Daybed Documentation

Release 1

Spiral Project

May 28, 2015

Contents

Daybed is an Open Source Web API providing validation and storage as a service.

CHAPTER 1

Why ?

Usually, when a Web application requires a backend for storage and validation, one of the following solutions is employed :

- A custom API is developed and deployed (*reinvent the wheel*)
- A commercial and closed-source service is used (*Google Forms*)
- A CMS is used as a backoffice for customizable content types (*twist*)

To avoid those situations and make it easy to get your storage backend ready in seconds, we've created *Daybed*.

Daybed is :

- a minimalist, robust, dynamic and generic API ;
- a validation layer with schemaless storage ;
- a reusable layer of permissions logic ;
- a universal REST endpoint for Web and mobile apps ;
- a key component for rapid application building ;
- a simple service deployed and integrated without coding.

CHAPTER 2

How ?

1. Create a model by posting its definition (*title, fields, ...*)
2. Define the permissions sets (*modify definition, create, update or delete records, ...*)
3. Use the allocated RESTful endpoint in your application (*GET, POST, PUT, ...*)
4. Store and query records !

Since *Daybed* talks REST and JSON, you can basically use it as a remote storage with any of your favorite technologies (*Python, Android, iOS, AngularJS, Ember.js, Backbone.js, etc.*).

Note: Currently, the authentication relies on Basic authentication and Hawk

CHAPTER 3

Use Cases

- Mobile apps
- Full JavaScript apps
- Online forms
- Data Wiki
- Collaborative Web mapping

Using *Daybed* in its first versions, we built:

- daybed-map, a *geo-pad* where you can create you own maps with custom fields, using backbone-forms
- A password manager
- A generic CRUD application using backbone-daybed ;
- A TODO list using jquery-spore ;

CHAPTER 4

Technologies

Daybed uses the following stack by default :

- Pyramid, a robust and powerful python Web framework
- Cornice, a REST framework for Pyramid
- Colander for the schema validation part
- Redis as the default persistence backend
- ElasticSearch as indexing and faceted search engine
- CouchDB as an alternative persistence backend

CHAPTER 5

Comparison

Daybed has many competitors, yet none of them shares the same approach or features set.

Strategy		Custom dedicated API	Generated code	Competitors		
Project	Daybed	Django REST framework, Restify, express, Struts...	Python Eve	Loop-back	Hoodie	Google Forms
Minimalist						
Validation						
Permissions						
Dynamic schemas						
Reusable instance						
Dynamic API end-points						
Raw data access						
Agnostic (REST)						
Requires SDK						
Open Source						
Faceted search						

Sources: http://python-eve.org ; http://hood.ie ; http://loopback.io ;

CHAPTER 6

More documentation

6.1 Installing Daybed

Daybed has the following requirements:

- A Python 2.6, 2.7, 3.x or PyPy installation
- A CouchDB or Redis, and an ElasticSearch server instance running

Daybed comes with a Makefile to simplify your life when developing. To install daybed in your current virtualenv and get started, just run:

```
$ make install
```

Then, running the test suite is a good way to check that everything is going well, and is correctly installed. You can run them with:

```
$ make tests
```

The test suite will run all the available tests for every supported Python environment. You can check the current build status on Travis.

6.1.1 Installation on *nix systems

Standard installation

First, make sure you have CouchDB installed on your system.

Note: If you're running OSX, you can use Homebrew to install Daybed required dependencies:

```
$ brew install python couchdb
```

Make sure you follow any extra setup instruction provided by Homebrew for these packages.

It's highly recommended to create and use a Python virtualenv local to the project:

```
$ virtualenv `pwd`/.venv
$ source .venv/bin/activate
```

Now, install Daybed's Python dependencies in this venv:

```
$ make install
```

Don't forget to start your CouchDB and ElasticSearch server instances:

```
$ sudo service couchdb start
$ sudo service elasticsearch start
```

Then start the Daybed server:

```
$ make serve
```

Development installation

If you start hacking on Daybed, a good practice is to ensure the tests keep passing for all supported Python environments:

```
$ make tests
```

Note: OSX users can install all supported Python platforms using this Homebrew command:

```
$ brew install python python3 pypy couchdb
```

Once you're all set, keep on reading for using daybed.

6.1.2 Using Docker images

Docker allows you to easily run a Daybed instance, locally or in production.

Two steps setup

Note: You shall use Docker links, available in version 0.11+.

Run a CouchDB instance:

```
$ sudo docker run --name couchdb klaemo/couchdb
```

Run an ElasticSearch instance:

```
$ sudo docker run --name couchdb dockerfile/elasticsearch
```

Run a *Daybed* container linked to the previous ones:

```
sudo docker run --link=couchdb:couchdb \
                --link=elasticsearch:elasticsearch \
                --publish=8000:8000 makinacorpus/daybed
```

Test it!

```
$ curl http://localhost:8000/v1/
```

Runtime parameters

A number of environment variables can be set at runtime, to control the backend connection for instance:

```
$ sudo docker run ... --env BACKEND_DB_NAME=mydb ...
```

See the `Dockerfile` file for a complete list of variables, and their default value.

Custom configuration

In order to run the container with a custom configuration file. Just create a file `production.ini` in a custom folder (e.g. `/myconf`), and mount it this way:

```
$ sudo docker run ... --volume=/myconf:/opt/apps/daybed/conf ...
```

Build the image from sources

From the repository folder:

```
$ make clean
$ sudo docker build -t daybed .
```

6.2 How to use the Daybed API

Daybed is a REST interface you can use to create model definitions, edit them and publish data that complies to these models.

Let's say you want to have a Daybed-managed *todo list*. Follow the steps and you will have a grasp of how Daybed works.

To simplify API calls, you can use httpie which performs HTTP requests easily.

All examples in this section are using *httpie* against a local Daybed server running on port 8000.

Daybed uses Hawk, so to run the following example, you'll need to install the requests-hawk module.

6.2.1 Listing supported fields

Daybed supports several field types to build your model definition. All details are given in the *dedicated documentation* section.

You can also get a list of these fields and their parameters programmatically, on the */fields* endpoint:

```
http GET http://localhost:8000/v1/fields --verbose --json
```

6.2.2 Authentication

You need to be authenticated to be able to run most of the commands. In order to get authenticated, the first thing to do is to get some *credentials* from Daybed.

In order to get yours, you need to send a `POST` request:

```
http POST http://localhost:8000/v1/tokens

HTTP/1.1 201 Created
Content-Length: 273
Content-Type: application/json; charset=UTF-8
Date: Thu, 24 Jul 2014 16:25:49 GMT
Server: waitress

{
    "credentials": {
        "algorithm": "sha256",
        "id": "e0394574578356252e2033b829b90291e2ff1f33ccbcbcec777485f3a5a10bca",
        "key": "416aa1287121218feeb9b751a9614959a1c95fd09aba5959bab7c484dcd1b198"
    },
    "token": "ad37fc395b7ba83eb496849f6db022fbb316fa11081491b5f00dfae5b0b1cd22"
}
```

In your next requests, you can either :

- use the `token`, and rely on a Hawk library to wrap everything (*recommended*);
- use the `credentials` pair of *id* and *key*, and build the Hawk `Authorization` header yourself (*or probably via ‘‘hawk.js‘‘*).

If you want to get always the same token for a given user, you can access the endpoint using Basic Auth Authorization scheme:

```
http POST localhost:8000/v1/tokens --auth admin:password -v

POST /v1/tokens HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate, compress
Authorization: Basic YWRtaW46cGFzc3dvcmQ=
Content-Length: 0
Host: localhost:8000
User-Agent: HTTPie/0.8.0


HTTP/1.1 201 Created
Content-Length: 266
Content-Type: application/json; charset=UTF-8
Date: Fri, 07 Nov 2014 15:09:01 GMT
Server: waitress

{
    "credentials": {
        "algorithm": "sha256",
        "id": "371ef18f8a054e5c9fb0961cc5b81006080e9ad22078d30b3727c7c32843579f",
        "key": "87e72a8e5dcaf8b4be00b8729462814da3d438ce2fd8b6efee335db722d8369d"
    },
    "token": "9f19de0237c9bd59f803de1785f7aea4e3499b6929df3428e1b415fed81f797a"
}
```

In that case you will get a 201 Created on the creation time and then a 200 each time you ask for the same token.

6.2.3 Model management

PUT and POST /v1/models

You can create models without being authenticated, since model creation is allowed to everyone by default.

When you are authenticated, all the objects you create will be associated to your credentials *id*.

First, you put a definition under the name "todo" using a PUT request on **/models**:

```
http PUT http://localhost:8000/v1/models/todo
```

Use the `token` as the `auth` value, as expected by the `requests-hawk` library.

```
echo '{"definition":
  {
    "title": "todo",
    "description": "A list of my stuff to do",
    "fields": [
        {
            "name": "item",
            "type": "string",
            "label": "The item"
        },
        {
            "name": "status",
            "type": "enum",
            "choices": [
                "done",
                "todo"
            ],
            "label": "is it done or not"
        }
     ]
  }
}' > definition

http PUT http://localhost:8000/v1/models/todo @definition \
     --verbose \
     --auth-type=hawk \
     --auth='ad37fc395b7ba83eb496849f6db022fbb316fa11081491b5f00dfae5b0b1cd22:'
```

And you receive the model id back

```
HTTP/1.1 200 OK
Content-Length: 14
Content-Type: application/json; charset=UTF-8
Date: Thu, 24 Jul 2014 18:35:10 GMT
Server: waitress

{
    "id": "todo"
}
```

Since the token was used, the new model was associated to your *id*, and you are the only one to get read *and* write permissions. Of course, the model permissions can be changed later.

Note: In case you don't want to define a name yourself for your model, you can do the exact same request, replacing the **PUT** http method by a **POST**. A random name will be generated.

The definition properties are:

- **title**: The model title
- **description**: The model description

- **fields**: The model fields' list. See *fields documentation*
- **extra**: An optional property to store custom data to your model.

GET /models

Returns the list of models where you have the permission to read the definition:

```
http GET http://localhost:8000/v1/models --verbose \
    --auth-type=hawk \
    --auth='ad37fc395b7ba83eb496849f6db022fbb316fa11081491b5f00dfae5b0b1cd22:'

GET /v1/models HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Authorization: Hawk mac="3NXv...=", hash="B0we...=", id="36...0", ts="1407166852", nonce="tQlJHv"
Host: localhost:8000
User-Agent: HTTPie/0.8.0


HTTP/1.1 200 OK
Content-Length: 202
Content-Type: application/json; charset=UTF-8
Date: Mon, 04 Aug 2014 15:40:52 GMT
Server: waitress

{
    "models": [
        {
            "description": "A list of my stuff to do.",
            "id": "todo",
            "title": "Todo"
        }
    ]
}
```

GET /v1/models/{modelname}

You can now get your models back:

```
http GET http://localhost:8000/v1/models/todo \
  --verbose \
  --auth-type=hawk \
  --auth='ad37fc395b7ba83eb496849f6db022fbb316fa11081491b5f00dfae5b0b1cd22:'

GET /v1/models/todo HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Authorization: Hawk mac="CEhSQuh8tqGY8RbdrnMvGyIRJBDmdxJeu2/HIRB0pbQ=", hash="B0weSUXsMcb5UhL41FZbrU
74578356252e2033b829b90291e2ff1f33ccbcbcec777485f3a5a10bca", ts="1406228025", nonce="4sEpMQ"
Host: localhost:8000
User-Agent: HTTPie/0.8.0



HTTP/1.1 200 OK
Content-Length: 1330
Content-Type: application/json; charset=UTF-8
Date: Thu, 24 Jul 2014 18:53:45 GMT
Server: waitress
```

```
{
    "permissions": {
        "e0394574578356252e2033b829b90291e2ff1f33ccbcbcec777485f3a5a10bca": [
            'create_record',
            'delete_all_records',
            'delete_model',
            'delete_own_records',
            'read_permissions',
            'read_all_records',
            'read_definition',
            'read_own_records',
            'update_permissions',
            'update_all_records',
            'update_definition',
            'update_own_records',
        ]
    },
    "definition": [
        {
            "description": "A list of my stuff to do",
            "fields": [
                {
                    "label": "The item",
                    "name": "item",
                    "type": "string"
                },
                {
                    "choices": [
                        "done",
                        "todo"
                    ],
                    "label": "is it done or not",
                    "name": "status",
                    "type": "enum"
                }
            ],
            "title": "todo"
        }
    ],
    "records": []
}
```

Note: You will get a `401 - Unauthorized` response if you don't have the permission to read the model definition.

6.2.4 Pushing records

POST /v1/models/{modelname}/records

PUT /v1/models/{modelname}/records/{id}

Now that you've defined the schema, you may want to push some real record there:

```
http POST http://localhost:8000/v1/models/todo/records item="work on daybed" status="done" \
    --verbose \
    --auth-type=hawk \
    --auth='ad37fc395b7ba83eb496849f6db022fbb316fa11081491b5f00dfae5b0b1cd22:'
```

```
POST /v1/models/todo/records HTTP/1.1
Accept: application/json
Accept-Encoding: gzip, deflate
Authorization: Hawk mac="4Sly1HVkkKsRk43dHOLw/e/AmWeoDEe9ZbVu9cugzg0=", hash="KE3ivKqZxHPTg1yzUAJHOu/
74578356252e2033b829b90291e2ff1f33ccbcbcec777485f3a5a10bca", ts="1406228375", nonce="T2NP4V"
Content-Length: 44
Content-Type: application/json; charset=utf-8
Host: localhost:8000
User-Agent: HTTPie/0.8.0

{
    "item": "work on daybed",
    "status": "done"
}

HTTP/1.1 201 Created
Content-Length: 42
Content-Type: application/json; charset=UTF-8
Date: Thu, 24 Jul 2014 18:59:35 GMT
Location: http://localhost:8000/v1/models/todo/records/ebc9f07c8faa4969a76f46b8c514fac6
Server: waitress

{
    "id": "ebc9f07c8faa4969a76f46b8c514fac6"
}
```

The server sends us back the **id** of the newly created record.

Note: You can also only validate the data your are sending, by setting the `Validate-Only` header, which will prevent storing it as a record.

GET /v1/models/{modelname}/records

Using the GET method, you can get back all the records you have created:

```
http GET http://localhost:8000/v1/models/todo/records \
    --json \
    --verbose \
    --auth-type=hawk \
    --auth='ad37fc395b7ba83eb496849f6db022fbb316fa11081491b5f00dfae5b0b1cd22:'

GET /v1/models/todo/records HTTP/1.1
Accept: application/json
Accept-Encoding: gzip, deflate
Authorization: Hawk mac="OQ9PYGfLhE7L0TPHFpYteHI0j3PBnKgEjyYjMQXMsaM=", hash="NVuBm+XMyya3Tq4EhpZ0cQW
Content-Type: application/json; charset=utf-8
Host: localhost:8000
User-Agent: HTTPie/0.8.0


HTTP/1.1 200 OK
Content-Length: 151
Content-Type: application/json; charset=UTF-8
Date: Thu, 24 Jul 2014 20:08:04 GMT
Server: waitress

{
    "records": [
```

```
        {
            "item": "work on daybed",
            "status": "done"
        },
    ]
}
```

6.2.5 Get back a definition

GET /v1/models/{modelname}/definition

```
http GET http://localhost:8000/v1/models/todo/definition \
    --verbose \
    --auth-type=hawk \
    --auth='504fd8148d7cdca10baa3c5208b63dc9e13cad1387222550950810a7bdd72d2c:'

GET /v1/models/todo/definition HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Authorization: Hawk mac="k9edIqpoz7cSUJQTroXgM4vgDoZb2Z2KO2u40QCbtYk=", hash="B0weSUXsMcb5UhL41FZbrU
Host: localhost:8000
User-Agent: HTTPie/0.8.0


HTTP/1.1 200 OK
Content-Length: 224
Content-Type: application/json; charset=UTF-8
Date: Tue, 29 Jul 2014 14:50:26 GMT
Server: waitress

{
    "description": "A list of my stuff to do",
    "fields": [
        {
            "label": "The item",
            "name": "item",
            "type": "string"
        },
        {
            "choices": [
                "done",
                "todo"
            ],
            "label": "is it done or not",
            "name": "status",
            "type": "enum"
        }
    ],
    "title": "todo"
}
```

6.2.6 Get back the model permissions

GET /v1/models/{modelname}/permissions

```
http GET http://localhost:8000/v1/models/todo/permissions \
    --verbose \
    --auth-type=hawk \
    --auth='504fd8148d7cdca10baa3c5208b63dc9e13cad1387222550950810a7bdd72d2c:'

GET /v1/models/todo/permissions HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Authorization: Hawk mac="G8PntYqGA0DiP4EC0qvvr70tmCZrsVBdTTTBq9ZeKYg=", hash="B0weSUXsMcb5UhL41FZbrU
Host: localhost:8000
User-Agent: HTTPie/0.8.0


HTTP/1.1 200 OK
Content-Length: 293
Content-Type: application/json; charset=UTF-8
Date: Tue, 29 Jul 2014 14:51:20 GMT
Server: waitress

{
    "220a1c4212d8f005f0f56191c5a91f8fe266282d38b042e6b35cad8034f22871": [
        "create_record",
        "delete_all_records",
        "delete_model",
        "delete_own_records",
        "read_all_records",
        "read_definition",
        "read_own_records",
        "read_permissions",
        "update_all_records",
        "update_definition",
        "update_own_records"
        "update_permissions",
    ]
}
```

6.2.7 Change model permissions

As described in *the dedicated section about permissions*, you can add or remove permissions from models.

For example, you may want to give the permission to read everyone's records to anonymous users (i.e. *Everyone*).

Using a `PATCH` request, existing permissions configuration is not overwritten completely :

PATCH /v1/models/{modelname}/permissions

```
echo '{"Everyone": ["+read_all_records"]}' | http PATCH http://localhost:8000/v1/models/todo/permiss
    --json \
    --verbose \
    --auth-type=hawk \
    --auth='504fd8148d7cdca10baa3c5208b63dc9e13cad1387222550950810a7bdd72d2c:'

 PATCH /v1/models/todo/permissions HTTP/1.1
 Accept: application/json
 Accept-Encoding: gzip, deflate
 Authorization: Hawk mac="CWT9du2YxOoTb2i5d15bBTA4XiSYY/99ybh6g7welLM=", hash="Nt8m2h1nc5lVUItOobOli
 Content-Length: 34
 Content-Type: application/json; charset=utf-8
```

```
Host: localhost:8000
User-Agent: HTTPie/0.8.0

{
    "Everyone": [
        "+read_all_records"
    ]
}

HTTP/1.1 200 OK
Content-Length: 333
Content-Type: application/json; charset=UTF-8
Date: Tue, 29 Jul 2014 14:59:00 GMT
Server: waitress

{
    "220a1c4212d8f005f0f56191c5a91f8fe266282d38b042e6b35cad8034f22871": [
        "create_record",
        "delete_all_records",
        "delete_model",
        "delete_own_records",
        "read_all_records",
        "read_definition",
        "read_own_records",
        "read_permissions",
        "update_all_records",
        "update_definition",
        "update_own_records"
        "update_permissions",
    ],
    "system.Everyone": [
        "read_all_records"
    ]
}
```

If you add an unknown permission or modify the permissions of an unknown *id*, you will get an error.

6.2.8 Reset permissions

Using a `PUT` request, existing permissions will be completely erased and replaced by the new ones.

Using the `ALL` shortcut, you can grant all available permissions.

PUT /v1/models/{modelname}/permissions

```
echo '{"Everyone": ["read_definition"], "Authenticated": ["ALL"]}' | http PUT http://localhost:8000/v
    --json \
    --verbose \
    --auth-type=hawk \
    --auth='504fd8148d7cdca10baa3c5208b63dc9e13cad1387222550950810a7bdd72d2c:'

PATCH /v1/models/todo/permissions HTTP/1.1
Accept: application/json
Accept-Encoding: gzip, deflate
Authorization: Hawk mac="CWT9du2YxOoTb2i5d15bBTA4XiSYY/99ybh6g7welLM=", hash="Nt8m2h1nc5lVUItOobOliV
Content-Length: 34
Content-Type: application/json; charset=utf-8
Host: localhost:8000
```

```
User-Agent: HTTPie/0.8.0

{
    "Everyone": [
        "read_definition"
    ],
    "Authenticated": [
        "ALL"
    ]
}

HTTP/1.1 200 OK
Content-Length: 333
Content-Type: application/json; charset=UTF-8
Date: Tue, 29 Jul 2014 14:59:00 GMT
Server: waitress

{
    "system.Authenticated": [
        "create_record",
        "delete_all_records",
        "delete_model",
        "delete_own_records",
        "read_all_records",
        "read_definition",
        "read_own_records",
        "read_permissions",
        "update_all_records",
        "update_definition",
        "update_own_records"
        "update_permissions",
    ],
    "system.Everyone": [
        "read_definition"
    ]
}
```

Note: It can be useful if you need to remove permissions associated to an unknown *id* for example.

6.3 Daybed field types

A field is defined with these global properties:

- **label**: human readable field label
- **name**: storage attribute name
- **type**: kind of value to be stored
- **required**: mandatory attribute (*optional, default: true*)
- **hint**: help text (*optional*)

You then have different types of fields, all detailed in this section.

6.3.1 Basic types

Basic types don't have specific properties.

- **int**: An integer
- **string**: A set of characters
- **text**: A text
- **email**: A valid email
- **url**: A valid URL
- **decimal**: A decimal number
- **boolean**: A boolean

For example, when combining global properties and one of those basic field, it becomes:

```
{
    "label": "Task item",
    "name": "task",
    "type": "string"
}
```

You just have to choose a type among those available.

6.3.2 Advanced types

- **enum: A choice among values**

  Specific parameters:

  - *choices*: An array of strings

```
{
    "label": "Task status",
    "name": "status",
    "type": "enum",
    "choices": [
        "done",
        "todo"
    ]
}
```

- **choices: Multiple choices among values**

  Specific parameters:

  - *choices*: An array of strings

```
{
    "label": "Hobbies",
    "name": "hobbies",
    "type": "choices",
    "choices": [
        "Litterature",
        "Cinema",
        "Mountain Bike",
        "Motor Bike",
        "Sailing"
```

```
    ]
}
```

- **range: A number within limits**

  Specific parameters:

  - *min*: An integer which is the minimum value of the field
  - *max*: An integer which is the maximum value of the field

It will accept a value that is greater than or equal to min and less than or equal to max.

```
{
    "label": "Mountain bike Wheel Size (in mm)",
    "name": "wheel-size",
    "type": "range",
    "min": 239,
    "max": 622
}
```

- **regex: A string matching a pattern**

  Specific parameters:

  - *regexp*: The pattern the value should match to be valid.

```
{
    "label": "French Mobile Phone Number",
    "name": "phone-number",
    "type": "regex",
    "regex": "^0[6-7][0-9]{8}$"
}
```

- **date: A date in *yyyy-mm-dd* format**

  Specific parameters:

  - *autonow*: Boolean, add the current date automatically if true. (default: false)

```
{
    "label": "Date of Birth",
    "name": "date",
    "type": "date",
    "autonow": true
}
```

- **datetime: A datetime in *yyyy-mm-ddTHH:MM:SS* format**

  Specific parameters:

  - *autonow*: Boolean, add the current datetime automatically if true. (default: false)

```
{
    "label": "Time of Birth",
    "name": "date_of_birth",
    "type": "datetime"
}
```

- **group: A group of fields, can define fieldsets or multi-pages forms.**

  Specific parameters:

  - *description*: A string to describe the group.

  - *fields*: A list of fields of the group.

```
{
    "label": "Fieldset",
    "type": "group",
    "fields": [
        {
            "label": "Gender",
            "name": "gender",
            "type": "enum",
            "choices": [
                "Mr",
                "Miss",
                "Ms"
            ]
        },
        {
            "label": "Firstname",
            "name": "firstname",
            "type": "string"
        },
        {
            "label": "Lastname",
            "name": "lastname",
            "type": "string"
        }
    ]
}
```

Groups are ignored during validation, and records are posted like this:

```
{"gender": "Mr", "firstname": "Remy", "lastname": "Hubscher"}
```

- **annotation: A model description field not used for validation** No specific parameters.

```
{
    "label": "Title 1",
    "type": "annotation",
}
```

The annotation type is not really a field because the record has no trace of it. It can be used to add a description between fields.

Like the `group` field type, it can help to build the form layout.

For instance:

```
{"definition":
  {
    "title": "Club members",
    "description": "Name and pictures of all members",
    "fields": [
      {
        "label": "First and last name",
        "name": "fullname",
        "type": "string"
      },
      {
        "label": "Providing a picture is optional",
        "type": "annotation",
```

```
        "css": "font-weigth: bold"
      },
      {
        "label": "Picture",
        "name": "picture",
        "type": "url",
        "required": false
      }
    ]
  }
}
```

The *css* property is just an example of how we could handle the styling of the annotation here, but it could be anything else.

- **json: A JSON value** No specific parameters.

  Beyond formatting, the content is not validated

```
{
    "label": "JSON object",
    "name": "movie",
    "type": "json"
}
```

Then you can use it like so:

```
{
  "movie": {
    "title": "The Island",
    "director": "Michael Bay",
    "actors": ["Scarlett Johnsson", "Erwan McGregor"],
    "year": 2005
  }
}
```

It will also work with a string :

```
{
  "movie": "{\"title\": \"The Island\"}"
}
```

6.3.3 Nested types

- **object: An object inside another model**

  Specific parameters, used to validate the content. Only one of them should be specified.

  - *fields*: A list of fields like in a model definition.
  - *model*: The id of an existing model.

Unlike the `json` type, the content will be validated, using either the list of fields or the definition of the specified model

```
{
  "label": "Movie",
  "name": "movie",
  "type": "object",
  "fields": [
```

```
    {
      "label": "Title",
      "name": "title",
      "type": "string"
    },
    {
      "label": "Director",
      "name": "director",
      "type": "string"
    },
    {
      "label": "Actors",
      "name": "actors",
      "type": "list",
      "item": {"type": "string"}
    }
  ]
}
```

For example, this record will be valid for the definition above:

```
{
  "movie": {
    "title": "Donnie Darko",
    "director": "Richard Kelly",
    "actors": ["Jake Gyllenhaal", "Patrick Swayze"],
  }
}
```

But this one will not:

```
{
  "movie": {
    "title": "Director and actors missing",
  }
}
```

- **list: A list of values inside another model**

  Specific parameters:

  - *item*: Defines the type of the list items. Specified like a field in a model definition.

Can be used to define a simple list of basic types (*integer, string, ...*):

```
{
  "label": "Movie titles",
  "name": "movies",
  "type": "list",
  "item": {
    "type": "string"
  }
}
```

Or a list of advanced field types (*dates, objects, ...*):

```
{
  "label": "Movie list",
  "name": "movies",
  "type": "list",
  "item": {
```

```
    "type": "object",
    "hint": "Description of a movie",
    "fields": [
      {
        "label": "Title",
        "name": "title",
        "type": "string"
      },
      {
        "label": "Director",
        "name": "director",
        "type": "string"
      }
    ]
  }
}
```

If `item` is not specified, the list items can be anything (e.g. no validation will be done on them):

```
{
  "label": "Last thoughts",
  "name": "toughts",
  "type": "list"
}
```

The following records will be considered valid with the definition above:

```
{ "toughts": [1, 2, 3] }

{ "toughts": [{"miam": true}, 42, ["OSM", "Mapnik"], "World Company"] }
```

6.3.4 Relation types

- **anyof: Any number of choices among records of a given model**

  Specific parameters:

  - *model*: The model id from which records can be selected

```
{
  "name": "actors",
  "type": "anyof",
  "model": "generic:people:moviestars",
  "label": "Movie actors"
}
```

- **oneof: One choice among records of a given model**

  Specific parameters:

  - *model*: The model id from which the record can be selected

```
{
  "name": "maincharacter",
  "type": "oneof",
  "model": "generic:people:moviestars",
  "label": "Main character"
}
```

6.3.5 Geometric types

- **geojson: A GeoJSON geometry (not a FeatureCollection)** No specific parameters.

```
{
  "label": "where is it?",
  "name": "place",
  "type": "geojson"
}
```

Then you can use it like so:

```
http POST http://localhost:8000/v1/models/todo/records \
item="work on daybed" status="done" \
place='{"type": "Point", "coordinates": [0.4, 45.0]}' \
--verbose --auth-type=hawk \
--auth='ad37fc395b7ba83eb496849f6db022fbb316fa11081491b5f00dfae5b0b1cd22:'

{
  "item": "work on daybed",
  "place": {
    "coordinates": [
      0.4,
      45.0
    ],
    "type": "Point"
  },
  "status": "done"
}
```

- **point: A point**

  Specific parameters:

  - *gps*: A boolean that tells if the point coordinates are GPS coordinates and it will validate that coordinates are between `-180,-90` and `+180,+90` (Default: *true*)

```
{
  "label": "where is it?",
  "name": "place",
  "type": "point"
}
```

Then you can use it like so:

```
http POST http://localhost:8000/v1/models/todo/records \
item="work on daybed" status="done" \
place="[0.4, 45.0]" \
--verbose --auth-type=hawk \
--auth='ad37fc395b7ba83eb496849f6db022fbb316fa11081491b5f00dfae5b0b1cd22:'

{
  "item": "work on daybed",
  "place": [
    0.4,
    45.0
  ],
  "status": "done"
}
```

- **line: A line made of points**

Specific parameters

- *gps*: A boolean that tells if the point coordinates are GPS coordinates and it will validate that coordinates are between `-180,-90` and `+180,+90` (Default: *true*)

```
{
  "label": "where is it?",
  "name": "place",
  "type": "line"
}
```

Then you can use it like so:

```
http POST http://localhost:8000/v1/models/todo/records \
item="work on daybed" status="done" \
place="[[0.4, 45.0], [0.6, 65.0]]" \
--verbose --auth-type=hawk \
--auth='ad37fc395b7ba83eb496849f6db022fbb316fa11081491b5f00dfae5b0b1cd22:'

{
  "item": "work on daybed",
  "place": [
    [
      0.4,
      45.0
    ],
    [
      0.6,
      65.0
    ]
  ],
  "status": "done"
}
```

- **polygon: A polygon made of a closed line**

  Specific parameters

  - *gps*: A boolean that tells if the point coordinates are GPS coordinates and it will validate that coordinates are between `-180,-90` and `+180,+90` (Default: *true*)

```
{
  "label": "where is it?",
  "name": "place",
  "type": "polygon"
}
```

Then you can use it like so:

```
http POST http://localhost:8000/v1/models/todo/records \
item="work on daybed" status="done" \
place="[[[0.4, 45.0], [0.6, 65.0], [0.8, 85.0], [0.4, 45.0]]]" \
--verbose --auth-type=hawk \
--auth='ad37fc395b7ba83eb496849f6db022fbb316fa11081491b5f00dfae5b0b1cd22:'

{
  "item": "work on daybed",
  "place": [
    [
      [
        0.4,
```

```
        45.0
      ],
      [
        0.6,
        65.0
      ],
      [
        0.8,
        85.0
      ],
      [
        0.4,
        45.0
      ]
    ]
  ],
  "status": "done"
}
```


6.4 Permissions

In *Daybed*, permissions will let you define access rules on models, records and even permissions.

They allow to express rules like:

- "Everyone can create new records on this model"
- "Alexis is able to delete records created by others"
- "Authenticated users can modify their own records"
- "Everyone can read the model definition"

This section describes how they work and how to use them.

6.4.1 Models permissions

Here's a list of permissions you can define on a model:

- **read_definition**: read the model definition
- **read_permissions**: read the model permissions (who can do what)
- **update_definition**: update the model definition
- **update_permissions**: change the permissions for the model
- **delete_model**: delete the model
- **create_record**: add an entry to the model
- **read_all_records**: read all model's records
- **update_all_records**: update all model's records
- **delete_all_records**: delete any model's records
- **read_own_records**: read records on which you are an author
- **update_own_records**: update and change records on which you are an author

- **delete_own_records**: delete records on which you are an author

6.4.2 Views permissions

At the API level, you will often need more than one permission to get access to an API resource.

For example, if you want to get a complete model (definition and records), you will need the following permissions:

- **read_definition** and **read_permissions**
- **read_all_records** or **read_own_records**

6.4.3 Global permissions

There are three extra permissions that are configured at the server level:

- **create_model**: List of identifiers allowed to create a model
- **create_token**: List of identifiers allowed to create tokens
- **manage_tokens**: List of identifiers allowed to delete tokens

Those are configured via *a configuration file*, with the following options:

- **create_model**: `daybed.can_create_model` (default: `Everyone`)
- **create_token**: `daybed.can_create_token` (default: `Everyone`)
- **manage_tokens**: `daybed.can_manage_token` (default: None)

Example:

```
[app:main]
daybed.can_create_model = Authenticated
daybed.can_create_token = Everyone
```

6.4.4 Usage

Permissions are set on models, as a *dictionnary* between *identifiers* (*key id*) and lists of *permissions*.

In order to refer to anyone in the world, use the special *id* `Everyone` and to authenticated users with `Authenticated`.

Note: As explained in the *API usage section*, the *key ids* look like *tokens*, but they are different : the *id* is the public part of your *credentials*. The session token is private, since it contains the secret key.

When you create a model, you gain the full set of available permissions.

This means that the identifier you used in the request will be associated to all permissions:

```
http GET http://localhost:8000/v1/models/todo/permissions --verbose \
--auth-type=hawk \
--auth='ad37fc395b7ba83eb496849f6db022fbb316fa11081491b5f00dfae5b0b1cd22:'

{
    "e0394574578356252e2033b829b90291e2ff1f33ccbcbcec777485f3a5a10bca": [
        "create_record",
        "delete_all_records",
        "delete_model",
```

```
        "delete_own_records",
        "read_all_records",
        "read_definition",
        "read_own_records",
        "read_permissions",
        "update_all_records",
        "update_definition",
        "update_own_records",
        "update_permissions"
    ]
}
```

Let's say you want to allow authenticated users to create records and manage their own records on this model.

Permissions become:

```
{
    "Authenticated": [
        "create_record",
        "read_own_records",
        "update_own_records",
        "delete_own_records"
    ],
    "e0394574578356252e2033b829b90291e2ff1f33ccbcbcec777485f3a5a10bca": [
        "create_record",
        "delete_all_records",
        "delete_model",
        "delete_own_records",
        "read_all_records",
        "read_definition",
        "read_own_records",
        "read_permissions",
        "update_all_records",
        "update_definition",
        "update_own_records",
        "update_permissions"
    ]
}
```

Modification

You can use − and + to modify the existing set of permissions for an identifier.

To grant `create_record` to anonymous users, `read_permissions` to authenticated users and remove `update_permissions` from *id* `220a1c..871` you would have to send the following request:

```
{
    "Everyone": ["+create_record"],
    "Authenticated": ["+read_permissions"],
    "220a1c..871": ["-update_permissions"]
}
```

In order to add/remove all permissions to/from somebody, use the `ALL` shortcut:

```
{
    "Authenticated": ["-ALL"],
    "220a1c..871": ["+ALL"]
}
```

Note: + is implicit, the permission is added if not specified (i.e. `ALL` is equivalent to `+ALL`).

6.4.5 Concrete examples

Collaborative editor (*pad*)

Everybody can read, create, modify and delete everyone's records. However only the owner (*id* `220a1c..871`) can modify the model definition and adjust permissions.

```
{
    "Everyone": [
        "read_definition",
        "create_record",
        "read_all_records",
        "update_all_records",
        "delete_all_records"
    ],
    "220a1c..871": [
        "ALL"
    ]
}
```

If the *administrator* wants to share her privileges with others, she can either:

- share her *token*
- create a new token, assign permissions to its *key id*, and share the new token

```
{
    "Everyone": [
        "read_definition",
        "create_record",
        "read_all_records",
        "update_all_records",
        "delete_all_records"
    ],
    "6780dd..df1": [
        "update_definition",
        "read_permissions",
        "update_permissions"
    ],
    "220a1c..871": [
        "ALL"
    ]
}
```

Online poll

Everybody can answer the poll, but are not allowed to correct their answers, nor to see the poll results.

`read_definition` is given to everyone, as it might be used to build the form on the client-side:

```
{
    "Everyone": [
        "read_definition",
        "create_record"
```

```
    ],
    "220a1c..871": [
        "ALL"
    ]
}
```

TODO-list application

The development team, who created the model, has the full set of permissions.

Everybody can manage their own records, but they are private.

```
{
    "Everyone": [
        "read_definition",
        "create_record",
        "read_own_records",
        "update_own_records",
        "delete_own_records"
    ],
    "220a1c..871": [
        "ALL"
    ]
}
```

Note: Using *Everyone* instead of *Authenticated* will allow anonymous to manage a set of records that are shared among all anonymous users.

Note: Users can share their todo list if they share their *token*. But they cannot share it as read-only.

In order to accomplish this, instead of having a unique model with everyone's records, each user will have to create their own model, on which they will gain the control of permissions.

6.5 Terminology

Daybed concepts are very similar to those of any other storage or model validation software.

Credentials Credentials are a way to authenticate yourself, and are composed of two parts:

1. an **id** – *identifier* that you can publicly share
2. a **key** – similar to a password (you may prefer to not share it)

Definition A schema defined as a list of fields, each one with a name, a type and potential parameters.

Field A model definition is composed of multiple fields. Each one contains a name and a type.

Field type A type, among those available, whose purpose is to validate values (e.g. `int`, `date`, ...). It may have mandatory or optional parameters, when used in a definition (e.g. `choices`, `regex`, ...).

Identifier, Identifiers A unique *id*, part of the *credentials*, that will be associated to the models and records you created.

Identifiers are used to define *permissions*.

Model, Models A model is made of a *definition*, a set of *records*, and a list of *permissions*.

Permission, Permissions An operation name used to allow or deny requests on *models*, *records* or *tokens*. Permissions are given to *identifiers* as an associative array on models.

For example, when trying to delete a record, if the request's *identifier* doesn't have `delete_records` among its permission on this model, the permission will be denied.

See *permissions section*.

Record, Records An item to be stored in a *model*. It should provide a value for each required *field* of the *definition*.

Token, Tokens, Hawk-Session-Token An unique string from each pair of *id* and *key*, and helps you keep, handle or share your credentials as a simple string.

d