err)
    }
    log.Println(put item with key:, key)

    // expiring items
    u = &User {
      Key: will_be_deleted,
      Username: test_user,
      Expires: time.Date(2023, 1, 1, 0, 0, 0, 0, time.UTC).Unix(),
    }
    key, err = db.Put(u)
    if err != nil {
        log.Fatal(failed to put item:, err)
    }
    log.Println(put item with key:, key)

    // maps with expiration timestamp
    um = map[string]interface{}{
      key: will_be_deleted,
      test: true,
      __expires: time.Date(2023, 1, 1, 0, 0, 0, 0, time.UTC).Unix(),
    }

    key, err = db.Put(um)
    if err != nil {
        log.Fatal(failed to put item:, err)
    }
    log.Println(put item with key:, key)
}
```

`Put(item interface{}) (string, error)`

Parameters

- **item** : The item to be stored, should be a `struct` or a `map`. If the item is a `struct` provide the field keys for the data with json struct tags. The key of the item must have a json struct tag of `key`. For storing expiring items, the field name `__expires` should be used with a Unix Time value, see also expiring items.

Note for storing numbers

Code Example

```
import (
    log
  time

    github.com/deta/deta-go/deta
    github.com/deta/deta-go/service/base
)

type User struct {
  // json struct tag 'key' used to denote the key

    Key      string   `json:key`
    Username string   `json:username`
    Active   bool     `json:active`
    Age      int      `json:age`
    Likes    []string `json:likes`
  // json struct tag '__expires' for expiration timestamp
  // 'omitempty' for omission of default 0 value
  Expires  int64 `json:__expires,omitempty`
}

func main() {
  // errors ignored for brevity
    d, _ := deta.New(deta.WithProjectKey(project_key))
    db, _ := base.New(d, users)

    u := &User{
        Key:       kasdlj1,
        Username: jimmy,
        Active:    true,
        Age:       20,
        Likes:     []string{ramen},
    }
    key, err := db.Put(u)
    if err != nil {
    log.Fatal(failed to put item:, err)
    }
    log.Println(successfully put item with key, key)

    // can also use a map
    um := map[string]interface{}{
        key:       kasdlj1,
        username: jimmy,
        active:    true,
        age:       20,
        likes:     []string{ramen},
    }
    key, err = db.Put(um)
    if err != nil {
    log.Fatal(failed to put item:, err)
    }
    log.Println(Successfully put item with key:, key)

  // put with expires
  u := &User{
    Key: will_be_deleted,
    Username: test_user,
    Expires: time.Date(2023, 1, 1, 0, 0, 0, 0, time.UTC).Unix(),
  }
  key, err = db.Put(u)
  if err != nil {
    log.Fatal(failed to put item:, err)
  }
  log.Println(put item with key:, key)

  // put map with expires
  um = map[string]interface{}{
    key: will_be_deleted,
    test: true,
    __expires: time.Data(2023, 1, 1, 0, 0, 0, 0, time.UTC).Unix(),
  }
  key, err = db.Put(um)
  if err != nil {
    log.Fatal(failed to put item:, err)
  }
  log.Println(put item with key:, key)
}
```

Returns

Returns the `key` of the item stored and an `error`. Possible error values:

- `ErrBadItem` : bad item, item is of unexpected type
- `ErrBadRequest`: item caused a bad request response from the server
- `ErrUnauthorized`: unuathorized
- `ErrInternalServerError`: internal server error

> Empty keys in objects/dictionaries/structs, like `{: value}` are invalid and will fail to be added during the backend processing stage.

Get

`get` retrieves an item from the database by it's `key`.

`async get(key)`

Parameters

- **key** (required) – Accepts: `string`
  - Description: the key of which item is to be retrieved.

Code Example

```
const item = await db.get('one'); // retrieving item with key one
```

Returns

If the record is found, the promise resolves to:

```
{
  name: 'alex', age: 77, key: 'one'
}
```

If not found, the promise will resolve to `null`.

```
get(key: str)
```

Parameter Types

- **key** (required) – Accepts: `str`
  - Description: the key of which item is to be retrieved.

Code Example

```
item = db.get(one) # retrieving item with key one
```

Returns

If the record is found:

```
{
  name: alex, age: 77, key: one
}
```

If not found, the function will return `None`.

<Tabs groupId=go-version defaultValue=new values={[{ label: 'version < 1.0.0', value: 'legacy', }, { label: 'version >= 1.0.0', value: 'new', },] }>

`Get(key string, dest interface{}) error`

Parameters

- **key**: the key of the item to be retrieved
- **dest**: the result will be stored into the value pointed by `dest`

Code Example

```

import (
    fmt
    github.com/deta/deta-go
)

type User struct{
    Key string `json:key` // json struct tag 'key' used to denote the key

    Username string `json:username`
    Active bool `json:active`
    Age int `json:age`
    Likes []string `json:likes`
}

func main(){
    // error ignored for brevity
    d, _ := deta.New(project key)
    db, _ := d.NewBase(users)

    // a variable to store the result
    var u User

    // get item
    // returns ErrNotFound if no item was found
    err := db.Get(kasdlj1, &u)
    if err != nil{
        fmt.Println(failed to get item:, err)
    }
}
```

`Get(key string, dest interface{}) error`

Parameters

- **key**: the key of the item to be retrieved
- **dest**: the result will be stored into the value pointed by `dest`

Code Example

```
import (
    fmt

    github.com/deta/deta-go/deta
    github.com/deta/deta-go/service/base
)

type User struct {
    Key      string   `json:key` // json struct tag 'key' used to denote the key

    Username string   `json:username`
    Active   bool     `json:active`
    Age      int      `json:age`
    Likes    []string `json:likes`
}

func main() {
    d, err := deta.New(deta.WithProjectKey(project_key))
    if err != nil {
        fmt.Println(failed to init new Deta instance:, err)
        return
    }

    db, err := base.New(d, users)
    if err != nil {
        fmt.Println(failed to init new Base instance:, err)
    }

    // a variable to store the result
    var u User

    // get item
    // returns ErrNotFound if no item was found
    err = db.Get(kasdlj1, &u)
    if err != nil {
        fmt.Println(failed to get item:, err)
    }
}
```

Returns

Returns an `error`. Possible error values:

- `ErrNotFound`: no item with such key was found
- `ErrBadDestination`: bad destination, result could not be stored onto `dest`
- `ErrUnauthorized`: unauthorized
- `ErrInternalServerError`: internal server error

Delete

`delete` deletes an item from the database that matches the key provided.

`async delete(key)`

Parameters

- **key** (required) – Accepts: `string`
  - Description: the key of which item is to be deleted.

Code Example

```
const res = await db.delete(one)
```

Returns

Always returns a promise which resolves to `null`, even if the key does not exist.

```
delete(key: str)
```

Parameters

- **key** (required) – Accepts: `str`
  - Description: the key of the item that is to be deleted.

Code Example

```
res = db.delete(one)
```

Returns

Always returns `None`, even if the key does not exist.

`Delete(key string) error`

Parameters

- **key**: the key of the item to be deleted #### Code Example

```
// delete item
// returns a nil error if item was not found
err := db.Delete(dakjkfa)
if err != nil {
  fmt.Println(failed to delete item:, err)
}
```

Returns

Returns an `error`. A `nil` error is returned if no item was found with provided `key`. Possible error values:

- `ErrUnauthorized`: unauthorized
- `ErrInternalServerError`: internal server error

Insert

The `insert` method inserts a single item into a **Base**, but is unique from `put` in that it will raise an error if the `key` already exists in the database.

`insert` is roughly 2x slower than `put`.

`async insert(data, key = null, options = null)`

Parameters

- **data** (required) – Accepts: `object` (serializable), `string`, `number`, `boolean` and `array`.
  - Description: The data to be stored.
- **key** (optional) – Accepts: `string` and `null`
  - Description: the key (aka ID) to store the data under. Will be auto generated if not provided.
- **options** (optional) - Accepts: `object`, `null` or `undefined`

  ```
  {
    expireIn: number,
    expireAt: Date
  }
  ```

  - Description: Optional parameters.
    - **expireIn** : item will expire in `expireIn` seconds after a successfull insert operation, see also expiring items.
    - **expireAt** : item will expire at `expireAt` date, see also expiring items.

Code Example

```
// will succeed, a key will be auto-generated
const res1 = await db.insert('hello, world');

// will succeed.
```

```
const res2 = await db.insert({message: 'hello, world'}, 'greeting1');

// will raise an error as key greeting1 already existed.
const res3 = await db.insert({message: 'hello, there'}, 'greeting1');

// expire item in 300 seconds
await db.insert({message: 'will be deleted'}, 'temp_key', {expireIn: 300})

// expire at date
await db.insert({message: 'will be deleted'}, 'temp_key_2', {expireAt: new
Date('2023-01-01T00:00:00')})
```

Returns

Returns a promise which resolves to the item on a successful insert, and throws an error if the key already exists.

```
insert(
  data: typing.Union[dict, list, str, int, float, bool],
  key: str = None,
  *,
  expire_in: int = None,
  expire_at: typing.Union[int, float, datetime.datetime] = None
)
```

Parameters

- **data** (required) – Accepts: `dict`, `str`, `int`, `float`, `bool` and `list`.
  - Description: The data to be stored.
- **key** (optional) – Accepts: `str` and `None`
  - Description: the key (aka ID) to store the data under. Will be auto generated if not provided.
- **expire_in** (optional) - Accepts: `int` and `None`
  - Description: seconds after which the item will expire in, see also expiring items
- **expire_at** (optional) - Accepts: `int`, `float`, `datetime.datetime` and `None`
  - Description: time at which the item will expire in, can provide the timestamp directly(`int` or `float`) or a datetime.datetime object, see also expiring items

Code Example

```
# will succeed, a key will be auto-generated
db.insert(hello, world)

# will succeed.
db.insert({message: hello, world}, greeting1)

# will raise an error as key greeting1 already existed.
db.insert({message: hello, there}, greeting1)

# expiring items
# expire in 300 seconds
db.insert({message: will be deleted}, temp_greeting, expire_in=300)

# expire at date
expire_at = datetime.datetime.fromisoformat(2023-01-01T00:00:00)
db.insert({message: will_be_deleted}, temp_greeting2, expire_at=expire_at)
```

Returns

Returns the item on a successful insert, and throws an error if the key already exists.

= 1.0.0', value: 'new', },] }>

`Insert(item interface{}) (string, error)`

Parameters

- **item** : similar to `item` parameter to `Put`

Code Example

```
import (
    fmt
    github.com/deta/deta-go
)

type User struct{
    Key string `json:key` // json struct tag 'key' used to denote the key

    Username string `json:username`
    Active bool `json:active`
    Age int `json:age`
    Likes []string `json:likes`
}

func main(){
    // error ignored for brevity
    d, _ := deta.New(project key)
    db, _ := d.NewBase(users)

    // a user
    u := &User{
        Key: kasdlj1,
        Username: jimmy,
        Active: true,
        Age: 20,
        Likes: []string{ramen},
    }

    // insert item in the database
    key, err := db.Insert(u)
    if err != nil {
        fmt.Println(failed to insert item:, err)
        return
    }
    fmt.Println(Successfully inserted item with key:, key)
}
```

`Insert(item interface{}) (string, error)`

Parameters

- **item** : similar to `item` parameter to `Put`

Code Example

```
import (
    fmt

    github.com/deta/deta-go/deta
    github.com/deta/deta-go/service/base
)

type User struct {
    Key      string   `json:key` // json struct tag 'key' used to denote the key

    Username string   `json:username`
    Active   bool     `json:active`
    Age      int      `json:age`
    Likes    []string `json:likes`
}

func main() {
    d, err := deta.New(deta.WithProjectKey(project_key))
    if err != nil {
        fmt.Println(failed to init new Deta instance:, err)
        return
    }

    db, err := base.New(d, users)
    if err != nil {
        fmt.Println(failed to init new Base instance:, err)
    }

    u := &User{
        Key:      kasdlj1,
        Username: jimmy,
        Active:   true,
        Age:      20,
        Likes:    []string{ramen},
    }

    // insert item in the database
    key, err := db.Insert(u)
    if err != nil {
        fmt.Println(failed to insert item:, err)
        return
    }
    fmt.Println(Successfully inserted item with key:, key)
}
```

Returns

Returns the `key` of the item inserted and an `error`. Possible error values:

- `ErrConflict` : if item with provided `key` already exists
- `ErrBadItem`: bad item, if item is of unexpected type
- `ErrBadRequest`: item caused a bad request response from the server
- `ErrUnauthorized`: unauthorized
- `ErrInternalServerError`: internal server error

Put Many

The Put Many method puts up to 25 items into a Base at once on a single call.

`async putMany(items, options)`

Parameters

- **items** (required) – Accepts: `Array` of items, where each item can be an `object` (serializable), `string`, `number`, `boolean` or `array`.
  - Description: The list of items to be stored.
- **options** (optional) - Accepts: `object`, `null` or `undefined`

```
{
  expireIn: number,
  expireAt: Date
}
```

  - Description: Optional parameters.
    - **expireIn** : item will expire in `expireIn` seconds after a successfull put operation, see also expiring items.
    - **expireAt** : item will expire at `expireAt` date, see also expiring items.

Code Example

```
await db.putMany([
  {name: Beverly, hometown: Copernicus City, key: one}, // key provided
  dude, // key auto-generated
  [Namaskāra, marhabaan, hello, yeoboseyo] // key auto-generated
]);

// putMany with expire in 300 seconds
await db.putMany(
  [
    {key: temp-1, name: test-1},
    {key: temp-2, name: test-2},
  ],
  {expireIn: 300}
);

// putMany with expire at
await db.putMany(
  [
    {key: temp-1, name: test-1},
    {key: temp-2, name: test-2},
  ],
  {expireAt: new Date('2023-01-01T00:00:00')}
);
```

Returns

Returns a promise which resolves to the put items on a successful insert, and throws an error if you attempt to put more than 25 items.

```
{
    processed: {
        items: [
            {
                hometown: Copernicus City,
                key: one,
                name: Beverly
            },
            {
                key: jyesxxlrezo0,
                value: dude
            },
            {
                key: 5feqybn7lb05,
                value: [
                    Namaskāra,
                    hello,
                    marhabaan,
                    yeoboseyo
                ]
            }
        ]
    }
}
```

```
put_many(
  items: list,
  *,
  expire_in: int = None,
  expire_at: typing.Union[int, float, datetime.datetime] = None,
)
```

Parameters

- **items** (required) – Accepts: `list` of items, where each item can be an `dict` (JSON serializable), `str`, `int`, `bool`, `float` or `list`.
  - Description: The list of items to be stored.
- **expire_in** (optional) - Accepts: `int` and `None`
  - Description: seconds after which the item will expire in, see also expiring items
- **expire_at** (optional) - Accepts: `int`, `float`, `datetime.datetime` and `None`
  - Description: time at which the item will expire in, can provide the timestamp directly(`int` or `float`) or a datetime.datetime object, see also expiring items

Code Example

```
db.put_many([
  {name: Beverly, hometown: Copernicus City, key: one}, // key provided
  dude, // key auto-generated
  [Namaskāra, marhabaan, hello, yeoboseyo] // key auto-generated
])
```

```
# put many to expire in 300 seconds
db.put_many(
  [{key: tmp-1, value: test-1}, {key: tmp-2, value: test-2}],
  expire_in=300,
)

# put many with expire at
expire_at = datetime.datetime.fromisoformat(2023-01-01T00:00:00)
db.put_many(
  [{key: tmp-1, value: test-1}, {key: tmp-2, value: test-2}],
  expire_at=expire_at,
)
```

Returns

Returns a dict with `processed` and `failed` (if any) items .

```
{
    processed: {
        items: [
            {
                hometown: Copernicus City,
                key: one,
                name: Beverly
            },
            {
                key: jyesxxlrezo0,
                value: dude
            },
            {
                key: 5feqybn7lb05,
                value: [
                    Namaskāra,
                    hello,
                    marhabaan,
                    yeoboseyo
                ]
            }
        ]
    }
}
```

<Tabs groupId=go-version defaultValue=new values={[{ label: 'version < 1.0.0', value: 'legacy', }, { label: 'version >= 1.0.0', value: 'new', },] }>

`PutMany(items interface{}) ([]string, error)`

Parameters:

- **items**: a slice of items, each item in the slice similar to the `item` parameter in `Put`

Code Example:

```

import (
    fmt
    github.com/deta/deta-go
)

```

```
type User struct{
    Key string `json:key` // json struct tag 'key' used to denote the key

    Username string `json:username`
    Active bool `json:active`
    Age int `json:age`
    Likes []string `json:likes`
}

func main(){
    // error ignored for brevity
    d, _ := deta.New(project key)
    db, _ := d.NewBase(users)

    // users
    u1 := &User{
        Key: kasdlj1,
        Username: jimmy,
        Active: true,
        Age: 20,
        Likes: []string{ramen},
    }
    u2 := &User{
      Key: askdjf,
      Username: joel,
      Active: true,
      Age: 23,
      Likes: []string{coffee},
    }
    users := []*User{u1, u2}

    // put items in the database
    keys, err := db.PutMany(users)
    if err != nil {
        fmt.Println(failed to put items:, err)
        return
    }
    fmt.Println(Successfully put item with keys:, keys)
}
```

`PutMany(items interface{}) ([]string, error)`

Parameters:

- **items**: a slice of items, each item in the slice similar to the `item` parameter in `Put`

Code Example:

```
import (
    fmt

    github.com/deta/deta-go/deta
    github.com/deta/deta-go/service/base
)

type User struct {
    Key      string   `json:key` // json struct tag 'key' used to denote the key

    Username string   `json:username`
    Active   bool     `json:active`
    Age      int      `json:age`
    Likes    []string `json:likes`
}

func main() {
    d, err := deta.New(deta.WithProjectKey(project_key))
    if err != nil {
        fmt.Println(failed to init new Deta instance:, err)
        return
    }

    db, err := base.New(d, users)
    if err != nil {
        fmt.Println(failed to init new Base instance:, err)
    }

    // users
    u1 := &User{
        Key:      kasdlj1,
        Username: jimmy,
        Active:   true,
        Age:      20,
        Likes:    []string{ramen},
    }
    u2 := &User{
        Key:      askdjf,
        Username: joel,
        Active:   true,
        Age:      23,
        Likes:    []string{coffee},
    }
    users := []*User{u1, u2}

    // put items in the database
    keys, err := db.PutMany(users)
    if err != nil {
        fmt.Println(failed to put items:, err)
        return
    }
    fmt.Println(Successfully put item with keys:, keys)
}
```

Returns

Returns the list of keys of the items stored and an `error`. In case of an error, none of the items are stored. Possible error values:

- `ErrTooManyItems`: if there are more than 25 items
- `ErrBadItem`: bad item/items, one or more item of unexpected type
- `ErrBadRequest`: one or more item caused a bad request response from the server
- `ErrUnauthorized`: unauthorized
- `ErrInternalServerError`: internal server error

Update

`update` updates an existing item from the database.

`async update(updates, key, options)`

Parameters

- **updates** (required) - Accepts: `object` (JSON serializable)
  - Description: a json object describing the updates on the item
- **key** (required) – Accepts: `string`
  - Description: the key of the item to be updated.
- **options** (optional) - Accepts: `object`

  ```
  {
    expireIn: number,
    expireAt: Date
  }
  ```

  - Description: Optional parameters.
    - **expireIn** : item will expire in `expireIn` seconds after a successfull update operation, see also expiring items.
    - **expireAt** : item will expire at `expireAt` date, see also expiring items.

Note for storing numbers

Update operations

- **Set** : `Set` is practiced through normal key-value pairs. The operation changes the values of the attributes provided in the `set` object if the attribute already exists. If not, it adds the attribute to the item with the corresponding value.

- **Increment**: `Increment` increments the value of an attribute. The attribute's value *must be a number*. The util `base.util.increment(value)` should be used to increment the value. The *default value is 1* if not provided and it can also be negative.

- **Append**: `Append` appends to a list. The util `base.util.append(value)` should be used to append the value. The value can be a `primitive type` or an `array`.

- **Prepend**: `Prepend` prepends to a list. The util `base.util.prepend(value)` should be used to prepend the value. The value can be a `primitive type` or an `array`.

- **Trim**: `Trim` removes an attribute from the item, the util `base.util.trim()` should be used as the value of an attribute.

Code Example

Consider we have the following item in a base `const users = deta.Base('users')`:

```
{
  key: user-a,
  username: jimmy,
  profile: {
    age: 32,
```

```
    active: false,
    hometown: pittsburgh
  },
  on_mobile: true,
  likes: [anime],
  purchases: 1
}
```

Then the following update operation :

```
const updates = {
  profile.age: 33, // set profile.age to 33
  profile.active: true, // set profile.active to true
  profile.email: jimmy@deta.sh, // create a new attribute 'profile.email'
  profile.hometown: users.util.trim(), // remove 'profile.hometown'
  on_mobile: users.util.trim(), // remove 'on_mobile'
  purchases: users.util.increment(2), // increment 'purchases' by 2, default value
is 1
  likes: users.util.append(ramen) // append 'ramen' to 'likes', also accepts an
array
}

const res = await db.update(updates, user-a);
```

Results in the following item in the base:

```
{
  key: user-a,
  username: jimmy,
  profile: {
    age: 33,
    active: true,
    email: jimmy@deta.sh
  },
  likes: [anime, ramen],
  purchases: 3
}
```

Returns

If the item is updated, the promise resolves to `null`. Otherwise, an error is raised.

```
update(
  updates: dict,
  key: str,
  *,
  expire_in: int = None,
  expire_at: typing.Union [int, float, datetime.datetime] = None
)
```

Parameters

- **updates** (required) - Accepts: `dict` (JSON serializable)
  - Description: a dict describing the updates on the item
- **key** (required) – Accepts: `string`
  - Description: the key of the item to be updated.

- **expire_in** (optional) - Accepts: `int` and `None`
  - Description: seconds after which the item will expire in, see also expiring items
- **expire_at** (optional) - Accepts: `int`, `float`, `datetime.datetime` and `None`
  - Description: time at which the item will expire in, can provide the timestamp directly(`int` or `float`) or a datetime.datetime object, see also expiring items

Note for storing numbers

Update operations

- **Set** : `Set` is practiced through normal key-value pairs. The operation changes the values of the attributes provided in the `set` dict if the attribute already exists. If not, it adds the attribute to the item with the corresponding value.

- **Increment**: `Increment` increments the value of an attribute. The attribute's value *must be a number*. The util `base.util.increment(value)` should be used to increment the value. The *default value is 1* if not provided and it can also be negative.

- **Append**: `Append` appends to a list. The util `base.util.append(value)` should be used to append the value. The value can be a `primitive type` or a `list`.

- **Prepend**: `Prepend` prepends to a list. The util `base.util.prepend(value)` should be used to prepend the value. The value can be a `primitive type` or a `list`.

- **Trim**: `Trim` removes an attribute from the item, the util `base.util.trim()` should be used as the value of an attribute.

Code Example

Consider we have the following item in a base `users = deta.Base('users')`:

```
{
  key: user-a,
  username: jimmy,
  profile: {
    age: 32,
    active: false,
    hometown: pittsburgh
  },
  on_mobile: true,
  likes: [anime],
  purchases: 1
}
```

Then the following update operation:

```
updates = {
  profile.age: 33,  # set profile.age to 33
  profile.active: True, # set profile.active to true
  profile.email: jimmy@deta.sh, # create a new attribute 'profile.email'
  profile.hometown: users.util.trim(), # remove 'profile.hometown'
  on_mobile: users.util.trim(), # remove 'on_mobile'
  purchases: users.util.increment(2), # increment by 2, default value is 1
  likes: users.util.append(ramen) # append 'ramen' to 'likes', also accepts a list
}

db.update(updates, user-a)
```

Results in the following item in the base:

```
{
  key: user-a,
  username: jimmy,
  profile: {
    age: 33,
    active: true,
    email: jimmy@deta.sh
  },
  likes:[anime, ramen],
  purchases: 3
}
```

Returns

If the item is updated, returns `None`. Otherwise, an exception is raised.

= 1.0.0', value: 'new', },] }>

`Update(key string, updates Updates) error`

Parameters

- **key**: the key of the item to update
- **updates** : updates applied to the item, is of type `deta.Updates` which is a `map[string]interface{}`

Note for storing numbers

Update operations

- **Set** : `Set` is practiced through normal key-value pairs. The operation changes the values of the attributes provided if the attribute already exists. If not, it adds the attribute to the item with the corresponding value.

- **Increment**: `Increment` increments the value of an attribute. The attribute's value *must be a number*. The util `Base.Util.Increment(value interface{})` should be used to increment the value. The value can also be negative.

- **Append**: `Append` appends to a list. The util `Base.Util.Append(value interface{})` should be used to append the value. The value can be a slice.

- **Prepend**: `Prepend` prepends to a list. The util `Base.Util.Prepend(value interface{})` should be used to prepend the value. The value can be a slice.

- **Trim**: `Trim` removes an attribute from the item, the util `Base.Util.Trim()` should be used as the value of an attribute.

Code Example

Consider we have the following item in a base `users` :

```
{
  key: user-a,
  username: jimmy,
  profile: {
    age: 32,
    active: false,
    hometown: pittsburgh
  },
  on_mobile: true,
  likes: [anime],
  purchases: 1
}
```

Then the following update operation :

```
// define the updates
updates := deta.Updates{
  profile.age: 33, // set profile.age to 33
  profile.active: true, // set profile.active to true
  profile.email: jimmy@deta.sh, // create a new attribute 'profile.email'
  profile.hometown: users.Util.Trim(), // remove 'profile.hometown'
  on_mobile: users.Util.Trim(), // remove 'on_mobile'
  purchases: users.Util.Increment(2), // increment 'purchases' by 2
  likes: users.Util.Append(ramen) // append 'ramen' to 'likes', also accepts a
slice
}

// update
err := users.Util.Update(user-a, updates);
```

Results in the following item in the base:

```
{
  key: user-a,
  username: jimmy,
  profile: {
    age: 33,
    active: true,
    email: jimmy@deta.sh
  },
  likes: [anime, ramen],
  purchases: 3
}
```

`Update(key string, updates Updates) error`

Parameters

- **key**: the key of the item to update
- **updates** : updates applied to the item, is of type `base.Updates` which is a `map[string]interface{}`

Note for storing numbers

Update operations

- **Set** : `Set` is practiced through normal key-value pairs. The operation changes the values of the attributes provided if the attribute already exists. If not, it adds the attribute to the item with the corresponding value.
- **Increment**: `Increment` increments the value of an attribute. The attribute's value *must be a number*. The util `Base.Util.Increment(value interface{})` should be used to increment the value. The value can also be negative.
- **Append**: `Append` appends to a list. The util `Base.Util.Append(value interface{})` should be used to append the value. The value can be a slice.
- **Prepend**: `Prepend` prepends to a list. The util `Base.Util.Prepend(value interface{})` should be used to prepend the value. The value can be a slice.
- **Trim**: `Trim` removes an attribute from the item, the util `Base.Util.Trim()` should be used as the value of an attribute.

Code Example

Consider we have the following item in a base `users` :

```
{
  key: user-a,
  username: jimmy,
  profile: {
    age: 32,
    active: false,
    hometown: pittsburgh
  },
  likes: [anime],
  purchases: 1
}
```

Then the following update operation :

```
import (
    fmt

    github.com/deta/deta-go/deta
    github.com/deta/deta-go/service/base
)

type Profile struct {
    Active   bool   `json:active`
    Age      int    `json:age`
    Hometown string `json:hometown`
}

type User struct {
    Key       string   `json:key` // json struct tag 'key' used to denote the key

    Username  string   `json:username`
    Profile   *Profile `json:profile`
    Purchases int      `json:purchases`
    Likes     []string `json:likes`
}

func main() {
    d, err := deta.New(deta.WithProjectKey(project_key))
    if err != nil {
        fmt.Println(failed to init new Deta instance:, err)
        return
    }

    db, err := base.New(d, users)
    if err != nil {
        fmt.Println(failed to init new Base instance:, err)
    }

    // define the updates
    updates := base.Updates{
        profile.age: 33, // set profile.age to 33
        profile.active: true, // set profile.active to true
        profile.hometown: db.Util.Trim(), // remove 'profile.hometown'
        purchases: db.Util.Increment(2), // increment 'purchases' by 2
        likes: db.Util.Append(ramen), // append 'ramen' to 'likes', also accepts a
slice
    }
    // update
    err = db.Update(user-a, updates)
    if err != nil {
        fmt.Println(failed to update)
        return
    }
}
```

Results in the following item in the base:

```
{
  key: user-a,
  username: jimmy,
  profile: {
    age: 33,
    active: true,
  },
  likes: [anime, ramen],
  purchases: 3
}
```

Returns

Returns an `error`. Possible error values:

- `ErrBadRequest`: the update operation caused a bad request response from the server
- `ErrUnauthorized`: unauthorized
- `ErrInternalServerError`: internal server error

Fetch

Fetch retrieves a list of items matching a query. It will retrieve everything if no query is provided.

A query is composed of a single query object or a list of queries.

In the case of a list, the indvidual queries are OR'ed.

<Tabs groupId=js-version defaultValue=new values={[{ label: 'version < 1.0.0', value: 'legacy', }, { label: 'version >= 1.0.0', value: 'new', },] }>

`async fetch(query, pages=10, buffer=null)`

Parameters

- **query**: is a single query object or list of queries. If omitted, you will get all the items in the database (up to 1mb).
- **pages**: how many pages of items should be returned.
- **buffer**: the number of items which will be returned for each iteration (aka page) on the return iterable. This is useful when your query is returning more than 1mb of data, so you could buffer the results in smaller chunks.

Code Example

For the examples, let's assume we have a **Base** with the following data:

```
[
  {
    key: key-1,
    name: Wesley,
    age: 27,
    hometown: San Francisco,
  },
  {
    key: key-2,
    name: Beverly,
    age: 51,
    hometown: Copernicus City,
  },
  {
    key: key-3,
    name: Kevin Garnett,
    age: 43,
    hometown: Greenville,
  }
]
```

```
const {value: myFirstSet} = await db.fetch({age?lt: 30}).next();
const {value: mySecondSet} = await db.fetch([
  { age?gt: 50 },
  { hometown: Greenville }
]).next();
```

... will come back with following data:

`myFirstSet`:

```
[
  {
    key: key-1,
    name: Wesley,
    age: 27,
    hometown: San Francisco,
  }
]
```

`mySecondSet`:

```
[
  {
    key: key-2,
    name: Beverly,
    age: 51,
    hometown: Copernicus City,
  },
  {
    key: key-3,
    name: Kevin Garnett,
    age: 43,
    hometown: Greenville,
  },
]
```

Returns

A promise which resolves to a generator of objects that meet the `query` criteria.

The total number of items will not exceed the defined using `buffer` and `pages. Max. number of items

Iterating through the generator yields arrays containing objects, each array of max length `buffer`.

Example using buffer, pages

```
const foo = async (myQuery, bar) => {

  items = db.fetch(myQuery, 10, 20) // items is up to the limit length (10*20)

  for await (const subArray of items) // each subArray is up to the buffer length,
20
    bar(subArray)
}
```

`async fetch(query, options)`

Parameters

- **query**: is a single query object (`dict`) or list of queries. If omitted, you will get all the items in the database (up to 1mb).
- **options**: optional params:
  - `limit`: the limit of the number of items you want to retreive, min value `1` if used.

   - `last` : the last key seen in a previous paginated response, provide this in a subsequent call to fetch further items.

> Upto 1 MB of data is retrieved before filtering with the query. Thus, in some cases you might get an empty list of items but still the `last` key evaluated in the response.
>
> To apply the query through all the items in your base, you have to call fetch until `last` is empty.

Returns

A promise which resolves to an object with the following attributes:

- `count` : The number of items in the response.
- `last` : The last key seen in the fetch response. If `last` is not `undefined` further items are to be retreived.
- `items` : The list of items retreived.

Example

For the examples, let's assume we have a **Base** with the following data:

```
[
  {
    key: key-1,
    name: Wesley,
    age: 27,
    hometown: San Francisco,
  },
  {
    key: key-2,
    name: Beverly,
    age: 51,
    hometown: Copernicus City,
  },
  {
    key: key-3,
    name: Kevin Garnett,
    age: 43,
    hometown: Greenville,
  }
]
```

```
const { items: myFirstSet } = await db.fetch({age?lt: 30});
const { items: mySecondSet } = await db.fetch([
  { age?gt: 50 },
  { hometown: Greenville }
])
```

... will come back with following data:

`myFirstSet`:

```
[
  {
    key: key-1,
    name: Wesley,
    age: 27,
    hometown: San Francisco,
  }
]
```

`mySecondSet`:

```
[
  {
    key: key-2,
    name: Beverly,
    age: 51,
    hometown: Copernicus City,
  },
  {
    key: key-3,
    name: Kevin Garnett,
    age: 43,
    hometown: Greenville,
  },
]
```

Fetch All Items

```
let res = await db.fetch();
let allItems = res.items;

// continue fetching until last is not seen
while (res.last){
  res = await db.fetch({}, {last: res.last});
  allItems = allItems.concat(res.items);
}
```

<Tabs groupId=py-version defaultValue=new values={[{ label: 'version < 1.0.0', value: 'legacy', }, { label: 'version >= 1.0.0', value: 'new', },] }>

`fetch(query=None, buffer=None, pages=10):`

Parameters

- **query**: is a single query object (`dict`) or list of queries. If omitted, you will get all the items in the database (up to 1mb).
- **pages**: how many pages of items should be returned.
- **buffer**: the number of items which will be returned for each iteration (aka page) on the return iterable. This is useful when your query is returning more 1mb of data, so you could buffer the results in smaller chunks.

Code Example

For the examples, let's assume we have a **Base** with the following data:

```
[
  {
    key: key-1,
    name: Wesley,
    age: 27,
    hometown: San Francisco,
  },
  {
    key: key-2,
    name: Beverly,
    age: 51,
    hometown: Copernicus City,
  },
  {
    key: key-3,
    name: Kevin Garnett,
    age: 43,
    hometown: Greenville,
  }
]
```

```
my_first_set = next(db.fetch({age?lt: 30}))
my_second_set = next(db.fetch([{age?gt: 50}, {hometown: Greenville}]))
```

… will come back with following data:

`my_first_set`:

```
[
  {
    key: key-1,
    name: Wesley,
    age: 27,
    hometown: San Francisco,
  }
]
```

`my_second_set`:

```
[
  {
    key: key-2,
    name: Beverly,
    age: 51,
    hometown: Copernicus City,
  },
  {
    key: key-3,
    name: Kevin Garnett,
    age: 43,
    hometown: Greenville,
  },
]
```

Returns

A generator of objects that meet the `query` criteria.

The total number of items will not exceed the defined using `buffer` and `pages. Max. number of items

Iterating through the generator yields lists containing objects, each list of max length `buffer`.

Example using buffer, pages

```
def foo(my_query, bar):
  items = db.fetch(my_query, pages=10, buffer=20) # items is up to the limit
length (10*20)

  for sub_list in items: # each sub_list is up to the buffer length, 10
    bar(sub_list)
```

`fetch(query=None, limit=1000, last=None):`

Parameters

- **query**: is a single query object (`dict`) or list of queries. If omitted, you will get all the items in the database (up to 1mb or max 1000 items).
- **limit**: the limit of the number of items you want to retreive, min value `1` if used
- **last**: the last key seen in a previous paginated response

Upto 1 MB of data is retrieved before filtering with the query. Thus, in some cases you might get an empty list of items but still the `last` key evaluated in the response.

To apply the query through all the items in your base, you have to call fetch until `last` is empty.

Returns

Returns an instance of a `FetchResponse` class which has the following properties.

- `count` : The number of items in the response.
- `last` : The last key seen in the fetch response. If `last` is not `None` further items are to be retreived
- `items` : The list of items retreived.

Code Example

For the examples, let's assume we have a **Base** with the following data:

```
[
  {
    key: key-1,
    name: Wesley,
    age: 27,
    hometown: San Francisco,
  },
  {
    key: key-2,
    name: Beverly,
    age: 51,
    hometown: Copernicus City,
  },
  {
    key: key-3,
    name: Kevin Garnett,
    age: 43,
    hometown: Greenville,
  }
]
```

```
first_fetch_res = db.fetch({age?lt: 30})
second_fetch_res = db.fetch([{age?gt: 50}, {hometown: Greenville}])
```

… will come back with following data:

`first_fetch_res.items` :

```
[
  {
    key: key-1,
    name: Wesley,
    age: 27,
    hometown: San Francisco,
  }
]
```

`second_fetch_res.items` :

```
[
  {
    key: key-2,
    name: Beverly,
    age: 51,
    hometown: Copernicus City,
  },
  {
    key: key-3,
    name: Kevin Garnett,
    age: 43,
    hometown: Greenville,
  },
]
```

Fetch All Items

```
res = db.fetch()
all_items = res.items

# fetch until last is 'None'
while res.last:
  res = db.fetch(last=res.last)
  all_items += res.items
```

= 1.0.0', value: 'new', },] }>

`Fetch(i *FetchInput) error`

Parameters

- **i**: is a pointer to a `FetchInput`

```
// FetchInput input to Fetch operation
type FetchInput struct {
  // filters to apply to items
  // A nil value applies no queries and fetches all items
  Q Query
  // the destination to store the results
  Dest interface{}
  // the maximum number of items to fetch
  // value of 0 or less applies no limit
  Limit int
  // the last key evaluated in a paginated response
  // leave empty if not a subsequent fetch request
  LastKey string
}
```

  - `Q` : fetch query, is of type `deta.Query` which is a `[]map[string]interface{}`
  - `Dest` : the results will be stored into the value pointed by `Dest`
  - `Limit` : the maximum number of items to fetch, value of `0` or less applies no limit
  - `LastKey` : the last key evaluated in a paginated response, leave empty if not a subsequent fetch request

Code Example

```
import (
    github.com/deta/deta-go
)

type User struct {
    Key string `json:key`
    Name string `json:name`
    Age int `json:age`
    Hometown string `json:hometown`
}

func main(){
    // errors ignored for brevity
    d, _ := deta.New(project key)
    db, _ := deta.NewBase(users)

    // query to get users with age less than 30
    query := deta.Query{
      {age?lt: 50},
    }

    // variabe to store the results
    var results []*User

    // fetch items
    _, err := db.Fetch(&deta.FetchInput{
      Q: query,
      Dest: &results,
    })
    if err != nil {
        fmt.Println(failed to fetch items:, err)
    }
}
```

… `results` will have the following data:

```
[
  {
    key: key-1,
    name: Wesley,
    age: 27,
    hometown: San Francisco,
  },
  {
    key: key-3,
    name: Kevin Garnett,
    age: 43,
    hometown: Greenville,
  },
]
```

Paginated example

```
import (
    github.com/deta/deta-go
)

type User struct {
    Key string `json:key`
    Name string `json:name`
    Age int `json:age`
    Hometown string `json:hometown`
}

func main(){
    // errors ignored for brevity
    d, _ := deta.New(project key)
    db, _ := deta.NewBase(users)

    // query to get users with age less than 30
    query := deta.Query{
      {age?lt: 50},
    }

    // variabe to store the results
    var results []*User

    // variable to store the page
    var page []*User

    // fetch input
    i := &deta.FetchInput{
      Q: query,
      Dest: &page,
      Limit: 1, // limit provided so each page will only have one item
    }

    // fetch items
    lastKey, err := db.Fetch(i)
    if err != nil {
        fmt.Println(failed to fetch items:, err)
        return
    }

    // append page items to results
    results = append(allResults, page...)

    // get all pages
    for lastKey != {
      // provide the last key in the fetch input
      i.LastKey = lastKey

      // fetch
      lastKey, err := db.Fetch(i)
      if err != nil {
          fmt.Println(failed to fetch items:, err)
          return
      }

      // append page items to all results
      results = append(allResults, page...)
    }
}
```

`Fetch(i *FetchInput) error`

Parameters

- **i**: is a pointer to a `FetchInput`

```
// FetchInput input to Fetch operation
type FetchInput struct {
  // filters to apply to items
  // A nil value applies no queries and fetches all items
  Q Query
  // the destination to store the results
  Dest interface{}
  // the maximum number of items to fetch
  // value of 0 or less applies no limit
  Limit int
  // the last key evaluated in a paginated response
  // leave empty if not a subsequent fetch request
  LastKey string
}
```

  - `Q` : fetch query, is of type `deta.Query` which is a `[]map[string]interface{}`
  - `Dest` : the results will be stored into the value pointed by `Dest`
  - `Limit` : the maximum number of items to fetch, value of `0` or less applies no limit
  - `LastKey` : the last key evaluated in a paginated response, leave empty if not a subsequent fetch request

Code Example

```
import (
    fmt

    github.com/deta/deta-go/deta
    github.com/deta/deta-go/service/base
)

type User struct {
  Key string `json:key`
  Name string `json:name`
  Age int `json:age`
  Hometown string `json:hometown`
}

func main() {
    d, err := deta.New(deta.WithProjectKey(project_key))
    if err != nil {
        fmt.Println(failed to init new Deta instance:, err)
        return
    }

    db, err := base.New(d, users)
    if err != nil {
        fmt.Println(failed to init new Base instance:, err)
    }

    // query to get users with age less than 30
    query := base.Query{
        {age?lt: 50},
    }

    // variabe to store the results
    var results []*User

    // fetch items
    _, err = db.Fetch(&base.FetchInput{
        Q:     query,
        Dest: &results,
    })
    if err != nil {
        fmt.Println(failed to fetch items:, err)
    }
}
```

… `results` will have the following data:

```
[
  {
    key: key-1,
    name: Wesley,
    age: 27,
    hometown: San Francisco,
  },
  {
    key: key-3,
    name: Kevin Garnett,
    age: 43,
    hometown: Greenville,
  },
]
```

Paginated example

```
import (
    fmt

    github.com/deta/deta-go/deta
    github.com/deta/deta-go/service/base
)

type User struct {
    Key      string   `json:key` // json struct tag 'key' used to denote the key

    Username string   `json:username`
    Active   bool     `json:active`
    Age      int      `json:age`
    Likes    []string `json:likes`
}

func main() {
    d, err := deta.New(deta.WithProjectKey(project_key))
    if err != nil {
        fmt.Println(failed to init new Deta instance:, err)
        return
    }

    db, err := base.New(d, users)
    if err != nil {
        fmt.Println(failed to init new Base instance:, err)
    }

    // query to get users with age less than 30
    query := base.Query{
        {age?lt: 50},
    }

    // variabe to store the results
    var results []*User

    // variable to store the page
    var page []*User

    // fetch input
    i := &base.FetchInput{
        Q:      query,
        Dest:  &page,
        Limit: 1, // limit provided so each page will only have one item
    }

    // fetch items
    lastKey, err := db.Fetch(i)
    if err != nil {
        fmt.Println(failed to fetch items:, err)
        return
    }

    // append page items to results
    results = append(results, page...)

    // get all pages
    for lastKey !=  {
        // provide the last key in the fetch input
        i.LastKey = lastKey
        // fetch
        lastKey, err = db.Fetch(i)
        if err != nil {
            fmt.Println(failed to fetch items:, err)
            return
        }

        // append page items to all results
        results = append(results, page...)
    }
}
```

Returns

Returns an `error`. Possible error values:

- `ErrBadDestination`: bad destination, results could not be stored onto `dest`
- `ErrBadRequest`: the fetch request caused a bad request response from the server
- `ErrUnauthorized`: unauthorized
- `ErrInternalServerError`: internal server error

Issues

If you run into any issues, consider reporting them in our Github Discussions.

Go to TOC

The deta cli authenticates in two ways:

Login From The Browser

Use deta command `deta login` to login from the web browser. It will open the login page in a web browser.

> If the login page could not be opened automatically for some reason, the cli will display the login url. Open the link in a browser for the login to complete.

Deta Access Tokens

The deta cli also authenticates with deta access tokens. You can create an access token under the `Settings` tab in your dashboard(https://web.deta.sh). The access tokens are **valid for a year**.

> The access token can only be retreived once after creation. Please, store it in a safe place after the token has been created.

Token Provider Chain

The access token (called *Access Token* when you create the token from the UI) can be set up to be used by the cli in the following ways in order of preference:

- `DETA_ACCESS_TOKEN` environment variable:
    - provide the access token through the cli's environment under the environment variable `DETA_ACCESS_TOKEN`
- `$HOME/.deta/tokens` file:
    - create a file called `tokens` in `$HOME/.deta/`
    - provide the token in the field `deta_access_token` in the file as *json* :

```
{
    deta_access_token: your_access_token
}
```

Using Deta in GitPod with Access Tokens

To use the Deta CLI in GitPod, first install the Deta CLI to your GitPod terminal:

```
curl -fsSL https://get.deta.dev/cli.sh | sh
```

Then add the `deta` command to the path of the Gitpod environment:

```
source ~/.bashrc
```

Finally, add your authentication token to the GitPod environment, which will authenticate the Deta CLI commands against our backend:

```
export DETA_ACCESS_TOKEN=<access_token_here>
```

You are now free to use the `deta` command within GitPod!

Using Deta in GitHub Actions with Access Tokens

Use this ready-to-use GitHub action to deploy your app to Deta.

Issues

If you run into any issues, consider reporting them in our Github Discussions.

Go to TOC

Summary

deta is a CLI for managing Deta Micros.

To install the CLI, check out Installing the CLI.

Commands

`deta login` - Trigger the login process for the Deta CLI.

`deta version` - Print the Deta version

`deta projects` - List Deta projects

`deta new` - Create a new Deta Micro

`deta deploy` - Deploy new code to a Deta micro

`deta details` - Get detailed information about a Deta Micro

`deta watch` - Auto-deploy local changes in real time to your Micro

`deta auth` - Change auth settings for a Deta Micro (public access, api keys)

`deta pull` - Pull the live code from a Deta Micro onto your local machine

`deta clone` - Clone a Deta Micro onto your local machine

`deta update` - Update a Deta Micro

`deta visor` - Change the Visor settings for a Deta Micro

`deta cron` - Change cron settings for a Deta Micro

deta login

Login to deta via the CLI. This is necessary before using any other commands.

Command

```
deta login [flags]
```

Flags

```
  -h, --help   help for login
```

deta version

Print the Deta version

Command

```
deta version [flags]
deta version [command]
```

Flags

```
  -h, --help   help for version
```

Version commands

```
  upgrade
```

deta version upgrade

Upgrade deta cli version

Command

```
  deta version upgrade [--version <version>]
```

Flags

```
  -h, --help             help for upgrade
  -v, --version string   version number
```

Examples

```
1. deta version upgrade

Upgrade cli to latest version.

2. deta version upgrade --version v1.0.0

Upgrade cli to version 'v1.0.0'.
```

deta projects

List deta projects

Command

```
deta projects [flags]
```

Flags

```
  -h, --help   help for projects
```

deta new

`deta new` creates a new Micro(server) from the Deta cli

Command

```
deta new [flags] [path]
```

Flags

```
  -h, --help              help for new
      --name string       name of the new micro
  -n, --node              create a micro with node (node14.x) runtime
      --project string    project to create the micro under
  -p, --python            create a micro with python (python 3.9) runtime
      --runtime string    create a micro with the specified runtime
                          Python: python3.7, python3.9
                          Node: node12, node14
```

Examples

```
1. deta new

Create a new deta micro from the current directory with an entrypoint file (either
'main.py' or 'index.js') already present in the directory.

2. deta new --runtime python3.7

Create a new python deta micro from the current directory with an entrypoint file
('main.py') already present in the directory and runtime python3.7.

3. deta new my-micro

Create a new deta micro from './my-micro' directory with an entrypoint file
(either 'main.py' or 'index.js') already present in the directory.

4. deta new --node my-node-micro

Create a new deta micro with the node runtime in the directory './my-node-micro'.
'./my-node-micro' must not contain a python entrypoint file ('main.py') if
directory is already present.

5. deta new --python --name my-github-webhook webhooks/github-deta

Create a new deta micro with the python runtime, name 'my-github-webhook' and in
directory 'webhooks/github-deta'.
'./my-node-micro' must not contain a node entrypoint file ('index.js') if
directory is already present.
```

Notes

- The `path` defaults to the current working directory if not provided.
- `deta new` will first check `path` for a `main.py` or `index.js` file. If one is found, `deta new` will bootstrap the Micro runtime based on the local file. If `path` is an empty directory, a runtime (with `--node` or `--python` or `--runtime`) must be provided and `deta` will create a starter app in `path`.
- If `path` is not an empty directory and does not have an entrypoint file (either `main.py` or `index.js`) a `name` must be provided, under which `deta` will create a micro with a starter app.

deta deploy

Deploy your local code (and dependencies) to your Deta Micro.

Command

```
deta deploy [flags] [path]
```

Flags

```
   -h, --help   help for deploy
```

Examples

```
1. deta deploy

Deploy a deta micro rooted in the current directory.

2. deta deploy micros/my-micro-1

Deploy a deta micro rooted in 'micros/my-micro-1' directory.
```

Notes

`deta deploy` will ignore the following files and folders by default when deploying a micro:

- all files and folders with names starting with a `.` (both in unix and windows systems)
- vim swap files
- `node_modules` directory for `node` runtime
- `__pycache__` directory for `python` runtime
- `env` and `venv` directories for `python` runtime
- `.pyc` and `.rst` files for `python` runtimes

Deta Ignore

`deta deploy` respects a `.detaignore` file. You can specify paths additional to the default ignore paths in this file.

For eg, a `.detaignore` file with the following content will ignore the `test` and `docs` paths when deploying your micro.

```
test
docs
```

You can also use the `!` character to include paths that are ignored by default by the cli.

For eg, a `.detaignore` file with the following content will not ignore the `.env` file when deploying your micro.

```
!.env
```

deta details

Get detailed information about a specific Deta micro.

Command

```
deta details [flags] [path]
```

Flags

```
  -h, --help   help for details
```

deta watch

Auto-deploy locally saved changes in real time to your Deta micro.

Command

```
deta watch [path] [flags]
```

Flags

```
  -h, --help   help for watch
```

deta auth

Change auth settings for a Deta Micro

Command

```
deta auth [auth_command] [flags]
```

Auth Commands

```
  disable, enable, create-api-key, delete-api-key
```

Flags

```
  -h, --help   help for auth
```

deta auth disable

Disable Deta's *Http Auth* for a Deta Micro (this makes the HTTP endpoint publicly accesible).

Command

```
deta auth disable [flags]
```

Flags

```
  -h, --help   help for disable
```

deta auth enable

Enable Deta's *Http Auth* for a Deta Micro (a Micro's endpoint will require authenticated access or via a valid api key).

Command

```
deta auth enable [flags]
```

Flags

```
  -h, --help   help for enable
```

deta auth create-api-key

Create an API key for a Deta Micro

Command

```
deta auth create-api-key [flags]
```

Flags

```
  -d, --desc string      api-key description
  -h, --help             help for create-api-key
  -n, --name string      api-key name, required
  -o, --outfile string   file to save the api-key
```

Examples

```
1. deta auth create-api-key --name agent1 --desc api key for agent 1

Create an api key with name 'agent1' and description 'api key for agent 1'

2. deta auth create-api-key --name agent1 --outfile agent_1_key.txt

Create an api key with name 'agent1' and save it to file 'agent_1_key.txt'
```

deta auth delete-api-key

Delete an API key for a Deta Micro

Command

```
deta auth delete-api-key [flags]
```

Flags

```
    -d, --desc string   api-key description
    -h, --help          help for delete-api-key
    -n, --name string   api-key name, required
```

Examples

```
1. deta auth delete-api-key --name agent1

Delete api key with name 'agent1'
```

deta pull

Pull the latest deployed code of a Deta Micro to your local machine.

Command

```
deta pull [flags]
```

Flags

```
    -f, --force    force overwrite of existing files
    -h, --help     help for pull
```

Examples

```
1. deta pull

Pull latest changes of deta micro present in the current directory.
Asks for approval before overwriting the files in the current directory.

2. deta pull --force

Force pull latest changes of deta micro present in the current directory.
Overwrites the files present in the current directory.
```

deta clone

Clone a deta micro

Command

```
deta clone [path] [flags]
```

Flags

```
    -h, --help              help for clone
        --name string       deta micro name
        --project string    deta project
```

Examples

```
1. deta clone --name my-micro

Clone latest deployment of micro 'my-micro' from 'default' project to directory
'./my-micro'.

2. deta clone --name my-micro --project my-project micros/my-micro-dir

Clone latest deployment of micro 'my-micro' from project 'my-project' to directory
'./micros/my-micro-dir'.
'./micros/my-micro-dir' must be an empty directory if it already exists.
```

deta update

Update a deta micro

Command

Update a Deta Micro's name or environment variables.

```
deta update [flags]
```

Flags

```
  -e, --env string     path to env file
  -h, --help           help for update
  -n, --name string    new name of the micro
  -r, --runtime string new runtime of the micro
```

Examples

```
1. deta update --name a-new-name

Update the name of a deta micro with a new name a-new-name.

2. deta update --env env-file

Update the enviroment variables of a deta micro from the file 'env-file'.
File 'env-file' must have env vars of format 'key=value'.

3. deta update --runtime nodejs14

Update the runtime of the micro to nodejs14.x.
```

deta visor

Change the Visor settings for a Deta Micro.

If *Deta Visor* is enabled, Deta will log all incoming requests to and responses from a Deta Micro, letting you inspect, edit, and replay these request / response pairs.

To access a Micro's visor, navigate to:

`https://web.deta.sh/home/:username/:projectName/micros/:microName/visor`

Command

Change visor settings for a deta micro

```
deta visor [visor_command] [flags]
```

Visor Commands

```
  open, enable, disable
```

Flags

```
  -h, --help   help for visor
```

deta visor open

Open Micro's visor page in the browser.

Command

```
deta visor open [flags]
```

Flags

```
  -h, --help help for open
```

deta visor enable

Enable Visor for a Deta Micro

Command

```
deta visor enable [flags]
```

Flags

```
  -h, --help   help for enable
```

deta visor disable

Disable Visor for a deta micro

Command

```
deta visor disable [flags]
```

Flags

```
  -h, --help   help for disable
```

deta run

Run a Deta Micro from the cli

Command

```
deta run [flags] [action] [-- <input args>]
```

Flags

```
  -h, --help   help for run
  -l, --logs   show micro logs
```

Examples

```
1. deta run -- --name Jimmy --age 33 -active

Run deta micro with the following input:
{
    name: Jimmy,
    age: 33,
    active: true
}

2. deta run --logs test -- --username admin

Run deta micro and show micro logs with action 'test' and the following input:
{
    username: admin
}

3. deta run delete -- --emails jimmy@deta.sh --emails joe@deta.sh

Run deta micro with action 'delete' and the following input:
{
    emails: [jimmy@deta.sh, joe@deta.sh]
}

See https://docs.deta.sh for more examples and details.
```

deta cron

Change cron settings for a deta micro

Command

```
deta cron [cron_command] [flags]
```

Cron Commands

```
  set, remove
```

Flags

```
  -h, --help   help for cron
```

deta cron set

Set deta micro to run on a schedule

Command

```
deta cron set [path] <expression> [flags]
```

Flags

```
  -h, --help   help for set
```

Examples

```
Rate:

1. deta cron set 1 minute : run every minute
2. deta cron set 5 hours : run every five hours

Cron expressions:

1. deta cron set 0 10 * * ? * : run at 10:00 am(UTC) every day
2. deta cron set 30 18 ? * MON-FRI * : run at 6:00 pm(UTC) Monday through Friday
3. deta cron set 0/5 8-17 ? * MON-FRI * : run every 5 minutes Monday through
Friday between 8:00 am and 5:55 pm(UTC)
```

deta cron remove

Remove a schedule from a deta micro

Command

```
deta cron remove [path] [flags]
```

Flags

```
  -h, --help   help for remove
```

Issues

If you run into any issues, consider reporting them in our Github Discussions.

Go to TOC

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

Installing & Configuring the Deta CLI

<Tabs defaultValue=mac values={[{ label: 'Mac', value: 'mac', }, { label: 'Linux', value: 'linux', }, { label: 'Windows', value: 'win', },] }>

To install the Deta CLI, open a Terminal and enter:

```
curl -fsSL https://get.deta.dev/cli.sh | sh
```

To install the Deta CLI, open a Terminal and enter:

```
curl -fsSL https://get.deta.dev/cli.sh | sh
```

To install the Deta CLI, open PowerShell and enter:

```
iwr https://get.deta.dev/cli.ps1 -useb | iex
```

This will download the binary which contains the CLI code. It will try to export the `deta` command to your path. If it does not succeed, follow the directions outputted by the install script to export `deta` to your path.

Logging into Deta via the CLI

Once you have successfully installed the Deta CLI, you'll need to log in to Deta.

From a Terminal, type `deta login`.

This command will open your browser and authenticate your CLI through Deta's web application.

Upon a successful log-in, you are ready to start building Micros.

Issues

If you run into any issues, consider reporting them in our Github Discussions.

Go to TOC

v1.3.3-beta

Updates

- The CLI no longer checks the `Hidden File Attribute` in Windows systems

Fixes

- Fix overwriting files on `deta new` in Windows

v1.3.2-beta

Fixes

- Fix bad file path bug for hidden file check in windows

v1.3.1-beta

Updates

- `deta details`: show cron time in output if cron is set for Micro
- `deta clone`: save `state` after cloning

v1.3.0-beta

New

- `deta deploy purge-dependencies`: remove all installed dependencies from your micro

Updates

- `deta new`: support new runtime `python3.8`
- `deta update`: support new runtime `python3.8`
- `deta deploy`: purge all installed dependencies instead of uninstalling individual dependencies, if all of them are removed at the same time; use standard http lib to detect binary files and remove third party lib dependency
- `deta new`, `deta pull`, `deta clone`: use new api to download micro code

Fixes

- Fix installation order bug when upgrading packages
- Fix removal of `$HOME/.deta` folder bug on some cases in `deta new`

v1.2.0-beta

New

- `deta new` : add `--runtime` flag, choose runtime when creating a new micro
- `deta update`: add `--runtime` flag, update micro's runtime
- `deta logs`: add `--follow` flag, follow logs

Updates

- `deta details`: add id, project and region to details output

Fixes

- Fix bad state file path in windows
- Fix `deta version upgrade` in windows
- Fix `.detaignore` issues in windows

v1.1.4-beta

Fixes

- fix file parsing bug with end of line sequence on windows

v1.1.3-beta

Updates

- add support for `.detaignore` file
- always show login url when logging in with `deta login`
- confirm a new version is not a pre-release version on new version check

Fixes

- fix parsing of environment variables with double quotes
- fix issues with intermittent discrepancy of local and server state of a micros' details

v1.1.2-beta

Updates

- add access token authentication
- `deta watch` does an initial deployment
- show login page url if failed to open login page in browser
- suppress usage messages on errors

Fixes

- fix cases of corrupted deployments on `deta watch`
- fix parsing environment key value pairs

v1.1.1-beta

Fixes

- `deta new`: Fix issues with incomplete/failed deployment if application code already exists in the root directory
- recognize `.mo` files as binary files

v1.1.0-beta

New Commands

`deta clone`: Clone an existing micro

`deta cron`: Set a micro to run on a schedule

`deta run`: Run a micro from the CLI

`deta version upgrade`: Upgrade Deta version

`deta visor open`: Open the Visor page for a micro

Updates

`deta pull`: Pull only pulls latest code of an existing micro from the micro's root directory, add `--force` flag

`deta deploy`: Checks for latest available version on successful deployments

Fixes

`deta pull`: Fix deta pull creating an empty directory even if pull fails

`deta deploy`: Fix issues with deploying binary files (edited)

`deta details`: Fix issue with displaying wrong values of visor and http_auth

Go to TOC

Adding/Updating dependencies fails with no output

On windows, if adding or updating dependencies fails with the following or similar error message but there is no further output:

```
Error: failed to update dependencies: error on one or more dependencies,
no dependencies were added, see output for details.
```

Then, please make sure the requirements file encoding is `UTF-8`.

> If you have used the command `pip freeze > requirements.txt` in `powershell` on Windows, the file created is not encoded in `UTF-8`. Please change the encoding or create a new file with the right encoding.

Nodejs Micros cannot serve binary files

Serving some binary files (images, fonts etc) fails on nodejs micros currently because of a bug. We are sorry for this and will push a fix as soon as we can.

As a workaround, please add the environment variable `BINARY_CONTENT_TYPES={file_mime_type}` in the micro's environment to serve the file. The `file_mime_type` can accept wildcards (`*`) and comma separated values.

For instance, if you want to serve images and fonts from your nodejs micro, setting

```
BINARY_CONTENT_TYPES=image/*,font/*
```

will let the micro serve these files.

Please, use the `deta update -e [env_file_name]` command to update the environment variables of the Micro.

Cannot login from the web browser after `deta login`

If logging in with `deta login` from the Brave Browser fails, disable the shields for that page and try again. If it's not the Brave Browser, please use a different browser to login (you can copy the link in a different browser after the login page opens up), or use deta access tokens instead.

Account is unconfirmed but I did not receive a confirmation email

Please check your spam folder as well for the confirmation email. If you have not received one, please send us an email at team@deta.sh for activating your account with the email address you used at sign up.

Go to TOC

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

General

> You can get your **Project Key** and your **Project ID** from your [Deta dashboard](https://we-b.deta.sh). You need these to talk with the Deta Drive API.

Root URL

This URL is the base for all your HTTP requests:

`https://drive.deta.sh/v1/{project_id}/{drive_name}`

> The `drive_name` is the name given to your drive. If you already have a **Drive**, then you can go ahead and provide it's name here. Additionally, you could provide any name here when doing any `PUT` or `POST` request and our backend will automatically create a new drive for you if it does not exist. There is no limit on how many Drives you can create.

Auth

A **Project Key** *must* be provided in the request **headers** as a value for the `X-Api-Key` key for authentication and authorization.

Example `X-Api-Key: a0kjsdfjda_thisIsYourSecretKey`

File Names And Directories

Each file needs a unique `name` which identifies the file. Directorial hierarchies are represented logically by the `name` of the file itself with the use of backslash `/`.

For example, if you want to store a file `world.txt` under the directory `hello` , the `name` of the file should be `hello/world.txt`.

The file name **must not** end with a `/`. This also means that you can not create an empty directory.

A directory ceases to exist if there are no more files in it. ## Endpoints ### Put File `POST /files?name={name}`

Stores a smaller file in a single request. Use this endpoint if the file size is small enough to be sent in a single request. The file is overwritten if the file with given `name` already exists.

> We do not accept payloads larger than 10 MB on this endpoint. For larger uploads, use chunked uploads.

<Tabs defaultValue=request values={[{label: 'Request', value:'request', }, {label: 'Response', value: 'response', },] }>

Headers	Required	Description
`Content-Type`	No	The content type of the file. If the content type is not specified, the file `name` is used to figure out the content type. Defaults to `application/octet-stream` if the content type can not be figured out.

Query Params	Required	Description
`name`	Yes	The `name` of the file. More here.

`201 Created`

```
Content-Type: application/json

{
    name: file name,
    project_id: deta project id,
    drive_name: deta drive_name
}
```

Client Errors

`400 Bad Request`

```
Content-Type: application/json
{
    errors: [
        // error messages
    ]
}
```

`413 Payload Too Large`

```
Content-Type: text/html
<h1> 413 - Request Entity Too Large </h1>
```

Initialize Chunked Upload

Initializes a chunked file upload. If the file is larger than 10 MB, use this endpoint to initialize a chunked file upload.

`POST /uploads?name={name}`

<Tabs defaultValue=request values={[{label: 'Request', value:'request', }, {label: 'Response', value: 'response', },] }>

Params	Required	Description
`name`	Yes	The `name` of the file. More here.

`202 Accepted`

```
Content-Type: application/json

{
    name: file name,
    upload_id: a unique upload id
    project_id: deta project id,
    drive_name: deta drive name
}
```

Client Errors

`400 Bad Request`

```
Content-Type: application/json
{
    errors: [
        // error messages
    ]
}
```

Upload Chunked Part

Uploads a chunked part.

`POST /uploads/{upload_id}/parts?name={name}&part={part}`

> Each chunk must be at least 5 MB and at most 10 MB. The final chunk can be less than 5 MB.

<Tabs defaultValue=request values={[{label: 'Request', value:'request', }, {label: 'Response', value: 'response', },] }>

Params	Required	Description
`upload_id`	Yes	The `upload_id` received after initiating a chunked upload
`name`	Yes	The `name` of the file. More here.
`part`	Yes	The chunk part number, start with `1`

`200 Ok`

```
Content-Type: application/json

{
    name: file name,
    upload_id: a unique upload id
    part: 1, // upload part number
    project_id: deta project id,
    drive_name: deta drive name
}
```

Client Errors

`400 Bad Request`

```
Content-Type: application/json
{
    errors: [
        // error messages
    ]
}
```

`404 Not Found`

```
Content-Type: application/json
{
    errors: [
        // error messages
    ]
}
```

`413 Payload Too Large`

```
Content-Type: text/html
<h1> 413 Request Entity Too Large </h1>
```

End Chunked Upload

End a chunked upload.

`PATCH /uploads/{upload_id}?name={name}`

<Tabs defaultValue=request values={[{label: 'Request', value:'request', }, {label: 'Response', value: 'response', },] }>

Params	Required	Description
`upload_id`	Yes	The `upload_id` received after initiating a chunked upload
`name`	Yes	The `name` of the file. More here.

`200 Ok`

```
Content-Type: application/json

{
    name: file name,
    upload_id: a unique upload id
    project_id: deta project id,
    drive_name: deta drive name
}
```

Client Errors

`400 Bad Request`

```
Content-Type: application/json
{
    errors: [
        // error messages
    ]
}
```

`404 Not Found`

```
Content-Type: application/json
{
    errors: [
        // error messages
    ]
}
```

Abort Chunked Upload

Aboart a chunked upload.

`DELETE /uploads/{upload_id}?name={name}`

<Tabs defaultValue=request values={[{label: 'Request', value:'request', }, {label: 'Response', value: 'response', },] }>

Params	Required	Description
`upload_id`	Yes	The `upload_id` received after initiating a chunked upload
`name`	Yes	The `name` of the file. More here.

`200 Ok`

```
Content-Type: application/json

{
    name: file name,
    upload_id: a unique upload id
    project_id: deta project id,
    drive_name: deta drive name
}
```

Client Errors

`400 Bad Request`

```
Content-Type: application/json
{
    errors: [
        // error messages
    ]
}
```

`404 Not Found`

```
Content-Type: application/json
{
    errors: [
        // error messages
    ]
}
```

Download File

Download a file from drive.

`GET /files/download?name={name}`

<Tabs defaultValue=request values={[{label: 'Request', value:'request', }, {label: 'Response', value: 'response', },] }>

Params	Required	Description
`name`	Yes	The `name` of the file. More here.

`200 Ok`

```
Accept-Ranges: bytes
Content-Type: {content_type}
Content-Length: {content_length}
{Body}
```

Client Errors

`400 Bad Request`

```
Content-Type: application/json
{
    errors: [
        // error messages
    ]
}
```

`404 Not Found`

```
Content-Type: application/json
{
    errors: [
        // error messages
    ]
}
```

List Files

List file names from drive.

`GET /files?limit={limit}&prefix={prefix}&last={last}`

<Tabs defaultValue=request values={[{label: 'Request', value:'request', }, {label: 'Response', value: 'response', },] }>

Params	Required	Description
`limit`	No	The `limit` of number of file names to get, defaults to `1000`
`prefix`	No	The `prefix` that each file name must have.
`last`	No	The `last` file name seen in a paginated response.

`200 Ok`

The response is paginated based on the `limit` provided in the request. By default, maximum `1000` file names are sent.

If the response is paginated, the response contains a `paging` object with `size` and `last` keys; `size` is the number of file names in the response, and `last` is the last file name seen in the response. The value of `last` should be used in subsequent requests to continue recieving further pages.

```
Content-Type: application/json

{
    paging: {
        size: 1000, // the number of file names in the response
        last: last file name in response
    },
    names: [file1, file2, ...]
}
```

Client Errors

`400 Bad Request`

```
Content-Type: application/json
{
    errors: [
        // error messages
    ]
}
```

`404 Not Found`

```
Content-Type: application/json
{
    errors: [
        // error messages
    ]
}
```

Delete Files

Delete files from drive.

`DELETE /files`

<Tabs defaultValue=request values={[{label: 'Request', value:'request', }, {label: 'Response', value: 're-sponse', },] }>

```
Content-Type: application/json
{
    names: [file_1, file_2]
}
```

Params	Required	Description
`names`	Yes	The `names` of the files to delete, maximum `1000` file names

`200 Ok`

```
Content-Type: application/json

{
    deleted: [file_1, file_2, ...] // deleted file names
    failed: {
        file_3: reason why file could not be deleted,
        file_4: reason why file could not be deleted,
        //...
    }
}
```

File names that did not exist will also be under `deleted`, `failed` will only contain names of files that existed but were not deleted for some reason

Issues

If you run into any issues, consider reporting them in our Github Discussions.

Go to TOC

Deta Drive is a managed, secure and scalable file storage service with a focus on end-user simplicity.

Is my data secure?

Your data is encrypted and stored safely at rest. Encryption keys are managed by us.

What's the storage limit?

10GB per Deta account.

How do I start?

1. Log in to Deta.
2. Grab your **Project Key** and start writing code in Python, Node.js, or use the HTTP API where you need file storage.

Go to TOC

Drive UI lets you inspect, add, delete files from a Drive via a GUI.

Opening Drive UI

You can open an individual Drive's UI within any project by clicking on the Drive name in the project sidebar.

drive_ui_1

All the files and folders from within the path `/` should load into the table view when clicked.

Preview and Navigation

Files that are previewable (and folders) are marked in blue.

drive_ui_2

Clicking a folder will navigate to that folder, while clicking a file will open up a preview.

drive_ui_3

Your current location is highlighted in black in the navigation bar at the top of Drive UI.

drive_ui_4

You can click a parent folder, or the Drive name, to jump to that location.

Uploading & Downloading Files

You can upload a file by dragging it into the list of files & folders. This will upload the file in the current directory.

drive_ui_5

To download a file, click the download icon, which is on the right side of the the table.

Deleting Files

To delete files, click on the checkbox(es) for any files(s) and then click the trash icon in the top right corner of the Drive UI panel.

drive_ui_6

You will be asked to confirm you want to delete the file(s).

Final Notes

We hope you enjoy Drive UI!

Drive UI is still in Beta; it has been internally tested but may have some uncaught bugs or issues.

Issues

If you run into any issues, consider reporting them in our Github Discussions.

Go to TOC

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

Building a Simple Image Server with Deta Drive

Setup

To get started, create a directory `image-server` and change the current directory into it.

```
$ mkdir image-server && cd image-server
```

Before we begin, let's install all the necessary dependencies for this project.

```
$ npm install deta express express-fileupload
```

In this tutorial, we are using `express` to build our server, and `express-fileupload` allows us to access the uploaded file data.

To configure the app, import the dependencies and instantiate drive in `index.js`

```
const { Deta } = require(deta);
const express = require(express);
const upload = require(express-fileupload);

const app = express();

app.use(upload());

const deta = Deta(Project_Key);
const drive = deta.Drive(images);
```

We have everything we need to 🚀

Uploading Images

First, we need to render a HTML snippet to display the file upload interface.

We'll expose a function that renders the HTML snippet on the base route `/`

```
app.get('/', (req, res) => {
    res.send(`
    <form action=/upload enctype=multipart/form-data method=post>
      <input type=file name=file>
      <input type=submit value=Upload>
    </form>`);
});
```

We are simply rendering a HTML form that sends a HTTP `POST` request to the route `/upload` with file data.

Let's complete file upload by creating a function to handle `/upload`

```
app.post(/upload, async (req, res) => {
    const name = req.files.file.name;
    const contents = req.files.file.data;
    const img = await drive.put(name, {data: contents});
    res.send(img);
});
```

We can access the image details from `req` and store it in Drive.

Downloading Images

To download images, we can simply use `drive.get(name)`

If we tie a `GET` request to the `/download` path with a param giving a name (i.e `/download/space.png`), we can return the image over HTTP.

```
app.get(/download/:name, async (req, res) => {
    const name = req.params.name;
    const img = await drive.get(name);
    const buffer = await img.arrayBuffer();
    res.send(Buffer.from(buffer));
});

app.listen(3000);
```

Running the server

To run the server locally, navigate to the terminal in the project directory (`image-server`) and run the following command:

```
$ node index.js
```

// /download/tut.jpg/

```
curl -X 'POST' \
  'http://127.0.0.1:3000/upload' \
  -H 'accept: application/json' \
  -H 'Content-Type: multipart/form-data' \
  -F 'file=@space.png;type=image/png'

Response
space.png

curl -X 'GET' \
  'http://127.0.0.1:3000/download/space.png' \
  -H 'accept: application/json'

Response
The server should respond with the image.
```

Issues

If you run into any issues, consider reporting them in our Github Discussions.

Go to TOC

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

Building a Simple Image Server with Deta Drive

Setup

To get started, create a directory `image-server` and change the current directory into it.

```
$ mkdir image-server && cd image-server
```

Before we begin, let's install all the necessary dependencies for this project. Create a `requirements.txt` with the following lines:

```
fastapi
uvicorn
deta
python-multipart
```

> If you are using Deta Drive within a Deta Micro, you should ignore `uvicorn`, but you must include `deta` in your `requirements.txt` file to install the lastest sdk version, other than that it won't work.work.

We are using `FastAPI` to build our simple image server, and `python-multipart` allows us access the uploaded files.

Run the following command to install the dependencies.

```
$ pip install -r requirements.txt
```

To configure the app, import the dependencies and instantiate drive in `main.py`

```
from fastapi import FastAPI, File, UploadFile
from fastapi.responses import HTMLResponse, StreamingResponse
from deta import Deta

app = FastAPI()
deta = Deta(Project_Key)  # configure your Deta project
drive = deta.Drive(images) # access to your drive
```

We have everything we need to 🚀

Uploading Images

First, we need to render a HTML snippet to display the file upload interface.

We'll expose a function that renders the HTML snippet on the base route `/`

```
@app.get(/, response_class=HTMLResponse)
def render():
    return
    <form action=/upload enctype=multipart/form-data method=post>
```

```
        <input name=file type=file>
        <input type=submit>
    </form>
```

We are simply rendering a form that sends a HTTP `POST` request to the route `/upload` with file data.

Let's complete file upload by creating a function to handle `/upload`

```
@app.post(/upload)
def upload_img(file: UploadFile = File(...)):
    name = file.filename
    f = file.file
    res = drive.put(name, f)
    return res
```

Thanks to the amazing tools from FastAPI, we can simply wrap the input around `UploadFile` and `File` to access the image data. We can retrieve the name as well as bytes from `file` and store it in Drive.

Downloading images

To download images, we can simply use `drive.get(name)`

If we tie a `GET` request to the `/download` path with a param giving a name (i.e `/download/space.png`), we can return the image over HTTP.

```
@app.get(/download/{name})
def download_img(name: str):
    res = drive.get(name)
    return StreamingResponse(res.iter_chunks(1024), media_type=image/png)
```

You can learn more about `StreamingResponse` here.

Running the server

To run the server locally, navigate to the terminal in the project directory (`image-server`) and run the following command:

```
$ uvicorn main:app
```

Your image server is now ready! You can interact with it at `/` and check it out!

// /download/tut.jpg/

```

curl -X 'POST' \
  'http://127.0.0.1:8000/upload' \
  -H 'accept: application/json' \
  -H 'Content-Type: multipart/form-data' \
  -F 'file=@space.png;type=image/png'

Response
space.png
```

```
curl -X 'GET' \
  'http://127.0.0.1:8000/download/space.png' \
  -H 'accept: application/json'

Response
The server should respond with the image.
```

Issues

If you run into any issues, consider reporting them in our Github Discussions.

Go to TOC

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

The Deta library is the easiest way to store and retrieve files from your Deta Drive. Currently, we support JavaScript, Typescript and Python 3. Drop us a line if you want us to support your favorite language.

Installing

Using NPM:

```
npm install deta
```

Using Yarn:

```
yarn add deta
```

```
pip install deta
```

```
go get github.com/deta/deta-go
```

Instantiating

To start working with your Drive, you need to import the `Deta` class and initialize it with a **Project Key**. Then instantiate a subclass called `Drive` with a database name of your choosing.

```
const { Deta } = require('deta'); // import Deta

// this also works
import { Deta } from 'deta';

// Initialize with a Project Key
const deta = Deta('project key');

// You can create as many as you want
const photos = deta.Drive('photos');
const docs = deta.Drive('docs');
```

If you are using Deta Drive within a [Deta Micro](/docs/micros/about), you must include `deta` in your `package.json` file to install the latest sdk version. A valid project key is pre-set in the Micro's environment. There is no need to pass a key in the initialization step.

```
const { Drive } = require('deta');
const drive = Drive('simple_drive');
```

If you are using the `deta` npm package of `0.0.6` or below, `Deta` is the single default export and should be imported as such.

```
const Deta = require('deta');
```

```
from deta import Deta  # Import Deta

# Initialize with a Project Key
deta = Deta(project key)

# This how to connect to or create a database.
drive = deta.Drive(simple_drive)

# You can create as many as you want
photos = deta.Drive(photos)
docs = deta.Drive(docs)
```

If you are using Deta Drive within a [Deta Micro](/docs/micros/about), you must include `deta` in your `requirements.txt` file to install the latest sdk version. A valid project key is pre-set in the Micro's environment. There is no need to pass a key in the the initialization step.

```
from deta import Drive

drive = Drive(simple_drive)
```

```
import (
    fmt

    github.com/deta/deta-go/deta
    github.com/deta/deta-go/service/drive
)

func main() {

    // initialize with project key
    // returns ErrBadProjectKey if project key is invalid
    d, err := deta.New(deta.WithProjectKey(project_key))
    if err != nil {
        fmt.Println(failed to init new Deta instance:, err)
        return
    }

    // initialize with drive name
    // returns ErrBadDriveName if base name is invalid
    drive, err := drive.New(d, drive_name)
    if err != nil {
        fmt.Println(failed to init new Drive instance:, err)
        return
    }
}
```

Your project key is confidential and meant to be used by you. Anyone who has your project key can access your database. Please, do not share it or commit it in your code.

Using

Deta's **`Drive`** offers the following methods to interact with your Deta Drive:

`put` - Stores a file to drive. It will overwrite the file if the file already exists.

`get` - Retrieves a file from drive by the file name.

`delete` - Deletes a file from drive.

`list` - Lists the file names in a drive.

Put

`Put` uploads and stores a file in a drive with a given `name`. It will overwrite the file if the file name already exists.

<Tabs groupId=preferred-language defaultValue=js values={[{label:'JavaScript', value: 'js', }, {label:'Python', value: 'py', }, {label:'Go', value: 'go', }]}

`async put(name, options)`

Parameters

- **name** (required) - `string`
  - Description: The name of the file.
- **options** (required) - `{data : string | Uint8Array | Buffer, path: string, contentType: string}`
  - Description: An object with three optional parameters.
    - **data** - `string` or `Buffer`
      - Description: Either the data string or a buffer.
    - **path** - `string`
      - Description: The path of the file to be uploaded to drive.
    - **contentType** - `string`
      - Description: The content type of the file to be uploaded to drive. If the content type is not provided, `drive` tries to figure out the content type from the `name` provided. It defaults to `application/octet-stream` if the content type can not be figured out from the file name.

  `options` must have at least and at most one of two properties `data` or `path` defined.

Returns

Returns a promise which resolves to the name of the item on a successful `put`, otherwise, it throws an `Error` on error.

Example

```
drive.put('hello.txt', {data: Hello world});
drive.put('hello.txt', {data: Hello world, contentType: 'text/plain'});

drive.put('hello.txt', {data: Buffer.from('Hello World'), contentType:
'text/plain'});
drive.put('hello.txt', {path: './my/file/path/file.txt'});
drive.put('hello.txt', {path: './my/file/path/file.txt', contentType:
'text/plain'}});
```

`put(name, data, *, path, content_type)`

Parameters

- **name** (required) - `string`
  - Description: The name of the file.
- **data** - `string | bytes | io.TextIOBase | io.BufferedIOBase | io.RawIOBase`
  - Description: The data content of the file.
- **path** - `string`
  - Description: The local path of the file to be uploaded to drive.
- **content_type** - `string`
  - Description: The content type of the file to be uploaded to drive. If the content type is not provided, `drive` tries to figure out the content type from the `name` provided. It defaults to `application/octet-stream` if the content type can not be figured out from the file name.

  At least and at most one of two args `data` or `path` must be provided. `path` and `content_type` must be provided with the key words.

Returns

Returns the name of the file on a successful `put`, otherwise, raises an `Exception` on error.

Example

```
drive.put('hello.txt', 'Hello world')
drive.put(b'hello.txt', 'Hello world')
drive.put('hello.txt', content_type='text/plain')

import io
drive.put('hello.txt', io.StringIO('hello world'))
drive.put('hello.txt', io.BytesIO(b'hello world'))

f = open('./hello.txt', 'r')
drive.put('hello.txt', f)
f.close()

drive.put('hello.txt', path='./hello.txt')
```

`Put(i *PutInput) (string, error)`

Parameters

- **i** (required) - pointer to a `PutInput`

    -
      ```
      // PutInput input to Put operation
      type PutInput struct {
        // Name of the file
        Name string
        // io.Reader with contents of the file
        Body io.Reader
        // ContentType of the file to be uploaded to drive.
        ContentType string
      }
      ```
    - `Name` (required) - `string`
        - Description: Name of the file to be uploaded.
    - `Body` (required) - `io.Reader`
        - Description: File content to be uploaded.
    - `ContentType` - `string`
        - Description: If the content type is not provided, drive tries to figure out the content type from Name provided. It defaults to application/octet-stream if the content type can not be figured out from the file name.

Returns

Returns the `name` of the file on a successful put (otherwise empty name), and an `error`.

Example

```
import (
    bufio
    fmt
    os

    github.com/deta/deta-go/deta
    github.com/deta/deta-go/service/drive
)

func main() {

    // initialize with project key
    // returns ErrBadProjectKey if project key is invalid
    d, err := deta.New(deta.WithProjectKey(project_key))
    if err != nil {
        fmt.Println(failed to init new Deta instance:, err)
        return
    }

    // initialize with drive name
    // returns ErrBadDriveName if drive name is invalid
    drawings, err := drive.New(d, drawings)
    if err != nil {
        fmt.Println(failed to init new Drive instance:, err)
        return
```

```
    }
    // PUT
    // reading from a local file
    file, err := os.Open(./art.svg)
    defer file.Close()

    name, err := drawings.Put(&drive.PutInput{
        Name:        art.svg,
        Body:        bufio.NewReader(file),
        ContentType: image/svg+xml,
    })
    if err != nil {
        fmt.Println(failed to put file:, err)
        return
    }
    fmt.Println(Successfully put file with name:, name)
}
```

Get

`Get` retrieves a file from a drive by its name.

<Tabs groupId=preferred-language defaultValue=js values={[{label:'JavaScript', value: 'js', }, {label:'Python', value: 'py', }, {label:'Go', value: 'go', }]}>

`async get(name)`

Parameters

- **name** (required) - `string`
  - Description: The `name` of the file to get.

Returns

Returns a promise that resolves to a `blob` of data if found, else `null`. Throws an `Error` on errors.

Example

```
const buf = await drive.get('hello.txt');
```

`get(name)`

Parameters

- **name** (required) - `string`
  - Description: The `name` of the file to get.

Returns

Returns a instance of a `DriveStreamingBody` class which has the following methods/properties.

- `read(size=None)` : Reads all or up to the next `size` bytes. Calling `read` after all the content has been read will return empty bytes.

- `iter_chunks(chunk_size:int=1024)` : Returns an iterator that yields chunks of bytes of `chunk_size` at a time.
- `iter_lines(chunk_size:int=1024)` : Returns an iterator that yields lines from the stream. Bytes of `chunk_size` at a time is read from the raw stream and lines are yielded from there. The line delimiter is always `b'\n'`.
- `close()` : Closes the stream.
- `closed` : Returns `True` if the stream has been closed.

Example

```
hello = drive.get('hello.txt')
content = hello.read()
hello.close()

# larger files
# iterate chunks of size 4096 and save to disk
large_file = drive.get('large_file.txt')
with open(large_file.txt, wb+) as f:
  for chunk in large_file.iter_chunks(4096):
      f.write(chunk)
  large_file.close()
```

`Get(name string) (io.ReadCloser, error)`

Parameters

- **name** (required) - `string`
  - Description: The `name` of the file to get.

Returns

Returns a `io.ReadCloser` for the file.

Example

```
import (
    fmt
    io/ioutil

    github.com/deta/deta-go/deta
    github.com/deta/deta-go/service/drive
)

func main() {

    // initialize with project key
    // returns ErrBadProjectKey if project key is invalid
    d, err := deta.New(deta.WithProjectKey(project_key))
    if err != nil {
        fmt.Println(failed to init new Deta instance:, err)
        return
    }
    // initialize with drive name
    // returns ErrBadDriveName if drive name is invalid
    drawings, err := drive.New(d, drawings)
    if err != nil {
        fmt.Println(failed to init new Drive instance:, err)
        return
    }

    // GET
    name := art.svg
    f, err := drawings.Get(name)
    if err != nil {
        fmt.Println(failed to get file with name:, name)
        return
    }
    defer f.Close()

    c, err := ioutil.ReadAll(f)
    if err != nil {
        fmt.Println(failed read file content with err:, err)
        return
    }
    fmt.Println(file content:, string(c))
}
```

Delete

`Delete` deletes a file from drive.

<Tabs groupId=preferred-language defaultValue=js values={[{label:'JavaScript', value: 'js', }, {label:'Python', value: 'py', }, {label:'Go', value: 'go', }]}

`async delete(name)`

Parameters

- **name** (required) - `string`
  - Description: The name of the file to delete

Returns

Returns a promise that resolves to the `name` of the deleted file on successful deletions, otherwise raises an `Error`

> If the file did not exist, the file is still returned as deleted.

Example

```
const deletedFile = await drive.delete(hello.txt);
```

`delete(name)`

Parameters

- **name** (required) - `string`
  - Description: The name of the file to delete

Returns

Returns the `name` of the deleted file on successful deletions, otherwise raises an `Exception`

> If the file did not exist, the file is still returned as deleted.

Example

```
deleted_file = drive.delete(hello.txt)
```

`Delete(name string) (string, error)`

Parameters

- **name** (required) - `string`
  - Description: The name of the file to delete

Returns

Returns the `name` of the deleted file on successful deletions, and an `error`.

> If the file did not exist, the name is still returned.

Example

```
import (
    fmt

    github.com/deta/deta-go/deta
    github.com/deta/deta-go/service/drive
)

func main() {

    // initialize with project key
    // returns ErrBadProjectKey if project key is invalid
    d, err := deta.New(deta.WithProjectKey(project_key))
    if err != nil {
        fmt.Println(failed to init new Deta instance:, err)
        return
    }

    // initialize with drive name
    // returns ErrBadDriveName if drive name is invalid
    drawings, err := drive.New(d, drawings)
    if err != nil {
        fmt.Println(failed to init new Drive instance:, err)
        return
    }
    // DELETE
    name, err := drawings.Delete(art.svg)
    if err != nil {
        fmt.Println(failed to delete file with name:, name)
        return
    }
    fmt.Println(Successfully deleted file with name:, name)
}
```

Delete Many

Deletes multiple files (up to 1000) from a drive.

<Tabs groupId=preferred-language defaultValue=js values={[{label:'JavaScript', value: 'js', }, {label:'Python', value: 'py', }, {label:'Go', value: 'go', }]}>

`async deleteMany(names)`

Parameters

- **names** (required) - `Array of string`
    - description: The names of the files to be deleted.

Returns

Returns a promise which resolves to an object with `deleted` and `failed` keys indicating deleted and failed file names.

```
{
  deleted : [file1.txt, file2.txt, ...],
  failed: {
    file_3.txt: reason for failure,
  }
}
```

If a file did not exist, the file is still returned as deleted.

Example

```
const result = await drive.DeleteMany([file1.txt, file2.txt]);
console.log(deleted:, result.deleted)
console.log(failed:, result.failed)
```

`delete_many(names)`

Parameters

- **names** (required): `string`
    - Description: The names of the files to be deleted.

Returns

Returns a `dict` with `deleted` and `failed` keys indicating deleted and failed file names.

```
{
  deleted : [file1.txt, file2.txt, ...],
  failed: {
    file_3.txt: reason for failure
  }
}
```

> If a file did not exist, the file is still returned as deleted.

Example

```
result = drive.delete_many([file1.txt, file2.txt]);
print(deleted:, result.get(deleted))
print(failed:, result.get(failed))
```

`DeleteMany(names []string) (*DeleteManyOutput, error)`

Parameters

- **names** (required): `[]string`
  - Description: The names of the files to be deleted.

Returns

Returns a pointer to a `DeleteManyOutput` and an `error`.

```
// DeleteManyOutput output to DeleteMany operation
type DeleteManyOutput struct {
    Deleted []string          `json:deleted`
    Failed  map[string]string `json:failed`
}
```

- `Deleted` - string slice indicating deleted file names.
- `Failled` - map indicating the names of failed file names along with an error message.

Example

```
import (
    fmt

    github.com/deta/deta-go/deta
    github.com/deta/deta-go/service/drive
)

func main() {
    // initialize with project key
    // returns ErrBadProjectKey if project key is invalid
    d, err := deta.New(deta.WithProjectKey(project_key))
    if err != nil {
        fmt.Println(failed to init new Deta instance:, err)
        return
    }

    // initialize with drive name
    // returns ErrBadDriveName if drive name is invalid
    drawings, err := drive.New(d, drawings)
    if err != nil {
        fmt.Println(failed to init new Drive instance:, err)
        return
    }

    names := []string{a, b, c}
    dr, err := drawings.DeleteMany(names)

    if err != nil {
        fmt.Println(failed to delete files)
        return
    }
    fmt.Println(deleted:, dr.Deleted)
    fmt.Println(failed:, dr.Failed)
}
```

List

`List` files in your drive.

<Tabs groupId=preferred-language defaultValue=js values={[{label:'JavaScript', value: 'js', }, {label:'Python', value: 'py', }, {label:'Go', value: 'go', }]}

`async list(options)`

Parameters

- **options** (required) : `{prefix: string, limit: number, last: string}`
  - Description: An object with three optional parameters.
    - **prefix**: `string`
      - Description: The prefix that file names must have.
    - **limit**: `number`
      - Description: The maximum number of files names to be returned, defaults to `1000`
    - **last**: `string`
      - Description: The `last` name seen in a previous paginated result. Provide `last` to fetch further pages.

Returns

Returns a promise which resolves to an `object` with `paging` and `names` keys.

```
{
  names: [file1.txt, file2.txt, ...],
  paging: {
    size: 2,
    last: file_2.txt
  }
}
```

- `names` : The names of the files
- `paging` : Contains paging information.
  - `size` : The number of files returned.
  - `last` : The last name seen in the paginated response. Provide this value in subsequent api calls to fetch further pages. For the last page, `last` is not present in the response.

Example

```
// get all files
let result = await drive.list();
let allFiles = result.names;
let last = result.paging.last;

while (last){
  // provide last from previous call
  result = await drive.list({last:result.paging.last});

  allFiles = allFiles.concat(result.names)

  // update last
  last = result.paging.last
}
console.log(all files:, allFiles)

const resultWithPrefix = await drive.list({prefix: blog/})
const resultWithLimit = await drive.list({limit: 100})
const resultWIthLimitAndPrefix = await drive.list({limit: 100, prefix:blog/})
```

`list(limit, prefix, last)`

Parameters

- **limit**: `int`
  - Description: The maximum number of files names to be returned, defaults to `1000`
- **prefix**: `string`
  - Description: The prefix that file names must have.
- **last**: `string`
  - Description: The `last` name seen in a previous paginated result. Provide `last` from previous response to fetch further pages.

Returns

Returns a `dict` with `paging` and `names` keys.

```
{
  names: [file1.txt, file2.txt, ...],
  paging: {
    size: 2,
    last: file_2.txt
  }
}
```

- `names` : The names of the files
- `paging` : Contains paging information.
  - `size` : The number of files returned.
  - `last` : The last name seen in the paginated response. Provide this value in subsequent api calls to fetch further pages. For the last page, `last` is not present in the response.

Example

```
# get all files
result = drive.list()

all_files = result.get(names)
paging = result.get(paging)
last = paging.get(last) if paging else None

while (last):
  # provide last from previous call
  result = drive.list(last=last)

  all_files += result.get(names)
  # update last
  paging = result.get(paging)
  last = paging.get(last) if paging else None

print(all files:, all_files)

res_with_prefix = drive.list(prefix=/blog)
res_with_limit = drive.list(limit=100)
res_with_prefix_limit = drive.list(prefix=/blog, limit=100)
```

`List(limit int, prefix, last string) (*ListOutput, error)`

Parameters

- **limit** (required) - `int`
  - Description: Maximum number of file names to be returned.
- **prefix** (required) - `string`
  - Description: The prefix that file names must have.
- **last** (required) - `string`
  - Description: The `last` name seen in a previous paginated result. Provide `last` from previous response to fetch further pages.

Returns

Returns a pointer to a `ListOutput`

```
type ListOutput struct {
    Paging *paging  `json:paging`
    Names  []string `json:names`
}

type paging struct {
    Size int     `json:size`
    Last *string `json:last`
}
```

- `Paging` - indicates the size and last name of the current page. `nil` if there are no further pages.
- `Names` - names of the files.

Example

```
import (
    fmt

    github.com/deta/deta-go/deta
    github.com/deta/deta-go/service/drive
)

func main() {

    // initialize with project key
    // returns ErrBadProjectKey if project key is invalid
    d, err := deta.New(deta.WithProjectKey(project_key))
    if err != nil {
        fmt.Println(failed to init new Deta instance:, err)
        return
    }

    // initialize with drive name
    // returns ErrBadDriveName if drive name is invalid
    drawings, err := drive.New(d, drawings)
    if err != nil {
        fmt.Println(failed to init new Drive instance:, err)
        return
    }

    lr, err := drawings.List(1000, , )
    if err != nil {
        fmt.Println(failed to list names from drive with err:, err)
    }
    // [a, b, c/d]
    fmt.Println(names:, lr.Names)

    lr, err = drawings.List(1, , )
    if err != nil {
        fmt.Println(failed to list names from drive with err:, err)
    }
    // [a]
    fmt.Println(names:, lr.Names)


    lr, err = drawings.List(2, , )
    if err != nil {
        fmt.Println(failed to list names from drive with err:, err)
    }
    // b
    fmt.Println(last:, *lr.Paging.Last)
}
```

Issues

If you run into any issues, consider reporting them in our Github Discussions.

Go to TOC

What are all the different 'keys' for?

Deta offers 3 types of secrets / keys / tokens:

1. **Project Keys**: Project keys interact with both your data and your resources inside a project (currently allowing reads and writes for the project's Bases and Drives). A project key will be created for you when you create a project, and you can manage all of a project's keys from the *'Project Keys'* option in any project's sidebar. If this key is leaked, the data stored in any of the connected project's Bases and Drives can be compromised.

2. **API Keys**: API Keys are optional keys to protect your Micro HTTP endpoint or to implement client-based rules. Creating API Keys from the Deta CLI is described here. Read more about API Keys here. If this key is leaked, your Micro's http endpoint can be used by unwanted clients.

3. **Access Tokens**: Access tokens are used for authenticating the Deta CLI. Use cases: CI/CD, Gitpod, GitHub Actions. Information on Deta Access Tokens can be found here. If this key is leaked, your whole Deta account is compromised.

These types of keys are not unique to Deta; all other cloud providers use some variant of them.

Can I delete a project?

No, sorry but deleting projects is not available yet.

Go to TOC

Intro

Deta is a free cloud crafted with the developer and user experience at heart.

It is our mission to dramatically reduce the gap between ideas and working cloud applications.

Why is Deta Cloud free for ever?

We want anyone, at any age from anywhere in the world to experiment and build their ideas without worrying about limits of credit cards.

How are you going to make money?

We're working on a big, parallel product that we will be announcing soon. This product will be responsible for generating revenue. Stay tuned!

Resources

To accomplish our mission, we offer a set of minimal-config cloud primitives:

1. **Deta Base** (beta): Instantly usable database with a feature-rich API.
2. **Deta Micros** (beta): Deploy scalable Node & Python apps in seconds.
3. **Deta Drive** (beta): Upload, host and serve images and files.

Local Development and the Deta CLI

Most developers like their local development flow. We provide a CLI that ties your local development workflow to your micro cloud, see:

- Deta CLI

Support

If you have any questions, we provide a few ways to get help ASAP:

- Getting Help

Deta Base

Deta Base is our instantly usable NoSQL database.

- About Deta Base: General Information about Bases.
- Base SDK: SDK for using a Base in Node.js and Python.
- HTTP API: API for interacting with a Base over HTTP.
- Node.js Tutorial: Build a micro-crud in Node.js using Base and Express.js.
- Python Tutorial: Build a micro-crud in Python using Base and Flask.

Deta Micros

Deta Micros(servers) are a lightweight but scalable cloud runtime tied to an HTTP endpoint. Currently Node.js and Python Micros are supported.

- About Deta Micros: General Information about Micros.
- Getting Started with Micros: Get your first Micro live.

Deta Drive

Deta Drive is our easy-to-use file storage service.

- About Deta Drive: General Information about Drive.
- Drive SDK: SDK for using Drive in Node.js and Python.
- HTTP API: API for interacting with Drive over HTTP.
- Node.js Tutorial: Build a simple image server in Node.js using Drive.
- Python Tutorial: Build a simple image server in Python using Drive.

Deta CLI

The Deta CLI allows you to tie your local machine and dev setup to your own personal micro cloud.

You create and update micros using the CLI.

- Install the Deta CLI
- Complete CLI Reference

Getting Help

We have answered some General FAQs and FAQs on Micros in these docs, but we are here to help and would love to hear what you think!

If you have any questions or feedback, you can reach us:

- Github Discussions
- Discord

Reporting Abuse

If you detect any kind of fraudulent, abusive and/or malicious content hosted on Deta, please report the site(s) by sending an email to us at abuse@deta.sh.

We review each report manually.

Go to TOC

Summary

Deta Micros (micro servers) are a lightweight but scalable cloud runtime tied to an HTTP endpoint. They are meant to get your apps up and running *blazingly fast*. Focus on writing your code and Deta will take care of everything else.

Technical Specifications

1. Micros support the following runtimes:
   a. Python: **3.7**, **3.8**, **3.9**
   b. Nodejs: **12.x**, **14.x**

   New micros will use `Python 3.9` or `Nodejs 14.x` by default. If you want to select a different runtime or update the runtime of your micro, please refer to deta new and deta update.

2. Every Micro you use gets its own sandboxed Linux VM.

3. Each Micro has a key and secret keys set in the environment, these are specific to your Micro and not the Deta system. Make sure to not share them to keep your own data safe.

4. An execution times out after 10s. Request an increase.

5. 512 MB of RAM for *each* execution. Request an increase.

6. Read-only file system. **Only `/tmp` can be written to**. It has a 512 MB storage limit.

7. Invocations have an execution processes/threads limit of 1024.

8. HTTP Payload size limit is 5.5 MB.

Important Notes

1. Micros automatically respond to your incoming requests and go to sleep when there's nothing to do. You do not have to manage their lifecycle manually.
2. You (and only you) have access to the VM, which means there's no SSH access.
3. You are supposed to see the VM filesystem, it's not a security vulnerability.
4. Deta Micros do not support connecting to MongoDB at the moment. We recommend using Deta Base instead.
5. We don't believe that Micros will work well with RDMBS like PostgreSQL and MySQL unless you use a pool manager.
6. Micros only support read-only SQLite, which you could deploy with your code.
7. The total upload size of your source code and assets is limited to 250 MB.
8. Dependencies (pip, npm, etc) also can't exceed a combined size of 250mb.
9. For unknown reasons, Google and Firebase packages for Python do not install successfully on Micros.
10. Currently, all requests received by Micros do not contain the client IP addresses. This makes most rate-limiting logic and other IP-dependant logic not work on Micros.
11. Micros support most micro web frameworks (Node.js: express, fastify, koa, etc && Python: FastAPI, Flask, etc.). Other frameworks are not likely to work.

12. Websockets and long-running processes do not work on Micros. (examples: socket.io or Discord bots won't work).
13. Features like Background Tasks and Startup/Shutdown Events will currently not work as expected on Micros.

Go to TOC

API keys are a great way to protect your APIs, which is why Deta has them built into Micros. API keys will work with any framework that runs on the Micros – no coding required from your end!

1. Enable Deta Access

To use API keys with your Micro, first you need to enable **Deta Access** by running the following command from the root directory of your Micro's source code.

```
deta auth enable
```

Now your API is protected by **Deta Access** and only *accessible* by you. Next, you need to create an API key to authenticate your requests.

2. Create an API key

To create a new API key, run the following command, make sure to provide at least the `name` argument.

> For more info, refer to the `deta auth create-key` docs.

```
deta auth create-api-key --name first_key --desc api key for agent 1
```

This will send a response similar to this:

```
{
    name: first_key,
    description: api key for agent 1,
    prefix: randomprefix,
    api_key: randomprefix.supersecretrandomstring,
    created: 2021-04-22T11:50:08Z
}
```

In this example, the API key would be `randomprefix.supersecretrandomstring`.

The key will only be shown to you once, make sure to save it in a secure place.

> API keys are passwords Anyone with this API key can access your Micro, make sure to treat it as a password and save it in a secure place like a password manager.

3. Using API keys

Simply add this key to the header of every request you make to your Micro.

For example, suppose your Micro has an endpoint `/tut` that responds with a simple string `Hello`. You can set the value of `X-API-Key` header along with your request like the following:

```
curl --request GET \
  --url https://2kzkof.deta.dev/tut \
  --header 'Content-Type: application/json' \
  --header 'X-API-Key: awEGbjqg.D1sETRFJKHWtcxSRSmofY2-akZjqFPh7j'
```

That is all you need to protect your APIs!

Issues

If you run into any issues, consider reporting them in our Github Discussions.

Go to TOC

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

Deta Cron

> Cron is a feature of Deta Micros. For the cron to trigger, your code must be deployed on a Micro.

A Deta Micro can be set to run on a schedule from the `deta cli` using the deta cron set command.

In order to set a micro to run on a schedule, the micro's code needs to define a function that will be run on the schedule with the help of our library `deta`.

The `deta` library is pre-installed on a micro and can just be imported directly.

Set Cron

Use `deta cron set` to schedule the micro.

You can set the cron in two ways:

Rate

You can define the `rate` at which a micro should run. It is comprised of a `value` and `unit`.

- `value`: is a non-zero positive integer
- `unit`: unit of time, can be `minute, minutes, hour, hours, day, days`. If the `value` is `1` the unit must be `minute`, `hour` or `day`.

Examples

- `deta cron set 1 minute` : Run every minute
- `deta cron set 2 hours` : Run every two hours
- `deta cron set 5 days` : Run every five days

Cron expressions

Cron expressions allow you more flexibility and precision when setting a cron job. Cron expressions have six required fields, which are separated by white space.

Field	Values	Wildcards
Minutes	0-59	,-*/
Hours	0-23	,-*/
Day-of-month	1-31	,-*?/LW
Month	1-12 or JAN-DEC	,-*/
Day-of-week	1-7 or SUN-SAT	,-*?L#
Year	1970-2199	,-*/

Wildcards

- The , (comma) wildcard includes additional values. In the Month field, JAN,FEB,MAR would include January, February, and March.
- The - (dash) wildcard specifies ranges. In the Day field, 1-15 would include days 1 through 15 of the specified month.
- The *(asterisk) wildcard includes all values in the field. In the Hours field,* would include every hour. You cannot use * in both the Day-of-month and Day-of-week fields. If you use it in one, you must use ? in the other.
- The / (forward slash) wildcard specifies increments. In the Minutes field, you could enter 1/10 to specify every tenth minute, starting from the first minute of the hour (for example, the 11th, 21st, and 31st minute, and so on).
- The ? (question mark) wildcard specifies one or another. In the Day-of-month field you could enter 7 and if you didn't care what day of the week the 7th was, you could enter ? in the Day-of-week field.
- The L wildcard in the Day-of-month or Day-of-week fields specifies the last day of the month or week.
- The W wildcard in the Day-of-month field specifies a weekday. In the Day-of-month field, 3W specifies the weekday closest to the third day of the month.
- The # wildcard in the Day-of-week field specifies a certain instance of the specified day of the week within a month. For example, 3#2 would be the second Tuesday of the month: the 3 refers to Tuesday because it is the third day of each week, and the 2 refers to the second day of that type within the month.

Limits

- You can't specify the Day-of-month and Day-of-week fields in the same cron expression. If you specify a value (or a *) in one of the fields, you must use a ? (question mark) in the other.
- Cron expressions that lead to rates faster than 1 minute are not supported.

Examples

- `deta cron set 0 10 * * ? *` : Run at 10:00 am(UTC) every day
- `deta cron set 15 12 * * ? *` : Run at 12:15 pm(UTC) every day
- `deta cron set 0 18 ? * MON-FRI *` : Run at 6:00 pm(UTC) every Monday through Friday
- `deta cron set 0 8 1 * ? *` : Run at 8:00 am(UTC) every 1st day of the month
- `deta cron set 0/15 * * * ? *` : Run every 15 minutes
- `deta cron set 0/5 8-17 ? * MON-FRI *` : Run every 5 minutes Monday through Friday between 8:00 am and 5:55 pm(UTC)

Code

<Tabs groupId=preferred-language defaultValue=js values={[{label: 'JavaScript', value:'js', }, {label: 'Python', value: 'python',},]}

```
const { app } = require('deta');

// define a function to run on a schedule
// the function must take an event as an argument
app.lib.cron(event => running on a schedule);

module.exports = app;
```

```
from deta import app

# define a function to run on a schedule
# the function must take an event as an argument
@app.lib.cron()
def cron_job(event):
    return running on a schedule
```

With this code deployed on a deta micro, the `deta cron set` commands will execute the function based on the cron rate or expression. For example

```
$ deta cron set 10 minutes
```

will set the function to run every 10 minutes. In order to see the execution logs, you can use the visor

Events

A function that is triggered from the cron must take an `event` as the only argument. The `event` will have the following attribute.

- `event.type`: `string` type of an event, will be `cron` when triggered by a cron event

Remove Cron

Use `deta cron remove` to remove a schedule from the micro.

Cron and HTTP

You can combine both cron and HTTP triggers in the same deta micro. For this you need to instantiate your app using the `deta` library that is pre-installed on a micro.

<Tabs groupId=preferred-language defaultValue=js values={[{label: 'JavaScript', value:'js', }, {label: 'Python', value: 'python',},]}

```
const { App } = require('deta');
const express = require('express');
const app = App(express());

app.get('/', async(req, res) => {
    res.send('Hello Deta, I am running with HTTP');
});

app.lib.cron(event => {
    return 'Hello Deta, I am a cron job';
});

module.exports = app;
```

```
from deta import App
from fastapi import FastAPI

app = App(FastAPI())

@app.get(/)
def http():
    return Hello Deta, I am running with HTTP

@app.lib.cron()
def cron_job(event):
    return Hello Deta, I am a cron job
```

Issues

If you run into any issues, consider reporting them in our Github Discussions.

Go to TOC

You can set custom sub-domains and point your own domains to a Deta Micro. Navigate to the `domains` tab in your Micro's dashboard to set domains.

custom_domain

Sub-domains

Deta provides your Micro a random sub-domain by default under `deta.dev`. You can assign a different sub-domain of your choice to your Micro. The Micro can be accessed from both the default sub-domain and the sub-domain you assign.

You need to consider the following when assigning a sub-domain:

- You can only assign a single sub-domain for each Micro.
- The sub-domain must be at least `4` characters in length.
- Only alphanumeric characters, dashes and underscores are allowed. The sub-domain must begin and end with an alphanumeric character.

Custom Domains

You can assign custom domains to a Micro.

Assignment

You need to consider the following when assigning a custom domain:

- You can assign up to `5` custom domains for a single Micro.
- After you have added a custom domain, we provide you with an IP Address. You need to add an `A record` in your DNS settings and point your domain to this IP Address.
- We continously check if your domain is pointing to this IP Address. If it is, we will create an SSL certificate for your domain and serve your Micro from the domain.
- The time taken for the DNS record changes to take effect depends on your DNS settings and provider. It may take up to 24 hours for your domain to go live.
- If you get an SSL error after the DNS record changes have propagated, please wait until the system has created an SSL certificate for your domain. This is usually fast but sometimes it may take a few hours.
- If you have a CAA record set up for you domain, make sure you have set `sectigo.com` as an allowed CA for your domain.

> If you are using [Cloudflare](https://www.cloudflare.com) as your DNS provider, please make sure to turn the proxy off in your DNS settings when adding the Deta provided IP address.

Completion

Your domain will be marked as `active` in your dashboard after the process is complete.

CAA Records

CAA records specify which Certificate Authorities are allowed to issue certificates on your behalf.

If you have a CAA record set up for your domain (check your DNS settings or use a tool to check your CAA records), add the following record so that we can issue certificates for your domain.

```
example.com. 3600 IN CAA 0 issue sectigo.com
```

This record allows us to issue certificates for `example.com` and all subdomains of `example.com`. For your domain, replace `example.com` with your domain.

Troubleshoot

Check the following if you are experiencing issues in setting up a custom domain:

1. Check if your top level domain is supported.
2. Make sure you have added an A record in your DNS settings with our IP address. You can use tools like nslookup or other online tools to check what IP address your domain is currently being resolved to.
3. If you are using `Cloudflare`, make sure to turn the proxy off in your DNS settings when adding the Deta provided IP address.
4. Check if there is a CAA record set up for your domain. If yes, make sure you have set up the necessary CAA record.

> If you had to change something from the steps mentioned above, [email us]() or [let us know in discord](https://go.deta.dev/discord). This is important as our systems eventually stop trying to assign the custom domain on errors. Manual re-enabling of the domain might be required.

Unsupported Top Level Domains

Our SSL provider does not support some country code top level domains. Please check them out here.

Issues

If you run into any issues, consider reporting them in our Github Discussions.

Go to TOC

A Deta Micro is a micro server where you can deploy your Python or Node web app without worrying about ops.

Python

Most Python micro frameworks are supported, like FastAPI and Flask. Deta should run both pure WSGI & ASGI apps without any issue. Full frameworks like Django are not fully supported yet, as they require various other commands.

For Deta to be able to run your code, you need to call your app instance `app` and it has to be in a file called `main.py`, which is the only required file for a Python Micro. Of course you could add more files and folders.

No need to use a server like `uvicorn`, Deta has its own global server. Make sure you have the framework in your `requirements.txt`.

Example code

FastAPI

```
from fastapi import FastAPI

app = FastAPI()  # notice that the app instance is called `app`, this is very
important.

@app.get(/)
async def read_root():
    return {Hello: World}
```

Bottlepy

```
from bottle import Bottle

app = Bottle()  # notice that the app instance is called `app`, this is very
important.

@app.route('/')
def home():
    return 'Hello, World!'
```

Flask

```
from flask import Flask

app = Flask(__name__)  # notice that the app instance is called `app`, this is
very important.

@app.route(/)
def hello_world():
    return <p>Hello, World!</p>
```

Starlette

```python
from starlette.applications import Starlette
from starlette.responses import JSONResponse
from starlette.routing import Route


async def homepage(request):
    return JSONResponse({'hello': 'world'})


# notice that the app instance is called `app`, this is very important.
app = Starlette(debug=True, routes=[
    Route('/', homepage),
])
```

Node.js

Most Node.js micro frameworks are supported, like Express, Koa, etc.

For Deta to be able to run your code, you need to call your app instance `app` and it has to be in a file called `index.js`, which is the only required file for a Node Micro. You also need to export `app`. Of course you could add more files and folders.

Make sure you have the framework in your `package.json`.

Example code

Express.js

```javascript
const express = require('express');
const app = express(); // notice that the app instance is called `app`, this is
very important.

app.get('/', (req, res) => {
  res.send('Hello World!');
});

// no need for `app.listen()` on Deta, we run the app automatically.
module.exports = app; // make sure to export your `app` instance.
```

Koa.js

```javascript
const Koa = require('koa');
const app = new Koa();

app.use(async ctx => {
  ctx.body = 'Hello World';
});

// no need for `app.listen()` on Deta, we run the app automatically.
module.exports = app; // make sure to export your `app` instance.
```

Fastify.js

```
const app = require('fastify')();

// Declare a route
app.get('/', async (request, reply) => {
  return { hello: 'world' };
});

// no need for `app.listen()` on Deta, we run the app automatically.
module.exports = app; // make sure to export your `app` instance.
```

Issues

If you run into any issues, consider reporting them in our Github Discussions.

Go to TOC

The **Deploy to Deta button** provides users a quick way to deploy a public Git repo to a Deta Micro directly from the browser.

It's an easy way to share your creations with the developer community.

You can use the Deploy To Deta Button in open-source repositories, blog posts, landing pages etc, allowing other developers to quickly deploy and use your app on Deta.

An example button that deploys a sample Python Micro to Deta:

Deploy

Usage

You can let users deploy your GitHub, Gitlab BitBucket (Git) repo quickly by adding the following markup:

```
[![Deploy](https://button.deta.dev/1/svg)](https://go.deta.dev/deploy?repo=your-
repo-url)
```

> Specify the __exact__ branch url if you want to use a different branch for deployment. If you provide the repository url without specifying a branch, the default branch will be used.
>
> The repository **must be a public git repository url.**

-->

Usage with HTML/JavaScript

The button image is hosted in the following url:

```
https://button.deta.dev/1/svg
```

and can be easily added to HTML pages or JavaScript applications. Example usage:

```
<a href=https://go.deta.dev/deploy?repo=your-repo-url>
    <img src=https://button.deta.dev/1/svg alt=Deploy>
</a>
```

Metadata, Environment Variables And Cron

You can optionally specify metadata, environment variables and a cron expression needed by the Micro in a `deta.json` file. Create this file in the root directory of your application.

The `deta.json` file has the following schema:

```
{
    name: your app name,
    description: your app description,
    runtime: micro's runtime,
    env: [
        {
            key: ENV_VAR_KEY,
            description: Human friendly description of the env var,
            value: Default value of the env var,
            required: true|false
        }
    ],
    cron: default cron expression (for eg. 3 minutes)
}
```

Fields

- `name` : the application name, we will use the repositry name by default if this is not provided, users can change this value during deployment
- `description` : the application description which will be shown to users on the deployment page
- `runtime` : the micro's runtime, the default runtime (`python3.9` or `nodejs14.x`) will be used unless this value is set, supported values:
  - `Python` : `python3.7` , `python3.8` , `python3.9`
  - `Nodejs` : `nodejs12.x` , `nodejs14.x`
- `env` : the environment variables your app uses
  - `key` : the environment variable key, users cannot change this value
  - `description` : the description of the environment variable so users know what it is used for
  - `value` : the default value of this variable, users can change this value
  - `required` : if the value of this variable *must be set* by the user
- `cron` : the default cron expression for the micro, if this is provided the deployed micro will have a cron job set with the provided value by default, users can change the value during deployment

You can test your `deta.json` file by visiting `https://go.deta.dev/deploy?repo=your-repo-url`

Get discovered

Make sure to add the `deta` tag to your repo for it to show up in our GitHub topic.

Deta

Issues

If you run into any issues, consider reporting them in our Github Discussions.

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

Setting Environment Variables

Environment variables can be set to your Micro through the Deta cli's update command. These variables are encrypted at rest.

You need to create a file where you specify your environment variables with their values.

> We strongly recommend creating a file called `.env` so that this file is skipped by the Deta cli and not deployed with your Micro's code. Also, please do not share/commit your env file publicly.

You can use the following command to update your environment variables:

```
deta update -e <env_file_name>
```

The environment variables must be defined in the following format:

```
VAR_NAME=VALUE
```

You should not change the value of the environment variables through your Micro's code, as the change will not persist in following invocations of your Micro.

Example

Suppose you want to make an environment variable `SECRET_TOKEN` with a value of `abcdefgh` accessible to your Micro's code.

- Create a `.env` file with the following content:

```
SECRET_TOKEN=abcdefg
```

- Update the Micro's environment variables with the `deta update` command:

```
deta update -e .env
```

Pre-Set Environment Variables

Deta pre-sets some environment variables in your Micro's environment which are used by our runtime.

> These pre-set variables are specific to your Micro and should not be shared publicly (unless we mention otherwise). These variables can not be updated from the cli.

The following pre-set environment variables could be useful for you:

- **DETA_PATH**

`DETA_PATH` stores a unique identifier for your Micro. This value is also used as the sub-domain under which your Micro is available through `http`. For example, if your `DETA_PATH` variable contained a value of `sd-fgh` then your Micro is accessible at https://sdfgh.deta.dev.

> This value does not store the self assigned sub-domain value at the moment.

- **DETA_RUNTIME**

`DETA_RUNTIME` indicates if your code is running on a Micro, with a string value `true`.

In some cases, you might need to run some code exclusively in local development and not in a Micro. For example, if you're running a FastAPI project on your Micro, and you've got an endpoint that dumps debug information like so:

```
@app.get(/debug_info)

async def debug_info():

    return get_debug_info()
```

If you want this endpoint to be only available locally, you can check the environment variable to selectively execute code:

```
import os

@app.get(/debug_info)

async def debug_info():

    if (not os.getenv('DETA_RUNTIME')):
        return get_debug_info()

    raise HTTPException(status_code=404, detail=Not Found)
```

- **DETA_PROJECT_KEY**

`DETA_PROJECT_KEY` is a project key generated for your Micro for authentication with Deta Base and Deta Drive.

Each micro has its own project key and this **should not be shared publicly**.

While using our SDKs(the latest versions) within a Deta Micro, you can omit specifying the project key when instantiating a service instance.

<Tabs groupId=preferred-language defaultValue=js values={[{label: 'JavaScript', value:'js', }, {label: 'Python', value: 'python',},]}

```javascript
const { Base, Drive } = require('deta');

const base = Base('base_name');
const drive = Drive('drive_name');
```

```python
from deta import Base, Drive

base = Base('base_name')
drive = Drive('drive_name')
```

Issues

If you run into any issues, consider reporting them in our Github Discussions.

Go to TOC

How can I request for a timeout or memory increase?

Please, use the following forms to request for respective limit increases:

- Timeout Limit Increase Request
- Memory Limit Increase Request

Can I use my own custom domain or subdomain with a Micro?

Yes, please refer to the documentation.

Do Micros support websockets?

No, Micros do not support websockets and other long-running processes.

Why can I not write to the filesystem in a Micro?

Micros have a read-only filesystem. Only `/tmp` can be written to, which is ephemeral and has a storage limit of 512 Mb.

Why am I getting a `Request Entity Too Large` error when deploying a Micro?

Micros have a maximum deployment size limit of 250 Mb. This limit includes the total size of your code and dependencies.

Why does my Micro behave different locally and deployed?

This can be because of a number of reasons. Please check out the specifications and notes for further information to find a possible reason.

Why is my Micro returning a 502 Bad Gateway?

The response comes from a reverse proxy between a client and your Micro. If there is a runtime error or a timeout when invoking a Micro, the proxy responds with a `502 Bad Gateway.`

In order to debug `502` responses, you can use `deta visor` which show you real-time logs of your application. Navigate to your Micro's visor page from the UI or use the deta cli command `deta visor open` to view the error logs.

If there are no logs in your visor and your Micro's response took roughly 10 seconds, then it's likely because of a timeout.

If it is not because of a timeout, then please contact us.

Is it safe to commit the `.deta` folder created by the cli?

Yes it is safe to commit your `.deta` folder and push it to public repositories.

Go to TOC

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

Configuring & Installing the Deta CLI

<Tabs defaultValue=mac values={[{ label: 'Mac', value: 'mac', }, { label: 'Linux', value: 'linux', }, { label: 'Windows', value: 'win', },] }>

To install the Deta CLI, open a Terminal and enter:

```
curl -fsSL https://get.deta.dev/cli.sh | sh
```

To install the Deta CLI, open a Terminal and enter:

```
curl -fsSL https://get.deta.dev/cli.sh | sh
```

To install the Deta CLI, open PowerShell and enter:

```
iwr https://get.deta.dev/cli.ps1 -useb | iex
```

This will download the binary which contains the CLI code. It will try to export the `deta` command to your path. If it does not succeed, follow the directions output by the install script to export `deta` to your path.

Logging in to Deta via the CLI

Once you have successfully installed the Deta CLI, you need to login to Deta.

From a Terminal, type `deta login`.

This command will open your browser and authenticate your CLI through Deta's web application.

Upon a successful login, you are ready to start building Micros.

Creating Your First Micro

To create a micro, navigate in your Terminal to a parent directory for your first micro and type:

<Tabs groupId=preferred-language defaultValue=py values={[{ label: 'JavaScript', value: 'js', }, { label: 'Python', value: 'py', },] }>

```
deta new --node first_micro
```

This will create a new Node.js Micro in the 'cloud' as well as a local copy inside a directory called `first_micro` which will contain an `index.js` file.

The CLI should respond:

```
{
    name: first_micro,
    runtime: nodejs12.x,
    endpoint: https://<path>.deta.dev,
    visor: enabled,
    http_auth: enabled
}
```

```
deta new --python first_micro
```

This will create a new Python Micro in the 'cloud' as well as a local copy inside a directory called `first_micro` which will contain a `main.py` file.

The CLI should respond:

```
{
    name: first_micro,
    runtime: python3.7,
    endpoint: https://<path>.deta.dev,
    visor: enabled,
    http_auth: enabled
}
```

Save this endpoint URL somewhere, as we will be visiting it shortly.

Creating a Micro Under a Specific Project

```
deta new --project <your-project>
```

This will create a new Micro under `<your-project>` in the 'cloud'.

Updating your Micro: Dependencies and Code

<Tabs groupId=preferred-language defaultValue=py values={[{ label: 'JavaScript', value: 'js', }, { label: 'Python', value: 'py', },] }>

Setup and Dependencies

Enter the directory `first_micro`, and then run the shell command:

```
npm init -y
```

This will initialize a Node.js project in your current directory with npm's wizard.

What is important is that the **main** file is `index.js`.

After following the npm wizard, let's add a dependency by running:

```
npm install express
```

Updating Code Locally

Let's also edit and save the `index.js` file locally so that your Micro responds to `HTTP GET` requests with *Hello World*.

```
const express = require('express');

const app = express();

app.get('/', async (req, res) => {
  res.send('Hello World')
});

module.exports = app;
```

Deploying Local Changes

After you have updated your dependencies (documented in a `package.json` file) and / or your code locally, use a `deta deploy` command to update your Micro.

```
deta deploy
```

The Deta CLI will notify you if your code has updated as well as if the dependencies were installed

```
Deploying...
Successfully deployed changes
Updating dependencies...

+ express@4.17.1
added 50 packages from 37 contributors and audited 50 packages in 1.967s
found 0 vulnerabilities
```

Setup and Dependencies

Enter the directory `first_micro`, and then create a file, `requirements.txt`, which tells Deta which dependencies to install.

Let's add *flask* to `requirements.txt` and save this file locally.

```
flask
```

Updating Code Locally

Let's also edit the `main.py` file so that your Micro responds to `HTTP GET` requests with *Hello World*.

```
from flask import Flask

app = Flask(__name__)

@app.route('/', methods=[GET])
def hello_world():
    return Hello World
```

Deploying Local Changes

After you have updated your `requirements.txt` and / or your code locally, use a `deta deploy` command to update your Micro.

```
deta deploy
```

The Deta CLI will notify you if your code has updated as well as if the dependencies were installed.

```
Deploying...
Successfully deployed changes
Updating dependencies...
Collecting flask
  Downloading
https://files.pythonhosted.org/packages/f2/28/2a03252dfb9ebf377f40fba6a7841b47083260bf8bd8e73
1.1.2-py2.py3-none-any.whl (94kB)
Collecting Werkzeug>=0.15 (from flask)
  Downloading
https://files.pythonhosted.org/packages/cc/94/5f7079a0e00bd6863ef8f1da638721e9da21e5bacee5975
1.0.1-py2.py3-none-any.whl (298kB)
Collecting itsdangerous>=0.24 (from flask)
  Downloading
https://files.pythonhosted.org/packages/76/ae/44b03b253d6fade317f32c24d100b3b35c2239807046a4c
1.1.0-py2.py3-none-any.whl
Collecting Jinja2>=2.10.1 (from flask)
  Downloading
https://files.pythonhosted.org/packages/30/9e/f663a2aa66a09d838042ae1a2c5659828bb9b41ea3a6efa
2.11.2-py2.py3-none-any.whl (125kB)
Collecting click>=5.1 (from flask)
  Downloading
https://files.pythonhosted.org/packages/d2/3d/fa76db83bf75c4f8d338c2fd15c8d33fdd7ad23a9b5e57e
7.1.2-py2.py3-none-any.whl (82kB)
Collecting MarkupSafe>=0.23 (from Jinja2>=2.10.1->flask)
  Downloading
https://files.pythonhosted.org/packages/98/7b/ff284bd8c80654e471b769062a9b43cc5d03e7a615048d9
1.1.1-cp37-cp37m-manylinux1_x86_64.whl
Installing collected packages: Werkzeug, itsdangerous, MarkupSafe, Jinja2, click,
flask
```

Visiting our Endpoint

Let's visit the endpoint (from the endpoint URL we saved earlier).

(If you didn't save it, simply type `deta details` into the CLI, which will give you the endpoint alongside other information about your Micro).

Open up your endpoint in a browser. You might be prompted to log in to your deta account on visiting your endpoint for the first time.

You should see **Hello, World**

If you're not accessing the endpoint from a browser (like from curl) or if you have disabled cookies in your browser, the response will be:

```
{
    errors:[Unauthorized]
}
```

This is because *Deta Auth* is protecting the endpoint from unauthorized access.

Opening Your Micro To the Public

Let's use one last command to open up the endpoint to the public:

```
deta auth disable
```

The CLI should respond:

```
Successfully disabled http auth
```

Congratulations, you have just deployed and published your first Micro!

Issues

If you run into any issues, consider reporting them in our Github Discussions.

Go to TOC

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

Run from the CLI

A Deta Micro can be run directly from the `deta cli` using the command deta run with an input.

In order to run a micro from the cli directly, the micro's code needs to define functions that will be run from the cli with the help of our library `deta`.

The `deta` library is pre-installed on a micro and can just be imported directly.

<Tabs groupId=preferred-language defaultValue=js values={[{label: 'JavaScript', value:'js', }, {label: 'Python', value: 'python',},]}

```
const { app } = require('deta');

// define a function to run from the cli
// the function must take an event as an argument
app.lib.run(event => Welcome to Deta!);

module.exports = app;
```

```
from deta import app

# define a function to run from the cli
# the function must take an event as an argument
@app.lib.run()
def welcome(event):
    return Welcome to Deta!
```

With this code deployed on a micro, you can simply run

```
$ deta run
```

And see the following output:

```
Response:
    Welcome to Deta!
```

Events

A function that is triggered from the cli must take an `event` as the only argument.

You can provide an input from the cli to the function which will be passed on as an `event`. It has four attributes:

- `event.json` : `object` provides the JSON payload
- `event.body` : `string` provides the raw JSON payload
- `event.type` : `string` type of an event, `run` when running from the cli
- `event.action` : `string` the action provided from the cli, defaults to an empty string

<Tabs groupId=preferred-language defaultValue=js values={[{label: 'JavaScript', value:'js', }, {label: 'Python', value: 'python',},]}

```
const { app } = require('deta');

app.lib.run(event => {
    return {
        // access input to your function with event.json
        message: `hello ${event.json.name}!`
    };
});

module.exports = app;
```

```
from deta import app

@app.lib.run()
def welcome(event):
    return {
        # access input to your function with event.json
        message: fhello {event.json.get('name')}!
    }
```

With this code deployed on a micro, you can run

```
$ deta run -- --name deta
```

And should see the following output.

```
Response:
    {
        message: hello deta!
    }
```

Input

The input to your function on a micro can be provided through the `deta cli` and accessed in the code from the `event` object. The input is a JSON object created from the arguments provided to the cli.

An *important* consideration is that the values in key-value pairs in the input are always either *strings*, *list of strings* or *booleans*.

Boolean flags are provided with a single dash, string arguments with double dash and if multiple values are provided for the same key, a list of strings will be provided.

For instance:

```
$ deta run -- --name jimmy --age 33 --emails jimmy@deta.sh --emails jim@deta.sh -
active
```

will provide the micro with the following input:

```
{
    name: jimmy,
    age: 33, // notice '33' here is a string not an int
    emails: [jimmy@deta.sh, jim@deta.sh],
    active: true
}
```

You need to explicitly convert the string values to other types in your code if needed.

Actions

Actions help you run different functions based on an `action` that you define for the function.

The `action` defaults to an empty string if not provided.

<Tabs groupId=preferred-language defaultValue=js values={[{label: 'JavaScript', value:'js', }, {label: 'Python', value: 'python',},]}

```
const { app } = require('deta');

app.lib.run(event => {
    return {
        message: `hello ${event.json.name}!`
    };
    // action 'hello'
}, hello);

app.lib.run(event => {
    return {
        message: `good morning ${event.json.name}!`
    };
    // action 'greet'
}, greet);

module.exports = app;
```

```
from deta import app

# action 'hello'
# the action does not need to have the same name as the function
@app.lib.run(action=hello)
def welcome(event):
    return {
        message: fhello {event.json.get('name')}!
    }

# action 'greet'
@app.lib.run(action=greet)
def greet(event):
    return {
        message: fgood morning {event.json.get('name')}!
    }
```

With this code deployed on a deta micro, if you run

```
$ deta run hello -- --name deta
```

where you tell the cli to run action `hello` with `name: deta` as input. You should see the following output:

```
Response:
    {
        message: hello deta!
    }
```

And if you do `deta run` with action `greet`

```
$ deta run greet -- --name deta
```

you should see the following output:

```
Response:
    {
        message: good morning deta!
    }
```

Run and HTTP

You can combine both run and HTTP triggers in the same deta micro. For this you need to instantiate your app using the `deta` library that is pre-installed on a micro.

<Tabs groupId=preferred-language defaultValue=js values={[{label: 'JavaScript', value:'js', }, {label: 'Python', value: 'python',},]}

```javascript
const { App } = require('deta');
const express = require('express');

const app = App(express());

// triggered with an HTTP request
app.get('/', async(req, res) => {
    res.send('Hello deta, i am running with HTTP');
});

// triggered from the cli
app.lib.run(event => {
    return 'Hello deta, i am running from the cli';
});

module.exports = app;
```

```python
from deta import App
from fastapi import FastAPI

app = App(FastAPI())

# triggered with an HTTP request
@app.get(/)
def http():
    return Hello deta, i am running with HTTP

# triggered from the cli
```

```
@app.lib.run()
def run(event):
    return Hello deta, i am running from the cli
```

Run and Cron

You can use both run and cron triggers in the same deta micro. You can also stack run and cron triggers for the same function.

<Tabs groupId=preferred-language defaultValue=js values={[{label: 'JavaScript', value:'js', }, {label: 'Python', value: 'python',},]}>

```
const { app } = require('deta');

const sayHello = event => 'hello deta';
const printTime = event => `it is ${(new Date).toTimeString()}`;

app.lib.run(sayHello);

// stacking run and cron
app.lib.run(printTime, 'time'); // action 'time'
app.lib.cron(printTime);

module.exports = app;
```

```
from deta import app
from datetime import datetime

# only run
@app.lib.run()
def say_hello(event):
    return hello deta

# stacking run and cron
@app.lib.run(action='time') # action 'time'
@app.lib.cron():
def print_time(event):
    return fit is {datetime.now()}
```

Issues

If you run into any issues, consider reporting them in our Github Discussions.

Go to TOC

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

The Deta Visor offers a user interface for you to see a live view of all the events (logs) processed by your Micro. You can use it to monitor and debug your requests, test your endpoints by sending requests, and more.

By default, Visor is disabled, to enable it run `deta visor enable` in your terminal inside you Micro's source code.

Opening Visor

<Tabs defaultValue=browser values={[{ label: 'Browser', value: 'browser' }, { label: 'Terminal', value: 'terminal' },] }>

Start by logging into your console at deta.sh. You should be redirected to the following screen:

visor_1

Once that's done, choose a micro to view by clicking on its name (under the *Micros* tab).

visor_2

Now click on *Visor*

visor_3

Start your terminal in the root directory of your deployed Deta project.

Run `deta visor open`.

Navigating the Visor

The Visor page has the link to your Micro in the top right, a button named *HTTP Client* in the bottom left, and a list of all the events handled by your Micro!

visor_4

The Event List

The Visor logs every single event made to your Micro, each event has the **HTTP Status Code**, the **HTTP Method**, and the **time** logged above it. Under this information are three tabs, letting you view the response, request, and logs of the event.

Response: visor

The Request sent to the Micro:

request

And logs (normally displaying errors if they were to occur):

logs

On the top right of every event are two icons:

icons

The *pen* icon lets you edit the request before re-sending it to your micro and the *redo* icon lets you send the same request again.

The HTTP Client

Visor also has a built in *HTTP Client* to make requests to your Micro. You can use this by clicking on the *HTTP Client* button.

HTTP

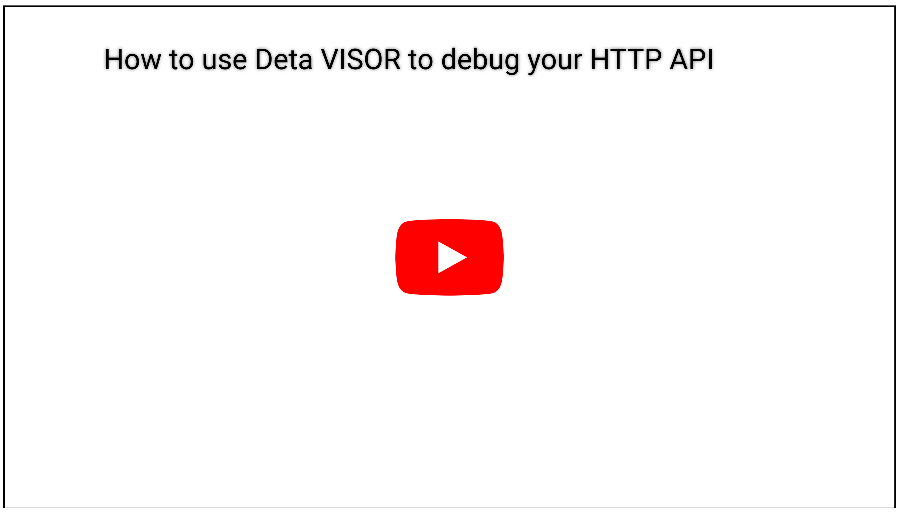
Simply specify the request method, path, headers and body and hit the *Send* button to send it off!

HTTP

The content-type can be changed from JSON to Text, using the selection box on the bottom left of this window.

HTTP

Video overview



Logs Retention

Visor logs are retained for two weeks (14 days).

Visor doesn't show the error

Sometimes you will get a `500` response from our server and no errors will show up in Visor, in that case you could quickly check our system logs for you Micro with `deta logs`.

Issues

If you run into any issues, consider reporting them in our Github Discussions.

Go to TOC

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

A Deta Micro can be set to run on a schedule from the `deta cli` using the deta cron set command.

In this guide we are going to configure a deta micro to run every five minutes.

The guide assumes you have already signed up for Deta and have the Deta CLI installed.

<Tabs groupId=preferred-language defaultValue=js values={[{ label: 'JavaScript', value: 'js', }, { label: 'Python', value: 'py', },] }>

1. Create a directory `cron-job` and change the current directory to it.

```
$ mkdir cron-job && cd cron-job
```

2. In order to set a micro to run on a schedule, the micro's code needs to define a function that will be run on the schedule with the help of our library `deta`, which is pre-installed on a micro and can just be imported directly.

   Create a `index.js` file and define the function that should run on the schedule.

```
// import app from 'deta'
const { app } = require('deta');

// use app.lib.cron to define the function that runs on the schedule
// takes an `event` as an argument
app.lib.cron(event => {
    console.log(running on a schedule);
});

module.exports = app;
```

3. Create a new `nodejs` micro with `deta new`.

```
$ deta new
Successfully created a new micro
{
    name: cron-job,
    runtime: nodejs12.x,
    endpoint: https://{micro_name}.deta.dev,
    visor: enabled,
    http_auth: enabled
}
```

   Even though the output shows a HTTP endpoint, you do not need it for cron.

4. Set the micro to run on a schedule with the deta cron set command.

```
$ deta cron set 5 minutes
Scheduling micro...
Successfully set micro to schedule for 5 minutes
```

   We have set the micro to run every five minutes. In order to see if a micro is running on a schedule, you can use the `deta details` command.

```
$ deta details
{
    name: cron-job,
    runtime: nodejs12.x,
    endpoint: https://{micro_name}.deta.dev,
    visor: enabled,
    http_auth: enabled,
    cron: 5 minutes
}
```

The details show that the `cron` has ben set for `5 minutes`

5. In order to see logs of your cron-job, you can use `Deta Visor`, which enables you to see real-time logs of a Deta Micro. Open your micro's visor page with `deta visor open` from the cli or by navigating to your micro's visor page on the browser.

```
$ deta visor open
Opening visor in the browser...
```

We have a micro which prints running on a schedule every five minutes. You should see the execution logs in your micro's visor page every five minutes.

1. Create a directory `cron-job` and change the current directory to it.

```
$ mkdir cron-job && cd cron-job
```

2. In order to set a micro to run on a schedule, the micro's code needs to define a function that will be run on the schedule with the help of our library `deta`, which is pre-installed on a micro and can just be imported directly.

   Create a `main.py` file and define the function that should run on the schedule.

```
from deta import app

# use app.lib.cron decorator for the function that runs on the schedule
# the function takes an `event` as an argument
@app.lib.cron()
def cron_task(event):
    print(running on a schedule)
```

3. Create a new `python` micro with `deta new`.

```
$ deta new
Successfully created a new micro
{
    name: cron-job,
    runtime: python3.7,
    endpoint: https://{micro_name}.deta.dev,
    visor: enabled,
    http_auth: enabled
}
```

   Even though the output shows a HTTP endpoint, you do not need it for cron.

4. Set the micro to run on a schedule with deta cron set command.

```
$ deta cron set 5 minutes
Scheduling micro...
Successfully set micro to schedule for 5 minutes
```

We have set the micro to run every five minutes. In order to see if a micro is running on a schedule, you can use the `deta details` command.

```
$ deta details
{
    name: cron-job
    runtime: python3.7,
    endpoint: https://{micro_name}.deta.dev,
    visor: enabled,
    http_auth: enabled,
    cron: 5 minutes
}
```

The details show that the `cron` has ben set for `5 minutes`

5. In order to see logs of your cron-job, you can use `Deta Visor`, which enables you to see real-time logs of a Deta Micro. Open your micro's visor page with `deta visor open` from the cli or by navigating to your micro's visor page on the browser.

```
$ deta visor open
Opening visor in the browser...
```

We have successfully deployed a micro which prints running on a schedule every five minutes. You should see the execution logs in your micro's visor page every five minutes.

Full documentation on cron is available here.

Go to TOC

You can deploy a FastAPI application on Deta very easily in a few steps.

The guide assumes you have already signed up for Deta and have the Deta CLI installed.

1. Create a directory `fastapi-app` and change the current directory to it.

```
$ mkdir fastapi-app && cd fastapi-app
```

2. Create a `main.py` file with a simple fastapi application.

```
from fastapi import FastAPI
app = FastAPI()

@app.get(/)
async def root():
    return Hello World!
```

3. Create a `requirements.txt` file specifying `fastapi` as a dependecy.

```
fastapi
```

4. Deploy your application with `deta new`.

```
$ deta new
Successfully created a new micro
{
        name: fastapi-app,
        runtime: python3.7,
        endpoint: https://{micro_name}.deta.dev,
        visor: enabled,
        http_auth: enabled
}
Adding dependencies...
Collecting fastapi
...
Successfully installed anyio-3.6.1 fastapi-0.85.0 idna-3.4 pydantic-1.10.2
sniffio-1.3.0 starlette-0.20.4 typing-extensions-4.3.0
```

We now have a FastAPI application deployed.

If you visit the `endpoint` shown in the output (your endpoint will be different from this one) in your browser, you should see a `Hello World!` response.

Go to TOC

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

Deta Micros can be used to easily deploy form endpoints and the form data can be stored easily with Deta Base.

In this guide, we are going to deploy a Deta Micro that offers a form endpoint for a simple contact form and stores the information in a Deta Base.

The guide assumes you have already signed up for Deta and have the Deta CLI installed.

The contact form will store a `name`, `email` and a `message`.

```
{
    name: str,
    email: str,
    message: str
}
```

<Tabs groupId=preferred-language defaultValue=js values={[{label: 'JavaScript', value: 'js', }, {label: 'Python', value: 'py', },] }>

We are going to deploy an `express` app for the form endpoint.

1. Create a directory `express-form` and change the current directory to it.

   ```
   $ mkdir express-form && cd express-form
   ```

2. Create an empty `index.js` file (we will add the code that handles the logic later).

3. Initialize a `nodejs` project with `npm init`.

   ```
   $ npm init -y
   ```

   You can skip the `-y` flag, if you want to fill the details about the pacakge interactively through npm's wizard.

4. Install `express` locally for your project.

   ```
   $ npm install express
   ```

5. Create a new `nodejs` micro with `deta new`. This will create a new `nodejs` micro for you and automatically install `express` as a dependecy.

   ```
   $ deta new
   Successfully created a new micro
   {
       name: express-form,
       runtime: nodejs12.x,
       endpoint: https://{micro_name}.deta.dev,
       visor: enabled,
       http_auth: enabled
   }
   Adding dependencies...
   ```

```
+ express@4.17.1
added 50 packages from 37 contributors and audited 50 packages in 1.967s
found 0 vulnerabilities
```

Your micro's `endpoint` will be different from the output shown above. This `endpoint` will be the form endpoint.

6. You can also see that the `http_auth` is `enabled` by default. We will disable the `http_auth` so that form data can be sent to the micro.

```
$ deta auth disable
Successfully disabled http auth
```

This is only for the tutorial, we recommended that you protect your endpoint with some authentication.

7. Open `index.js` and add a `POST` endpoint for the form using `express`.

```
const express = require('express');

// express app
const app = express();

// parse application/x-www-form-urlencoded
app.use(express.urlencoded());

app.post(/, (req, res)=>{
    const body = req.body;

    // validate input
    if (!body.name || !body.email || !body.message){
        return res.status(400).send('Bad request: missing some required
values');
    }

    // TODO: save the data
    return res.send('ok');
});
```

8. Update `index.js` to add logic to save the form data to a Deta Base.

```
const express = require('express');
const { Deta } = require('deta');

// express app
const app = express();

// contact form base
const formDB = Deta().Base(contact-form);

// parse application/x-www-form-urlencoded
app.use(express.urlencoded());

app.post(/, (req, res)=>{
    const body = req.body;

    // validate input
    if (!body.name || !body.email || !body.message){
        return res.status(400).send('Bad request: missing some required values')
    }

    // save form data
    formDB.put({
        name: body.name,
        email: body.email,
        message: body.message
    });

    return res.status(201).send('ok');
});
```

9. Deploy the changes with `deta deploy`

```
$ deta deploy
Successfully deployed changes
```

Your endpoint will now accept form `POST` data and save it to a database.

You can use Deta Base's UI to easily see what data has been stored in the database. Navigate to your `project` and click on your base name under `bases` in your browser to use the Base UI.

Here's an example view of the UI.

We are going to deploy a `flask` app for the form endpoint.

1. Create a directory `flask-form` and change the current directory to it.

```
$ mkdir flask-form && cd flask-form
```

2. Create an empty `main.py` file (we will add the code that handles the logic later).

3. Create a `requirements.txt` file and add `flask` as a dependency for your project.

```
flask
```

4. Create a new `python` micro with `deta new`. This will create a new `python` micro for you and automatically install `flask` as a dependecy.

```
$ deta new
Successfully created a new micro
{
    name: flask-form,
    runtime: python3.7,
    endpoint: https://{micro_name}.deta.dev,
    visor: enabled,
    http_auth: enabled
}
Adding dependencies...
Collecting flask
...
Installing collected packages: Werkzeug, itsdangerous, MarkupSafe, Jinja2,
click, flask
```

Your micro's `endpoint` will be different from the output shown above. This `endpoint` will be the form endpoint.

5. You can also see that the `http_auth` is `enabled` by default. We will disable the `http_auth` so that form data can be sent to the micro.

```
$ deta auth disable
Successfully disabled http auth
```

   This is only for the tutorial, we recommended that you protect your endpoint with some authentication.

6. Open `main.py` and add a `POST` endpoint for the form using `flask`.

```
from flask import Flask, request

#flask app
app = Flask(__name__)

@app.route(/, methods=[POST])
def save_form_data():
    # TODO: save data
    return ok
```

7. Update `main.py` to add logic to save the form data to a Deta Base.

```
from flask import Flask, request
from deta import Deta

# flask app
app = Flask(__name__)

# contact form base
form_db = Deta().Base(contact-form)

@app.route(/, methods=[POST])
def save_form_data():
    form_db.put({
        # flask sends a 400 automatically if there is a KeyError
        name: request.form[name],
        email: request.form[email],
        message: request.form[message]
    })
    return ok, 201
```

8. Deploy the changes with `deta deploy`.

```
$ deta deploy
Successfully deployed changes
```

Your endpoint will now accept form `POST` data and save it to a database.

You can use Deta Base's UI to easily see what data has been stored in the database. Navigate to your `project` and click on your base name under `bases` in your browser to use the Base UI.

Here's an example view of the UI.

Go to TOC

import Tabs from '@theme/Tabs'; import TabItem from '@theme/TabItem';

Deta Micros make it extremely easy to deploy webhook servers.

Each Deta Micro has a unique HTTPS endpoint which can be used as the webhook URL.

You can use your favorite web application framework for your webhook server.

The guide assumes you have already signed up for Deta and have the Deta CLI installed.

<Tabs groupId=preferred-language defaultValue=js values={[{label: 'JavaScript', value: 'js', }, {label: 'Python', value: 'py', },] }>

We are going to use `express` with our Deta Micro and deploy a simple `nodejs` webhook server.

1. Create a directory `express-webhook` and change the current directory to it.

   ```
   $ mkdir express-webhook && cd express-webhook
   ```

2. Create an empty `index.js` file (we will add the code that handles the logic later).

3. Initialize a `nodejs` project with `npm init`

   ```
   $ npm init -y
   ```

   You can skip the `-y` flag, if you want to fill the details about the package interactively through npm's wizard.

4. Install `express` locally for your project.

   ```
   $ npm install express
   ```

5. Create a new `nodejs` micro with `deta new`. This will create a new `nodejs` micro for you and automatically install `express` as a dependency.

   ```
   $ deta new
   Successfully created a new micro
   {
       name: express-webhook,
       runtime: nodejs12.x,
       endpoint: https://{micro_name}.deta.dev,
       visor: enabled,
       http_auth: enabled
   }
   Adding dependencies...

   + express@4.17.1
   added 50 packages from 37 contributors and audited 50 packages in 1.967s
   found 0 vulnerabilities
   ```

   Your micro's `endpoint` will be different from the output shown above. This `endpoint` will be the webhook URL when you set up your webhooks.

6. You can also see that the `http_auth` is `enabled` by default. We will disable the `http_auth` so that webhook events can be sent to the micro.

```
$ deta auth disable
Successfully disabled http auth
```

> Usually for security, a secret token is shared between the sender and the receiver and a hmac signature is calculated from the payload and the shared secret. The algorithm to calculate the signature varies based on the sender. You should consult the documentation of how the signature is calculated for the system you are receiving events from.

7. Add a `POST` route to `index.js`. Most webhook callbacks send a `POST` request on the webhook URL with the payload containing information about the event that triggered the webhook.

   Open `index.js` and add the handler to handle webhook events.

```
const express = require('express');

const app = express();

// a POST route for our webhook events
app.post('/', (req, res) => {
    // verify signature if needed
    // add your logic to handle the request
    res.send(ok);
});

// you only need to export your app for the deta micro. You don't need to start
the server on a port.
module.exports = app;
```

8. Deploy your changes with `deta deploy`.

```
$ deta deploy
Successfully deployed changes
```

Your micro will now trigger on `POST` requests to the micro's endpoint. Use your micro's endpoint as the webhook URL when you set up your webhooks elsewhere.

In order to see the logs when the webhook is triggered, you can use `deta visor` to see real-time logs of your application. You can also replay your webhook events from `deta visor` to debug any issues.

You can open your visor page with the command `deta visor open` from the cli.

We are going to use `fastapi` with our Deta Micro and deploy a simple `python` webhook server.

1. Create a directory `fastapi-webhook` and change the current directory to it.

```
$ mkdir fastapi-webhook && cd fastapi-webhook
```

2. Create an empty `main.py` file (we will add the code that handles the logic later).

3. Create a `requirements.txt` file and add `fastapi` as a dependency.

```
fastapi
```

4. Create a new `python` micro with `deta new`. This will create a new `python` micro for you and automatically install `fastapi` as a dependency.

```
$ deta new
Successfully created a new micro
{
    name: fastapi-webhook,
    runtime: python3.7,
    endpoint: https://{micro_name}.deta.dev,
    visor: enabled,
    http_auth: enabled
}
Adding dependencies...
Collecting fastapi
...
Successfully installed fastapi-0.61.1 pydantic-1.6.1 starlette-0.13.6
```

Your micro's `endpoint` will be different from the output shown above. This `endpoint` will be the webhook URL when you set up your webhooks.

5. You can also see that the `http_auth` is `enabled` by default. We will disable the `http_auth` so that webhook events can be sent to the micro.

```
$ deta auth disable
Successfully disabled http auth
```

Usually for security, a secret token is shared between the sender and the receiver and a hmac signature is calculated from the payload and the shared secret. The algorithm to calculate the signature varies based on the sender. You should consult the documentation of how the signature is calculated for the system you are receiving events from.

6. Add a `POST` route to `main.py`. Most webhook callbacks send a `POST` request on the webhook URL with the payload containing information about the event that triggered the webhook.

Open `main.py` and add the handler to handle webhook events.

```
from fastapi import FastAPI, Request

app = FastAPI()

# a POST route for our webhook events
@app.post(/)
def webhook_handler(req: Request):
    # verify signature if needed
    # add logic to handle the request
    return ok
```

7. Deploy your changes with `deta deploy`.

```
$ deta deploy
Successfully deployed changes
```

Your micro will now trigger on `POST` requests to the micro's endpoint. Use your micro's endpoint as the webhook URL when you set up your webhooks elsewhere.

In order to see the logs of when the webhook is triggered, you can use `deta visor` to see real-time logs of your application. You can also replay your webhook events from `deta visor` to debug any issues.

You can open your visor page with the command `deta visor open` from the cli.

Go to TOC

Colophon

This book is created by using the following sources:

- Deta - English
- GitHub source: deta/docs/docs
- Created: 2022-12-01
- Bash v5.2.2
- Vivliostyle, https://vivliostyle.org/
- By: @shinokada
- Viewer: https://read-html-download-pdf.vercel.app/
- GitHub repo: https://github.com/shinokada/markdown-docs-as-pdf
- Viewer repo: https://github.com/shinokada/read-html-download-pdf