Quetzal Documentation

Release 0.6.0-dev

David Ojeda

Mar 24, 2020

GENERAL

Quetzal (short for Quetzalcóatl, the feathered snake), a RESTful API designed to store data files and manage their associated metadata.

Quetzal is an application that uses Cloud storage providers and non-structured databases to help researchers organize their data and metadata files. Its main feature is to provide a remote, virtually infinite, storage location for researchers' data, while providing an API to encapsulate data/metadata operations. In other words, researchers and teams can work with large amounts of data that would be too large for local analyses, using Quetzal to simplify the complexity of Cloud resource management.

Quetzal's mid-term roadmap is to integrate with large public physiological signal databases like PhysioNet, MIPDB, TUH, among others. Tha main objective is to provide researchers and data scientists a unique bank of file datasets with a unified API to access the data and to encapsulate the heteronegeity of these datasets.

CHAPTER

ONE

FEATURES

There are two scenarios where Quetzal was designed to help:

- Imagine you want to apply a data processing pipeline to a large dataset. There are several solutions on how to execute and parallelize your code, but *where is the data?* Moreover, imagine that you want to do a transverse study: How do you manage the different sources? How to download them?

  Quetzal provides a single data source with a simple API that will let you define easily the scope of your study and, with a brief Python code that uses Quetzal client, you will be able to download your dataset.

- Let's say that you are preparing a new study implying some data collection protocol. You could define a procedure where the data operators or technicians take care to copy the data files in a disk, Google Drive or Dropbox, along with the notes associated with each session, like subject study identifier, date, age, temperature, etc. Doing this manually would be error-prone. Moreover, the structure of these notes (i.e. the metadata) may evolve quickly, so you either save them as manual notes, text files, or some database that gives you the flexibility to quickly adapt its structre.

  Using the Quetzal API, you automate the upload and safe storage of the study files, associate the metadata of these files while having the liberty to set and modify the metadata structure as you see fit.

In brief, Quetzal offers the following main features:

- **Storage** of data files, based on cloud storage providers, which benefits from all of the features from the provider, such as virtually infinite storage size.
- **Unstructured metadata** associated to each file*. Quetzal does not force the user to organize your metadata in a particular way, it lets the user keep whatever structure they prefer.
- **Structured metadata views** for metadata exploration or dataset definition. By leveraging Postgres SQL, unstructured metadata can be queried as JSON objects, letting the user express what subset of the data they want to use.
- **Metadata versioning**. Changes on metadata are versioned, which is particularly useful to ensure that a dataset are reproducible.
- Endpoints and operations defined using the OpenAPI v3 specification.

The rest of this documentation is divided in three main sections, a *General* explanation of Quetzal concepts, design decisions and how it works. For Quetzal users, that is, those who want to consume the API to explore or download from the public datasets, the *User documentation* section shows the most common use cases and examples. For developers or users that want to have their own Quetzal server, the *Developer documentation* includes all the details on creating a development environment, and procedures on how to deploy a server.

1.1 Introduction

1.1.1 Motivation

1.1.2 Main features

1.1.3 Alternatives

1.2 Design

This document explains the concepts and design decisions followed in Quetzal.

1.2.1 Concepts

File

In Quetzal, **the basic unit of data is the file**. A file (sometimes referred as *data file*) is not a Quetzal-specific term; it is exactly what a regular file is: a collection of bytes stored at a particular location. The specific location where a file is stored is handled by Quetzal and it depends on what storage backend was configured for the application.

When Quetzal stores a file, which is uploaded by a user through the file upload endpoint, it saves it in a storage bucket with a unique URL, while keeping track of a minimal set of metadata such as its URL, size, checksum, original path and filename (see the *Base family* for more details).

Metadata

Metadata are **key-value pairs associated to each file uploaded to Quetzal**. They are is intended to represent all that extra information not directly represented in the *contents* of file. This includes all that useful information that a researcher may need to provide context or annotations to the data. For example, the subject identifier, recording hardware used, software version number of the acquisition software, etc. Additionally, any information that may be useful to query and filter the data is also a good candidate to be saved as metadata. For example, sampling frequency, whether the file contains an error or not, and even references to other files.

To illustrate what metadata is, let us imagine that you have a dataset of three files, with the following associated information:

filename	subject	session	date	type
`study_foo/subject_1/session_1/eeg/signals.xdf`	S001	1	02/03/2019	EEG
`study_foo/subject_1/session_2/eeg/signals.xdf`	S001	2	03/03/2019	EEG
`study_foo/subject_2/ecg/cardiac.edf`	S002		23/01/2019	Holter

In this case, the first file has four metadata entries: the subject identifier, its session number, the date and a categorical value indicating that the data file contains EEG signals. We can even say that the filename is a metadata as well, because it has a key (*filename*) and a value (*study/subject_x/...*).

The other files have similar metadata, but note that the third file does not have a session number. This is one of the reasons Quetzal works with *unstructured* metadata; you are not obligated to follow the same metadata structure for all files.

Family

Once you start considering metadata, and how it is just a key-value pair associated to a file, you may think of a myriad of metadata to associate to your files. How do you organize these key-value pairs? *Families* provide a semantic organization of your metadata.

In Quetzal, a family is a **set of metadata keys**, defined for some common semantic or organizational purpose.

In the table presented above, it would make sense to keep the subject, session and date metadata grouped together, since this information is related to the study protocol. The data type, on the other hand, could be organized elsewhere, just for the purpose of this example, in a family that groups all signal-related information. Moreover, the filename can be related to a *base* family, which is information that Quetzal needs for bookkeeping.

Here is an updated table with the metadata and their families:

base family	*study* family			*signal* family
filename	subject	session	date	type
`study_foo/subject_1/session_1/eeg/signals.xdf`	S001	1	02/03/2019	EEG
`study_foo/subject_1/session_2/eeg/signals.xdf`	S001	2	03/03/2019	EEG
`study_foo/subject_2/ecg/cardiac.edf`	S002		23/01/2019	Holter

A row on this table, corresponding to the metadata of one file, can be represented as a JSON object as shown below. This representation is how the Quetzal API responds to a request for the metadata of a file.

```
{
  "base": {
    "filename": "study_foo/subject_1/session_1/eeg/signals.xdf"
  },
  "study": {
    "subject": "S001",
    "session": 1,
    "date": "2019-03-02"
  },
  "signal": {
    "type": "EEG"
  }
}
```

Base family

Each file has a **minimal set of metadata** needed by the Quetzal application for keeping track of files, where they are stored, etc. These metadata are defined under the *base* family. Its keys are defined in the `quetzal.app.models.BaseMetadataKeys` enumeration, which are:

- **id**: A unique identifier of the file. This is generated by Quetzal when the file is created as a UUID4 number. For example: `f5b460ad-b1e9-4e09-ac43-2c670ffeac6d`.
- **url**: A uniform resource locator that indicates where Quetzal stores this file. Usually (*but not necessarily!*), it has its **id** in it. For example: `gs://some_bucket/f5b460ad-b1e9-4e09-ac43-2c670ffeac6d`. Note that the URL does not include the filename: these are just metadata like any other and many files in Quetzal could have the exact same filename!

- **filename**: The *basename* of the file when it was uploaded. Note that it is *only* the basename, that is, there is no path in it. For example: `signals.xdf`.
- **path**: The *pathname* of the file when it was uploaded.
- **size**: Size in bytes of the file contents.
- **checksum**: MD5 digest of the files contents.
- **date**: Datetime when the file was uploaded.
- **state**: Enumeration indicating the state of the file. Used to mark temporary or deleted files.

The base family is entirely managed by Quetzal. It can only have the keys listed above. Their values are set by Quetzal when the file is uploaded. It is not possible to change them afterwards, with the sole exception of the path.

Other families

Quetzal lets the user define any number of families. Within each family, there can be any number of keys. There is only one constraint: the **id** key is reserved and managed by Quetzal.

Unstructured metadata

The contents of metadata in Quetzal are not constrained to a particular schema. They can be a string, a number, date, and even lists or nested objects. This features gives great flexibility on how and what to store as metadata.

Considering all the elements mentioned in the *Base family*, *Other families* and this section, we can expand completely define and expand the metadata of the first file in this page as:

```
{
  "base": {
    "id": "f5b460ad-b1e9-4e09-ac43-2c670ffeac6d",
    "url": "gs://some_bucket/f5b460ad-b1e9-4e09-ac43-2c670ffeac6d",
    "filename": "signals.xdf",
    "path": "study_foo/subject_1/session_1/eeg",
    "size": 19058370,
    "checksum": "9529f1439ec59ca105de75973a241574",
    "date": "2019-03-02T09:37:05.618034+00:00",
    "state": "READY"
  },
  "study": {
    "id": "f5b460ad-b1e9-4e09-ac43-2c670ffeac6d",
    "subject": "S001",
    "session": 1,
    "date": "2019-03-02"
  },
  "signal": {
    "id": "f5b460ad-b1e9-4e09-ac43-2c670ffeac6d",
    "type": "EEG",
    "sampling_rate": 512,
    "samples": 15360,
    "channels": ["Fpz", "F3", "F4", "Fz"],
    "device": {
      "name": "foo",
      "manufacturer": "bar",
      "firwmare_version": "1.0.1"
    }
```

(continues on next page)

(continued from previous page)

```
  }
}
```

Note that:

- All families have an **id** key with the same value.
- The base family has been populated with all the required keys.
- The signal family has been augmented with more complex objects types.

Family versioning

Metadata hold important information that is frequently used in many data analyses. For instance, questions like *"Is there a significant difference of X feature for each subject?"* is a question that needs to use the subject identifier, which is stored as metadata. Due to their importance, it is desirable to have some change or version control mechanism for the metadata.

Quetzal tracks the changes of metadata with family versioning. Each family has a version number. Quetzal guarantees that requests for the metadata of a particular family version are always the same. Changes of metadata values result in a new version number for its associated family.

Workspace

All data and metadata in Quetzal is stored in a freezed state. There are no changes of the file contents or its metadata, unless this happens inside a workspace.

In Quetzal, a workspace is **a configuration of exact metadata families and their version**. It is a **snapshot** of the data and metadata that permits the addition of new files and the addition or modification of metadata. It also provides a storage location for temporary files in a Cloud storage provider (typically a bucket). Finally, through a workspace, a number of API operations are available, such as uploading files, creating views, among others.

Local vs global metadata

When working on a workspace, the metadata of files requested through the workspace will contain the changes or additions that have been introduced in the workspace. On the other hand, when the metadata is requested *without* a workspace, it will be the metadata of the latest known version of each family. These two cases are referred to, respectively, as local and global metadata.

Let us illustrate with an example. Suppose that Quetzal currently has only one file, with metadata:

```
{
  "base": {
    "id": "f5b460ad-b1e9-4e09-ac43-2c670ffeac6d",
    "url": "gs://some_bucket/f5b460ad-b1e9-4e09-ac43-2c670ffeac6d",
    "filename": "signals.xdf",
    "path": "study_foo/subject_1/session_1/eeg",
    "size": 19058370,
    "checksum": "9529f1439ec59ca105de75973a241574",
    "date": "2019-03-02T09:37:05.618034+00:00",
    "state": "READY"
  },
  "study": {
    "id": "f5b460ad-b1e9-4e09-ac43-2c670ffeac6d",
```

(continues on next page)

(continued from previous page)

```
    "subject": "S001",
    "session": 1,
    "date": "2019-03-02"
  }
}
```

Now, assume that a user creates a workspace with id 1 that uses the **base** and **study** families. Immediately after its creation, both the local and global metadata are the same, because a workspace is a snapshot of the metadata.

Let us say that the user sends a metadata modification to fix an incorrect subject identification, setting `"subject"` to `"S123"` and adding an `"operator"` entry. After this operation, known in the API as Modify metadata, the local and global metadata differ:

Local metadata

```
{
  "base": {
    "id": "f5b460ad-b1e9-4e09-ac43-2c670ffeac6d",
    "url": "gs://some_bucket/f5b460ad-b1e9-4e09-ac43-2c670ffeac6d",
    "filename": "signals.xdf",
    "path": "study_foo/subject_1/session_1/eeg",
    "size": 19058370,
    "checksum": "9529f1439ec59ca105de75973a241574",
    "date": "2019-03-02T09:37:05.618034+00:00",
    "state": "READY"
  },
  "study": {
    "id": "f5b460ad-b1e9-4e09-ac43-2c670ffeac6d",
    "subject": "S123",
    "session": 1,
    "date": "2019-03-02",
    "operator": "John Doe"
  }
}
```

Global metadata

```
{
  "base": {
    "id": "f5b460ad-b1e9-4e09-ac43-2c670ffeac6d",
    "url": "gs://some_bucket/f5b460ad-b1e9-4e09-ac43-2c670ffeac6d",
    "filename": "signals.xdf",
    "path": "study_foo/subject_1/session_1/eeg",
    "size": 19058370,
    "checksum": "9529f1439ec59ca105de75973a241574",
    "date": "2019-03-02T09:37:05.618034+00:00",
    "state": "READY"
  },
  "study": {
    "id": "f5b460ad-b1e9-4e09-ac43-2c670ffeac6d",
    "subject": "S001",
    "session": 1,
    "date": "2019-03-02"
  }
}
```

Workspace views

Workspace state

Query

1.2.2 API

1.2.3 Usage workflow

1.3 License

BSD 3-Clause License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

1.4 Quickstart

1.4.1 API documentation

Link to redoc.

1.4.2 Python client

Info and link to quetzal-client and quetzal-openapi-client.

1.4.3 Other clients

Postman, etc.

1.4.4 Public databases

List of public data.

1.5 Use cases

1.5.1 Basic queries

1.5.2 Downloading files

1.5.3 Downloading metadata

1.5.4 Uploading files

1.5.5 Uploading metadata

Quetzal Client CLI

```
$ quetzal-client foo
```

Python

```
import quetzal.client
client = quetzal.client.Client()
client.foo()
```

cURL

```
$ curl -X POST https://api.quetz.al/api/v1/data/workspaces/
```

1.6 Quickstart

To get a quick development environment follow these steps:

1. Install poetry, Docker, and docker-compose (usually docker-compose is already included by Docker).
2. Clone a local copy of Quetzal:

   ```
   git clone git@github.com:quetz-al/quetzal.git
   ```

3. Create a virtual environment and install Quetzal:

```
cd quetzal
poetry install
```

4. For an environment based on docker-compose, build the Docker images:

```
docker-compose build
```

5. At this point you can get a local Quetzal server that saves files in a local filesystem. Launch it with:

```
docker-compose up
```

6. For a server that runs outside the docker-compose environment (for development or testing purposes), modify the `config.py` file according to your needs (in particular the hostnames to the database or the rabbit queue) and launch a server with:

```
FLASK_ENV=local-tests flask run --host 0.0.0.0 --port 5000
```

1.7 Cloud storage

Take the development environment to the next step by using cloud storage to store data... (description on how to do this coming soon).

1.8 Structure

1.8.1 Services structure

Application, database, queue, worker, proxy.

1.8.2 Application structure

Connexion, Flask.

1.8.3 Database structure

Models and relations.

1.9 Code organization

Where to find what.

1.10 Development use cases

Examples:

- Change behaviour of an existing API operation.
- Change the API definition.
- Change behaviour of background task.
- Cleaning and restarting the development environment.
- Change client behavior.
- Tag a new version.
- Update clients.

1.10.1 Create or modify a database model

This is done with SQLAlchemy. The procedure is to create a class like:

```
from quetzal.app import db


class Foo(db.Model):
    # ... contents, according to SQLAlchemy ...
    ...
```

Normally, this goes into the **:py:module:`quetzal.app.models`** module.

You should also add it to the dictionary created in the `make_shell_context` function inside the *`quetzal.app.create_app()`* factory function. This will automatically import the model when running `flask shell`.

Now, proceed to the *Migrations* procedure.

Migrations

Prepare a migration script

When creating or modifying a database model object, you need to make a migration script. This is done with alembic:

```
flask db migrate --rev-id ID -m MESSAGE
```

Here, `ID` is a revision identifier. Please use `0001`, `0002`, ... and so on, according to the number of the files in the `migrations/` directory. `MESSAGE` should be a short description of what the migration does.

Note that if you are running a development instance, you may need to set some environment variables for the command above to work. For example:

```
FLASK_ENV=development DB_HOST=localhost DB_USERNAME=postgres DB_PASSWORD=pg_password␣
↪flask db migrate --rev-id ID -m MESSAGE
```

The `flask db migrate` command above will create a script called `migrations/yyyymmdd_ID_MESSAGE.py`. Read and review its contents. It is important to remove any database modification that does not correspond to what you have created or modified in your models. Alembic does most of the job to create this for you, but it does make mistakes.

Apply a migration script

Finally to *apply* your migration:

```
flask db upgrade head
```

Whenever there is a model database modification and you update Quetzal, you need to run the migration script.

1.11 Testing

1.12 Deployment

1.12.1 Google Cloud Platform preparations

Quetzal can be deployed as a Kubernetes application on Google Cloud Platform (GCP). To achieve this, follow this guide.

Project

1. Create a project on the GCP console by selecting New project .

   When you create a GCP project, you will give it a unique name, its project id. In this guide, this identifier will be referred as `<your-project-id>`.

2. After creating the project, head to the IAM & admin menu to see the list of members of the project.

   Make sure that your email address is listed as a *project owner*.

3. Download and install gcloud.

   Most of the operations described in this guide can be done through the GCP console, a very rich web-based application to manage your cloud resources and services. However, this guide will do all operations on the command-line interface using `gcloud`, because it is easier to describe.

4. Once you have installed gcloud, authenticate with the email address listed in step 2.

   ```
   $ gcloud auth login
   # ... a browser window will appear to login ...
   ```

5. Configure the default settings of the project.

   ```
   $ gcloud config set project <your-project-id>
   $ gcloud config set compute/zone europe-west1-c # or some other region
   ```

5. Verify your configuration.

   ```
   $ gcloud config list
   [compute]
   region = europe-west1
   zone = europe-west1-c
   [core]
   account = your.email@example.com    # << verify that this is your email...
   disable_usage_reporting = True
   project = <your-project-id>         # << ... and that this is your GCP project
   ```

(continues on next page)

(continued from previous page)

```
Your active configuration is: [default]
```

Credentials

Quetzal uses and manages several GCP resources through the GCP JSON API. This access is subject to the permissions defined by the Identity and Access Management (IAM) component of GCP. You need to create a service account for Quetzal and associate a list of permissions to it. In other words, you need to setup some *credentials*. The following steps explain how to create these credentials.

1. Create a service account. Note the *email* entry, which will be used later.

```
$ gcloud iam service-accounts create quetzal-service-account \
    --display-name="Quetzal application service account" \
    --format json
Created service account [quetzal-service-account].
{
  "displayName": "Quetzal application service account",
  "email": "quetzal-service-account@<your-project-id>.iam.gserviceaccount.com",
  ...
}
```

2. Create a credentials key JSON file for the service account.

   In the following code example, it is saved as `conf/credentials.json`.

```
$ gcloud iam service-accounts keys create \
   conf/credentials.json \
   --iam-account=quetzal-service-account@<your-project-id>.iam.gserviceaccount.com
```

   Important: Anyone with this file could use your GCP resources, so this file should not be shared or committed to your version control system.

   Keep it secret, keep it safe.

3. Create an IAM role.

   We need to create a role that encapsulates all the permissions needed by the Quetzal application. These permissions are listed on the `gcp_role.yaml` file.

```
$ gcloud iam roles create quetzal_app_role \
  --project <your-project-id> \
  --file gcp_role.yaml
```

4. Associate the service account to the IAM role.

   Finally, the service account created before needs to be associated with the permissions defined in the IAM role.

```
$ gcloud projects add-iam-policy-binding <your-project-id> \
  --member=serviceAccount:quetzal-service-account@<your-project-id>.iam.
↪gserviceaccount.com \
  --role=projects/<your-project-id>/roles/quetzal_app_role
```

APIs

Quetzal uses several GCP services through their APIs. You need the enable the following APIs on GCP API library:

- Cloud Storage, used to store all files in Quetzal.
- Kubernetes Engine API, used to create a Kubernetes cluster that hosts the Quetzal services.

Docker & Kubernetes

Quetzal uses Docker images and the Google Container Registry (GCR).

1. Install Docker. Make sure you are able to create Docker images by following the test Docker installation instructions.
2. Use gcloud to configure a Docker registry. This will enable Docker to push images to GCR.

```
$ gcloud auth configure-docker
```

3. Finally, install the kubernetes client:

```
$ gcloud components install kubectl
```

IP address reservation

This step is optional. When deploying Quetzal, you might want to associate it to some fixed IP address (in order to associate it in your DNS records). You can reserve one IP as follows (change the region to your case):

```
$ gcloud compute addresses create quetzal-stage-server-ip \
 --description="Quetzal stage server external IP" \
 --region=europe-west1 \
 --network-tier=PREMIUM
```

Get the reserved IP with the following command:

```
$ gcloud compute addresses list
NAME                    ADDRESS/RANGE   TYPE  PURPOSE  NETWORK  REGION        SUBNET␣
↪ STATUS
quetzal-stage-server-ip  x.x.x.x                                 europe-west1        ␣
↪ RESERVED
```

Important: GCP reserved IPs incur in charges if they are not associated to a service. If you are not going to use it immediately, you may want to do this as late as possible.

1.12.2 Deploying on GCP

The following instructions create a Kubernetes (k8s) cluster with a Quetzal server running on the *staging* configuration. Change the *sandbox-* references to *prod-* to create a production server.

We need to do perform install several components: a k8s cluster, helm, ingress, certbot and the quetzal application.

Docker images

1. Follow the *Local development server* instructions and make sure that you are able to run and launch a development environment. You will need to activate your virtual environment.
2. Read and follow the *Google Cloud Platform preparations*. You will need to have a gcloud correctly configured, a JSON credentials file, and a reserved external IP address.
3. Build and upload the Docker container images to Google Container Registry.

```
$ flask quetzal deploy create-images \
  --registry eu.gcr.io/<your-project-id>
```

Kubernetes cluster

1. Create a kubernetes cluster using gcloud:

```
gcloud container clusters create quetzal-cluster \
  --num-nodes=1 \
  --enable-autoscaling --min-nodes=1 --max-nodes=4
```

2. Verify that the cluster is up and running:

```
gcloud container clusters list
NAME             LOCATION        MASTER_VERSION  MASTER_IP      MACHINE_TYPE  ␣
↪NODE_VERSION  NUM_NODES  STATUS
quetzal-cluster  europe-west1-c  1.11.7-gke.4    x.x.x.x        n1-standard-1  1.
↪11.7-gke.4  2          RUNNING
```

   If you need more resources, you can change the number of nodes with:

```
gcloud container clusters resize quetzal-cluster --size N
```

   or change the type of VM instance type for another machine type that uses more CPU or memory. This procedure is out of scope of this guide, but you can read more at the node pools documentation.

3. Verify that `kubectl` is using the correct cluster:

```
kubectl config get-contexts
```

Part 2: Helm

1. Install helm. In general, follow the installing helm guide. For the particular case of OSX (with homebrew), this can be done with:

```
brew install kubernetes-helm
```

2. Install helm k8s service account. This is explained in the helm installation guide:

```
kubectl create -f helm/rbac-config.yaml
```

3. Install helm k8s resources (also known as tiller) with a service account:

```
helm init --service-account tiller --wait
```

4. Verify that helm was correctly installed:

```
helm version

Client: &version.Version{SemVer:"v2.14.3", GitCommit:
↪"0e7f3b6637f7af8fcfddb3d2941fcc7cbebb0085", GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.14.3", GitCommit:
↪"0e7f3b6637f7af8fcfddb3d2941fcc7cbebb0085", GitTreeState:"clean"}
```

Part 3: Ingress

1. Install ingress resources. This is a prerequisite described in the ingress installation guide.

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/
↪master/deploy/static/mandatory.yaml
```

2 Install ingress. If you have a static IP reserved for the Quetzal application, you must set it here. Otherwise, remove the `--set controller.service.loadBalancerIP` flag:

```
helm install stable/nginx-ingress --set controller.service.loadBalancerIP=X.X.X.X␣
↪--name nginx-ingress
```

Certbot

This part is optional. You only need it if you want to have a signed certificate.

1. Install certbot. This part was inspired from the certbot acme nginx installation tutorial.

```
# Install the cert-manager CRDs. We must do this before installing the Helm
# chart in the next step for `release-0.9` of cert-manager:
kubectl apply -f https://raw.githubusercontent.com/jetstack/cert-manager/release-
↪0.9/deploy/manifests/00-crds.yaml

# Create the namespace for cert-manager
kubectl create namespace cert-manager

# Label the cert-manager namespace to disable resource validation
kubectl label namespace cert-manager certmanager.k8s.io/disable-validation=true

## Add the Jetstack Helm repository
```

(continues on next page)

(continued from previous page)

```
helm repo add jetstack https://charts.jetstack.io

## Updating the repo just incase it already existed
helm repo update

## Install the cert-manager helm chart
helm install \
  --name cert-manager \
  --namespace cert-manager \
  --version v0.9.1 \
  jetstack/cert-manager
```

2. Customize certbot issuer definition declared on the `helm/acme-issuer.yaml` file and install it:

```
kubectl create -f helm/acme-issuer.yaml
```

Quetzal

1. Create the TLS secret that will be used for the nginx proxy.

```
kubectl create secret tls sandbox-tls-secret \
  --cert=./conf/ssl/mysite.crt \
  --key=./conf/ssl/mysite.key
```

2. Create GCP credentials secret that will be used by the app to communicate with the GCP resources (e.g. the data buckets).

```
kubectl create secret generic sandbox-credentials-secrets \
  --from-file=./conf/credentials.json
```

3. Generate some passwords. You can do this manually, or use the following helper commands. Keep them secret, keep them safe.

```
# Generate a random password for the database user.
flask quetzal utils generate-secret-key 8
YRADKSrPzlM

# Generate a secret key for the Flask application.
flask quetzal utils generate-secret-key
sB-YgPO8ZVCmZyV5XKH0rg
```

4. Install quetzal using helm. Give it a name (like *foo*) and use the passwords generated in the previous step. Verify that all the configuration values in `helm/quetzal/values.yaml`. Also verify the `helm/quetzal/templates/ingress.yaml` file.

```
helm install \
  --set db.username=... \
  --set db.password=... \
  --set app.flaskSecretKey=... \
  --name foo ./helm/quetzal
```

   Note that it is **at this point** that you will set a database username, password and flask secret.

5. Verify that everything is running.

   You can check that all pods are running with:

```
kubectl get pods
NAME                                          READY   STATUS    RESTARTS   AGE
foo-quetzal-app-86669c8bc6-8vt9c              0/1     Pending   0          100s
foo-quetzal-app-86669c8bc6-dhwj6              1/1     Running   0          10m
foo-quetzal-app-86669c8bc6-s56wl              0/1     Pending   0          115s
foo-quetzal-app-86669c8bc6-w2ppm              0/1     Pending   0          115s
foo-quetzal-app-86669c8bc6-x5gvk              0/1     Pending   0          115s
foo-quetzal-db-cd68d97bc-tdj8l                1/1     Running   0          15m
foo-quetzal-rabbitmq-85bf9dddfd-kkvr7         1/1     Running   0          15m
foo-quetzal-worker-5dbb8c4dfd-fg8ct           1/1     Running   0
↪9m41s
foo-quetzal-worker-5dbb8c4dfd-fv6bj           1/1     Running   0          10m
nginx-ingress-controller-84df6c4c54-2v8n4     1/1     Running   0          22m
nginx-ingress-default-backend-7d5dd85c4c-mc89t   1/1     Running   0          22m
```

Similarly, you can do the same with the services:

```
kubectl get services
NAME                            TYPE           CLUSTER-IP     EXTERNAL-IP
↪PORT(S)                       AGE
app                             ClusterIP      10.0.11.94     <none>          5000/
↪TCP                    16m
db                              ClusterIP      10.0.13.162    <none>          5432/
↪TCP                    16m
kubernetes                      ClusterIP      10.0.0.1       <none>          443/
↪TCP                     26m
nginx-ingress-controller        LoadBalancer   10.0.3.187     x.x.x.x.
↪80:31388/TCP,443:32725/TCP   23m
nginx-ingress-default-backend   ClusterIP      10.0.11.182    <none>          80/
↪TCP                      23m
rabbitmq                        ClusterIP      10.0.10.159    <none>          5672/
↪TCP,15672/TCP          16m
```

If a pod fails to start correctly, examine it with:

```
kubectl describe pod foo-quetzal-app-7dcc756c9d-78n5w
... many details that can help determine the problem ...
```

6. Initialize the application.

   If this is the first time the application is deployed, you need to initialize its database, buckets and users. Connect to a web pod (like `foo-quetzal-app-7dcc756c9d-78n5w`, as listed above, but this will be specific to your deployment) as:

   ```
   kubectl exec -it foo-quetzal-app-7dcc756c9d-78n5w /bin/bash
   ```

   and then run the initialization script:

   ```
   ./init.sh
   ```

   which will ask for an administrator password. You can add new users at this point with:

   ```
   flask quetzal user create alice alice.smith@example.com
   flask quetzal role add alice public_read public_write
   ```

7. If you installed certbot, you should verify that the certificate was correctly generated with:

```
kubectl get certificates
NAME                 READY   SECRET               AGE
sandbox-tls-secret   True    sandbox-tls-secret   1m
```

And also, the following curl command should work without any errors:

```
curl -vL https://sandbox.quetz.al/healthz
```

That's all, you can now explore the documentation at https://sandbox.quetz.al/redoc, or wherever your configuration points to.

1.12.3 Backups

Database

When deploying Quetzal as a kubernetes application (in GCP, for example), there is a CronJob configured to save database dumps on a bucket dedicated to backups, once a week.

Manual backup

You can trigger a manual backup with kubernetes as follows:

```
# use this first command to determine CRONJOB_NAME
kubectl get cronjobs -l app.kubernetes.io/component=database
# then, create the job with:
kubectl create job --from=cronjob/CRONJOB_NAME CRONJOB_NAME-manual-001
```

Restoring a backup

The following procedure can restore one of these backups:

1. Connect to the database pod:

   ```
   # use this first command to determine DB_POD_NAME
   kubectl get pods -l app.kubernetes.io/component=database
   # then, connect with:
   kubectl exec -it DB_POD_NAME bash
   ```

2. Download and uncompress the backup file:

   ```
   gsutil cp gs://BACKUP_BUCKET/db/BACKUP_NAME.bak.gz
   gunzip BACKUP_NAME.bak
   ```

3. Make the following modifications on the `BACKUP_NAME.bak` file. You will need an editor on the pod so install one with:

   ```
   apt-get update && apt-get install --no-install-recommends --yes vim
   vim BACKUP_NAME.bak
   ```

   Then, comment (by adding `--` at the beginning of each line) the following lines:

```
CREATE ROLE dbuser;
ALTER ROLE dbuser;
...
CREATE DATABASE dbuser;
```

where `dbuser` is the username that you had set for the database when you deployed quetzal to kubernetes using helm.

4. Put quetzal on maintenance mode.

   Not sure how to do this yet.

Danger: On the following steps, you will erase your current database. Handle with care, because you may lose your data.

You may want to do a *Manual backup* first.

5. Connect to postgres, disconnect any connection and drop the database:

```
# First, connect to postgres but not to the quetzal database:
psql -U$POSTGRES_USER $POSTGRES_USER
```

```
SELECT pg_terminate_backend(pg_stat_activity.pid)
FROM pg_stat_activity
WHERE pg_stat_activity.datname = 'quetzal' -- change this if you changed the␣
↪quetzal database name
  AND pid <> pg_backend_pid();
```

```
DROP DATABASE quetzal;  -- change this if you change the quetzal database name
DROP DATABASE unittests;
DROP ROLE db_user;
DROP ROLE db_ro_user;
exit;
```

6. Restore the database from the backup:

```
psql -U$POSTGRES_USER --set ON_ERROR_STOP=on -f ./BACKUP_NAME.bak
```

1.13 Development

1.13.1 Local development server

The following instructions assume that you have are using Linux or OSX. For instructions under Windows, please help us by adapting them and filing a pull request.

1. Clone the Quetzal repository:

```
$ git clone git@github.com:quetz-al/quetzal.git
```

2. Install Docker. Make sure you are able to create Docker images by following the test Docker installation instructions.

3. Create a virtual environment with your favorite virtual environment manager, but make sure it is a Python 3 environment. Then, install the requirement libraries:

```
$ python3 -m venv ${HOME}/.virtualenvs/quetzal-env
$ source ${HOME}/.virtualenvs/quetzal-env/bin/activate
$ pip install -r requirements-dev.txt
```

At this point, you will need to prepare your Google Cloud Platform credentials (if you are going to use Google buckets to save data files) and prepare SSL certificates.

Google Cloud Platform

Using Google buckets to save data needs some preparations described in *Google Cloud Platform preparations*. For a development server you need to follow the *Project*, *Credentials* and *APIs* instructions.

SSL

Quetzal uses HTTPS for all its API operations. This needs a SSL certificate that can be generated as follows.

1. First, create a SSL key and certificate using `openssl`:

```
$ mkdir -p conf/ssl
$ openssl req -x509 -newkey rsa:4096 \
  -keyout conf/ssl/mysite.key -out conf/ssl/mysite.crt \
  -days 365 -nodes
```

2. Optionally, but highly recommended, generate a DH exchange key prime number:

```
$ openssl dhparam -out conf/ssl/dhparam.pem 2048
```

Note that these are auto-signed keys and they are only suitable for a development or testing scenario. When deploying on a production server, the recommended approach is to use Let's Encrypt as a certificate authority and CertBot to obtain the final, signed certificates. However, you will still need these auto-signed keys as a temporary solution until CertBot runs the first time.

Docker-compose

We are almost ready to have a Quetzal development server ready. This local server runs as a multi-container application managed by docker-compose.

1. Read the configuration entries in `config.py` and change them accordingly in the `docker-compose.yaml` file.

   If you are going to use Google buckets to store data, follow the instructions concerning the *Google Cloud Platform* and verify the configuration variables with the `QUETZAL_GCP_` prefix.

   If you prefer saving your files locally, set the `QUETZAL_DATA_STORAGE` to `'file'` and ignore the instructions related to Google Cloud Platform.

2. Build your docker-compose services:

```
$ docker-compose build
```

3. Run Quetzal through docker-compose:

```
$ docker-compose up
```

4. If this the first time you run Quetzal, you need to setup the database, create some roles and users. You can do this while the server is running with the following script:

```
$ docker-compose exec web ./init.sh
```

Usage notes

If you want to stop the Quetzal application, use:

```
$ docker-compose stop
```

To reset and erase the Quetzal application, use:

```
$ docker-compose down
```

Warning: Using `docker-compose down` will erase your database. You will lose your data. Use this only to reset and start a fresh Quetzal application.

1.13.2 Reference

Flask-SQLAlchemy Models

class quetzal.app.models.**ApiKey**(***kwargs*)

class quetzal.app.models.**BaseMetadataKeys**
Set of metadata keys that exist in the base metadata family

The base metadata family is completely managed by Quetzal; a user cannot set or change its values (with the exception of the value for the *path* or *filename* keys). This enumeration defines the set of keys that exist in this family.

CHECKSUM = 'checksum'
MD5 checksum of the file

DATE = 'date'
Date when this file was created.

FILENAME = 'filename'
Filename, without its path component.

ID = 'id'
Unique file identifier.

PATH = 'path'
Path component of the filename.

SIZE = 'size'
Size in bytes of the file.

STATE = 'state'
State of the file; see `FileState`.

URL = 'url'
URL where this file is stored.

class quetzal.app.models.**Family**(***kwargs*)
Quetzal metadata family

In quetzal, metadata are organized in semantic groups that have a name and a version number. This is the definition of a metadata _family_. This class represents this definition. It is attached to a workspace, until the workspace is committed: at this point the family will be disassociated from the workspace to become *global* (available as public information).

id
Identifier and primary key of a family.

Type int

name
Name of the family.

Type str

version
Version of the family. Can be `None` during a workspace creation, and until its initialization, to express the *latest* available version.

Type int

description
Human-readable description of the family and its contents, documentation, and any other useful comment.

Type str

fk_workspace_id
Reference to the workspace that uses this family. When `None`, it means that this family and all its associated metadata is public.

Type int

Extra attributes

metadata_set All `Metadata` entries associated to this family.

increment()
Create a new family with the same name but next version number

The new family will be associated to the same workspace.

class quetzal.app.models.**FileState**
State of a Quetzal file

Quetzal files have a status, saved in their *base* metadata under the *state* key. It can only have the values defined in this enumeration.

DELETED = 'deleted'
File has been deleted.

Deleted files will have their metadata cleared when the workspace is committed.

If it was an already committed file, its contents will not be removed from the global data storage directory or bucket, but its metadata will be cleared. If it was a file that was not committed yet, it will be erased from its workspace data directory or bucket.

Deleted files are not considered in queries.

READY = 'ready'
File is ready

It has been uploaded, it can be downloaded, its metadata can be changed and when its workspace is committed, it will be moved to the global data storage directory or bucket.

TEMPORARY = 'temporary'
File is ready but temporary

Like `READY`, but this file will not be considered when the workspace is committed. That is, it will not be copied to the global data storage directory or bucket.

class quetzal.app.models.**Metadata**(**kwargs*)
Quetzal unstructured metadata

Quetzal defines metadata as a dictionary associated with a family. Families define the semantic organization and versioning of metadata, while this class gathers all the metadata key and values in a dictionary, represented as a JSON object.

id
Identifier and primary key of a metadata entry.

Type int

id_file
Unique identifier of a file as a UUID number version 4. This identifier is also present and must be the same as the *id* entry in the *json* member.

Type uuid.UUID

json
A json representation of metadata. Keys are metadata names and values are the related values. It may be a nested object if needed.

Type dict

Extra attributes

family The related `Family` associated to this metadata.

static get_latest(*file_id*, *family*)
Retrieve the latest metadata of a file under a particular family

static get_latest_global(*file_id=None*, *family_name=None*)
Retrieve the latest metadata of a file under a particular family

to_dict()
Return a dictionary representation of the metadata

Used to conform to the metadata details object on the OpenAPI specification.

Returns Dictionary representation of this object.

Return type dict

update(*json*)
Update the underlying json metadata with the values of a new one

This function takes the current json saved in this metadata object and updates it (like `dict.update`) with the new values found in the *json* input parameter. This does not remove any key; it adds new keys or changes any existing one.

Since SQLAlchemy does not detect changes on a JSONB column unless a new object is assigned to it, this function creates a new dictionary and replaces the previous one.

Changes still need to be committed through a DB session object.

Parameters **json** (*dict*) – A new metadata object that will update over the existing one

Returns

Return type self

class quetzal.app.models.**MetadataQuery**(***kwargs*)
Query for metadata on Quetzal

Queries on Quetzal are temporarily saved as objects. This was initially thought as a mechanism for easier and faster paginations, to avoid verifying that a query is valid every time and possibly to compile these queries if needed.

id
Identifier and primary key of a query.

Type int

dialect
Dialect used on this query.

Type QueryDialect

code
String representation of the query. May change in the future.

Type str

fk_workspace_id
Reference to the Workspace where this query is applied. If None, the query is applied on the global, committed metadata.

Type int

fk_user_id
Reference to the User who created this query.

Type int

static get_or_404(*qid*)
Get a workspace by id or raise an APIException

static get_or_create(*dialect*, *code*, *workspace*, *owner*)
Retrieve a query by its fields or create a new one

to_dict(*results=None*)
Create a dict representation of the query and its results

Used to conform to the OpenAPI specification of the paginable query results

Parameters **results** (*dict*) – Results as a paginable object.

Returns Dictionary representation of this object.

Return type dict

class quetzal.app.models.**QueryDialect**
Query dialects supported by Quetzal

class quetzal.app.models.**Role**(***kwargs*)
Authorization management role on Quetzal

Quetzal operations are protected by an authorization system based on roles. A user may have one to many roles; a role defines what operations the associated users can do.

Note that the *n to n* relationship of roles and users is implemented through the roles_users_table.

id
Identifier and primary key of a role.

Type int

name
Unique name of the role.

Type str

description
Human-readable description of the role.

Type str

Extra attributes

users Set of users associated with this role. This attribute is defined through a backref in `User`.

class quetzal.app.models.**User**(***kwargs*)
Quetzal user

Almost all operations on Quetzal can only be done with an authenticated user. This model defines the internal information that Quetzal needs for bookeeping its users, permissions, emails, etc.

id
Identifier and primary key of a user.

Type int

username
Unique string identifier of a user (e.g. admin, alice, bob).

Type str

email
Unique e-mail address of a user.

Type str

password_hash
Internal representation of the user password with salt.

Type str

token
Unique, temporary authorization token.

Type str

token_expiration
Expiration date of autorization token.

Type datetime

active
Whether this user is active (and consequently can perform operations) or not.

Type bool

Extra attributes

roles Set of `Roles` associated with this user.

workspaces Set of `Workspaces` owned by this user.

queries Set of `Queries` created by this user.

check_password(*password*)
Check if a password is correct.

Parameters **password** (*str*) – The password to verify against the hash-salted stored password.

Returns `True` when the provided password matches the hash-salted stored one.

Return type bool

static check_token(*token*)
Retrieve a user by token

No user will be returned when the token is expired or does not exist.

Parameters **token** (*str*) – Authorization token.

Returns **user** – User with the provided token, or `None` when either the token was not found or it was expired.

Return type `User`

get_token(*expires_in=3600*)
Create or retrieve an authorization token

When a user already has an authorization token, it returns it.

If there is no authorization token or the existing authorization token for this user is expired, this function will create a new one as a random string.

The changes on this instance are not propagated to the database (this must be done by the caller), but this instance added to the current database session.

Parameters **expires_in** (*int*) – Expiration time, in seconds from the current date, used when creating a new token.

Returns The authorization token

Return type str

property is_active
Property accessor for *active*.

Needed to conform to the `flask_login.UserMixin` interface.

revoke_token()
Revoke the authorization token

The changes on this instance are not propagated to the database (this must be done by the caller), but this instance added to the current database session.

set_password(*password*)
Change the password of this user.

This function set and store the new password as a salt-hashed string.

The changes on this instance are not propagated to the database (this must be done by the caller), but this instance added to the current database session.

Parameters **password** (*str*) – The new password.

class quetzal.app.models.**Workspace**(***kwargs*)

Quetzal workspace

In Quetzal, all operations on files and metadata are *sandboxed* in workspaces. Workspaces define the exact metadata families and versions, which in turn provides a snapshot of what files and metadata are available. This is the base of the reproducibility of dataset in Quetzal and the traceability of the data changes.

Workspaces also provide a storage directory or bucket where the user can upload new and temporary data files.

id

Identifier and primary key of a workspace.

Type int

name

Short name for a workspace. Unique together with the owner's username.

Type str

_state

State of the workspace. Do not use directly, use its property accessors.

Type `WorkspaceState`

description

Human-readable description of the workspace, its purpose, and any other useful comment.

Type str

creation_date

Date when the workspace was created.

Type datetime

temporary

When `True`, Quetzal will know that this workspace is intended for temporary operations and may be deleted automatically when not used for a while. When `False`, only its owner may delete it.

Type bool

data_url

URL to the data directory or bucket where new files associated to this workspace will be saved.

Type str

pg_schema_name

Used when creating structured views of the structured metadata, this schema name is the postgresql schema where temporary tables exists with a copy of the unstructured metadata.

Type str

fk_user_id

Owner of this workspace as a foreign key to a `User`.

Type int

fk_last_metadata_id

Reference to the most recent `Metadata` object that has been committed at the time when this workspace was created. This permits to have a reference to which global metadata entries should be taken into account when determining the metadata in this workspace.

Type int

Extra attributes

families Set of `Families` (including its version) used for this workspace.

queries Set of `Queries` created on this workspace.

property can_change_metadata
Returns `True` when metadata can be changed on the current workspace state

get_base_family()
Get the base family instance associated with this workspace

get_current_metadata()
Get the metadata that has been added or modified in this workspace

In contrast to `get_previous_metadata()`, this function only retrieves the metadata that has been changed on this workspace after its creation.

get_metadata()
Get a union of the previous and new metadata of this workspace

This function uses a combination of the results of `get_previous_metadata()` and `get_current_metadata()` to obtain the merged version of both. This represents the definitive metadata of each file, regardless of changes before or after the creation of this workspace.

static get_or_404(*wid*)
Get a workspace by id or raise a *quetzal.app.api.exceptions.ObjectNotFoundException*

get_previous_metadata()
Get the global metadata of this workspace

The global metadata is the metadata that already has been committed, but it must also have a version value that is under the values declared for this workspace.

make_schema_name()
Generate a unique schema name for its internal structured metadata views

property state
Property accessor for the workspace state

to_dict()
Return a dictionary representation of the workspace

This is used in particular to adhere to the OpenAPI specification of workspace details objects.

Returns Dictionary representation of this object.

Return type dict

class quetzal.app.models.**WorkspaceState**
Status of a workspace.

Workspaces in Quetzal have a state that defines what operations can be performed on them. This addresses the need for long-running tasks that modify the workspace, such as initialization, committing, deleting, etc.

The transitions from one state to another is defined on this enumeration on the `transitions()` function. The following diagram illustrates the possible state transitions:

The verification of state transitions is implemented in the `quetzal.app.models.Workspace.state` property setter function.

COMMITTING = 'committing'
The workspace is committing its files and metadata.

The workpace will remain on this state until the committing routine finishes. No operation is possible until then.

CONFLICT = 'conflict'
The workspace detected a conflict during its commit routine.

The workpace will remain on this state until the administrator fixes this situation. No operation is possible.

DELETED = 'deleted'
The workspace has been deleted.

The instance of the workspace remains in database for bookeeping, but there is no operation possible with it at this point.

DELETING = 'deleting'
The workspace is deleting its files and itself.

The workpace will remain on this state until the deleting routine finishes. No operation is possible.

INITIALIZING = 'initializing'
The workspace has just been created.

The workspace will remain on this state until the initialization routine finishes. No operation is possible until then.

INVALID = 'invalid'
The workspace has encountered an unexpected error.

The workpace will remain on this state until the administrator fixes this situation. No operation is possible.

READY = 'ready'
The workspace is ready.

The workspace can now be scanned, updated, committed or deleted. Files can be uploaded to it and their metadata can be changed.

SCANNING = 'scanning'
The workspace is updating its internal views.

The workpace will remain on this state until the scanning routine finishes. No operation is possible until then.

UPDATING = 'updating'
The workspace is updating its metadata version definition.

The workpace will remain on this state until the updating routine finishes. No operation is possible until then.

quetzal.app.models.**roles_users_table = Table('roles_users', MetaData(bind=None), Column('fl**
Auxiliary table associating users and roles

All modules

quetzal package

Subpackages

quetzal.app package

quetzal.app.**create_app**(*config_name=None*)

Subpackages

quetzal.app.api package

Subpackages

quetzal.app.api.data package

Subpackages

quetzal.app.api.data.storage package

quetzal.app.api.data.storage.**set_permissions**(*file_obj*, *owner*)
Set the permissions of the file

Change the data object *file_obj* permissions to set *owner* as the user that owns this file.

Parameters

- **file_obj** (*object*) – Object pointing to a file, as returned by *upload()*.
- **owner** (*quetzal.app.models.User*) – User object that will own the file.

Raises ***quetzal.app.api.exceptions.QuetzalException*** – When the storage backend is unknown. Exceptions by the dispatched functions are not captured here.

quetzal.app.api.data.storage.**upload**(*filename*, *contents*, *location*)
Upload a file

Upload the *contents* as a file named *filename* in *location*.

This function dispaches the upload operation on the configured storage backend.

Parameters

- **filename** (*str*) – Target file name where the contents will be saved.
- **contents** (*file-like*) – A buffer of bytes with the file contents.
- **location** (*str*) – URL of the target location where the file will be saved. This should be the URL of a workspace

Returns

- **url** (*str*) – URL to where the file was uploaded.
- **obj** (*object*) – An object pointing where the file was uploaded for further manipulation. Its type depends on the data backend.

Raises ***quetzal.app.api.exceptions.QuetzalException*** – When the storage backend is unknown. Exceptions by the dispatched functions are not captured here.

Submodules

quetzal.app.api.data.storage.gcp module

quetzal.app.api.data.storage.gcp.**set_permissions**(*blob*, *owner*)

quetzal.app.api.data.storage.gcp.**upload**(*filename*, *content*, *location*)
Save a file on a local filesystem.

Implements the *upload* mechanism of the GCP backend.

Parameters

- **filename** (*str*) – Filename where the file will be saved. It can include a relative path.
- **content** (*file-like*) – Contents of the file.
- **location** (*str*) – URL of the bucket the file will be saved. The *filename* parameter will be relative to this parameter.

Returns

- **url** (*str*) – URL to the uploaded file. Its format will be `gs://url/to/file`.
- **blob_obj** (`google.cloud.storage.blob.Blob`) – Blob object where the file was saved.

Raises ***quetzal.app.api.exceptions.QuetzalException*** – When the location is the global data bucket. This is not permitted.

quetzal.app.api.data.storage.local module

quetzal.app.api.data.storage.local.**set_permissions**(*file_obj*, *owner*)

quetzal.app.api.data.storage.local.**upload**(*filename*, *content*, *location*)
Save a file on a local filesystem.

Implements the *upload* mechanism of the local file storage backend.

Parameters

- **filename** (*str*) – Filename where the file will be saved. It can include a relative path.
- **content** (*file-like*) – Contents of the file.
- **location** (*str*) – URL where the file will be saved. The *filename* parameter will be relative to this parameter.

Returns

- **url** (*str*) – URL to the uploaded file. Its format will be `file://absolute/path/to/file`.
- **path_obj** (`pathlib.Path`) – Path object where the file was saved.

Raises ***quetzal.app.api.exceptions.QuetzalException*** – When the location is the global data directory. This is not permitted.

Submodules

quetzal.app.api.data.file module

quetzal.app.api.data.file.**_all_metadata**(*file_id*, *workspace*)
Gather all metadata of a file in a workspace

If a file has metadata of families f1, f2, ..., this function returns a dictionary `{'f1': {...}, 'f2': {...}, ...}`. This structure is suitable for the responses of file fetch metadata operations.

quetzal.app.api.data.file.**_now**()
Get a datetime object with the current datetime (in UTC) as a string

This function is also created for ease of unit test mocks

quetzal.app.api.data.file.**_verify_filename_path**(*filename*, *path*)
Perform some security considerations on filename and path

quetzal.app.api.data.file.**create**(*, *wid*, *content=None*, *user*, *token_info=None*)
Create a file on a workspace

This function is the implementation of the upload file endpoint in the Quetzal API. After verifying the workspace and user permissions, it will save the contents of the file in the configured file backend. Finally, it initializes the base metadata family entries for the new file.

Parameters

- **wid** (*int*) – Workspace identifier where the file will be uploaded.
- **content** (*file-like*) – Contents of the file.
- **user** (*quetzal.app.models.User*) – User that owns the file. This parameter is set by connexion.
- **token_info** – Authentication token. This parameter is set by connexion.

Returns

- **details** (*dict*) – File details object.
- **code** (*int*) – HTTP response code.

API endpoints

- *POST /api/v1/data/workspaces/{wid}/files/* See in redoc.

quetzal.app.api.data.file.**delete**(*, *wid*, *uuid*, *user*, *token_info=None*)

quetzal.app.api.data.file.**details**(*, *uuid*)
Get the contents or metadata of a file that has been committed

quetzal.app.api.data.file.**details_w**(*, *wid=None*, *uuid*)
Get contents or metadata of a file on a workspace

quetzal.app.api.data.file.**fetch**(**args*, ***kwargs*)
Get all the files that have been committed.

quetzal.app.api.data.file.**fetch_w**(*, *wid*)
Get all the files on a workspace

quetzal.app.api.data.file.**set_metadata**(*, *wid*, *uuid*, *body*)

quetzal.app.api.data.file.**update_metadata**(*, *wid*, *uuid*, *body*)

quetzal.app.api.data.query module

quetzal.app.api.data.query.**create**(*, *body*, *user*, *token_info=None*)

quetzal.app.api.data.query.**create_w**(*, *wid*, *body*, *user*, *token_info=None*)

quetzal.app.api.data.query.**details**(*, *qid*, *user*, *token_info=None*)

quetzal.app.api.data.query.**details_w**(*, *wid*, *qid*, *user*, *token_info=None*)

quetzal.app.api.data.query.**fetch**(*, *user*, *token_info=None*)

quetzal.app.api.data.query.**fetch_w**(*, *wid*, *user*, *token_info=None*)

quetzal.app.api.data.tasks module

quetzal.app.api.data.tasks.**merge**(*ancestor*, *theirs*, *mine*)

quetzal.app.api.data.workspace module

quetzal.app.api.data.workspace.**commit**(*, *wid*)
Request commit of all metadata and files of a workspace

Parameters **wid** (*int*) – Workspace identifier

Returns

- *dict* – Workspace details
- *int* – HTTP response code

quetzal.app.api.data.workspace.**create**(*, *body*, *user*, *token_info=None*)
Create a new workspace

Returns

- *dict* – Workspace details
- *int* – HTTP response code

quetzal.app.api.data.workspace.**delete**(*, *user*, *wid*)
Request deletion of a workspace by id

Parameters **wid** (*int*) – Workspace identifier

Returns

- *dict* – Workspace details
- *int* – HTTP response code

quetzal.app.api.data.workspace.**details**(*, *wid*)
Get workspace details by id

Parameters **wid** (*int*) – Workspace identifier

Returns

- *dict* – Workspace details
- *int* – HTTP response code

quetzal.app.api.data.workspace.**fetch**(*, *user*)
List workspaces

Returns

- *list* – List of Workspace details as a dictionaries
- *int* – HTTP response code

quetzal.app.api.data.workspace.**scan**(*, *wid*)
Request an update of the views of a workspace

Parameters **wid** (*int*) – Workspace identifier

Returns

- *dict* – Workspace details
- *int* – HTTP response code

Submodules

quetzal.app.api.auth module

quetzal.app.api.auth.**check_apikey**(*key*, *required_scopes=None*)

quetzal.app.api.auth.**check_basic**(*username*, *password*, *required_scopes=None*)

quetzal.app.api.auth.**check_bearer**(*token*)

quetzal.app.api.auth.**get_token**(*, *user*)

quetzal.app.api.auth.**logout**(*, *user*)

quetzal.app.api.exceptions module

exception quetzal.app.api.exceptions.**APIException**(*status=400*, *title=None*, *detail=None*, *type=None*, *instance=None*, *headers=None*, *ext=None*)
Bases: connexion.exceptions.ProblemException

Exception for API-related problems

Use this class when a route function fails but the API should respond with an appropriate error response

exception quetzal.app.api.exceptions.**Conflict**
Bases: *quetzal.app.api.exceptions.QuetzalException*

exception quetzal.app.api.exceptions.**EmptyCommit**
Bases: *quetzal.app.api.exceptions.QuetzalException*

exception quetzal.app.api.exceptions.**InvalidTransitionException**
Bases: *quetzal.app.api.exceptions.QuetzalException*

exception quetzal.app.api.exceptions.**ObjectNotFoundException**(*status=400*, *title=None*, *detail=None*, *type=None*, *instance=None*, *headers=None*, *ext=None*)
Bases: *quetzal.app.api.exceptions.APIException*

Exception for cases when an object does not exist

Typically, when a workspace or file does not exist

exception quetzal.app.api.exceptions.**QuetzalException**
Bases: Exception

Represents an internal error in the data API

Use for exceptions that don't need to be transmitted back as a response

exception quetzal.app.api.exceptions.**WorkerException**
Bases: *quetzal.app.api.exceptions.QuetzalException*

quetzal.app.api.router module

Router controller to let Connexion link OAS operation ids to custom functions.

On a OpenAPI specification, operationIds can be as specific as: `app.api.data.workspace.create`. However, other clients may use this long name, which generates functions with long names. Moreover, the real use-case of operationId is to provide a unique identifier to each operation.

To simplify the client code, we use Connexion's vendor-specific tag `x-openapi-router-controller` to provide a class to associate operations to Python functions. Following Connexion's implementation, the resolved name is `controller.operationId` where `controller` is the value of the `x-openapi-router-controller` tag.

This Python function provides the functions and associations to use the `x-openapi-router-controller` tag and simplify the specification code.

class quetzal.app.api.router.**AuthRouter**
Bases: object

Router for authentication operations.

Use as:

```
operationId: auth.func
x-openapi-router-controller: app.api.router
```

Where `func` is a member of this class.

get_token()

logout()

class quetzal.app.api.router.**PublicRouter**
Bases: object

Router for operations on public resources.

Use as:

```
operationId: public.func
x-openapi-router-controller: app.api.router
```

Where `func` is a member of this class.

file_details()
Get the contents or metadata of a file that has been committed

file_fetch(***kwargs*)
Get all the files that have been committed.

query_create(*, *user*, *token_info=None*)

query_details(*, *user*, *token_info=None*)

query_fetch(*, *token_info=None*)

class quetzal.app.api.router.**WorkspaceFilesRouter**
Bases: object

Router for operations on files inside a workspace.

Use as:

```
operationId: workspace_file.func
x-openapi-router-controller: app.api.router
```

Where `func` is a member of this class.

create(*, *content=None*, *user*, *token_info=None*)
Create a file on a workspace

This function is the implementation of the upload file endpoint in the Quetzal API. After verifying the workspace and user permissions, it will save the contents of the file in the configured file backend. Finally, it initializes the base metadata family entries for the new file.

Parameters

- **wid** (*int*) – Workspace identifier where the file will be uploaded.
- **content** (*file-like*) – Contents of the file.
- **user** (*quetzal.app.models.User*) – User that owns the file. This parameter is set by connexion.
- **token_info** – Authentication token. This parameter is set by connexion.

Returns

- **details** (*dict*) – File details object.
- **code** (*int*) – HTTP response code.

API endpoints

- *POST /api/v1/data/workspaces/{wid}/files/* See in redoc.

delete(*, *uuid*, *user*, *token_info=None*)

details(*, *uuid*)
Get contents or metadata of a file on a workspace

fetch()
Get all the files on a workspace

set_metadata(*, *uuid*, *body*)

update_metadata(*, *uuid*, *body*)

class quetzal.app.api.router.**WorkspaceQueryRouter**
Bases: object

Router for operations on queries inside a workspace.

Use as:

```
operationId: workspace_query.func
x-openapi-router-controller: app.api.router
```

Where `func` is a member of this class.

create(*, *body*, *user*, *token_info=None*)

details(*, *qid*, *user*, *token_info=None*)

fetch(*, *user*, *token_info=None*)

class quetzal.app.api.router.**WorkspaceRouter**

Bases: `object`

Router for workspace operations.

Use as:

```
operationId: workspace.func
x-openapi-router-controller: app.api.router
```

Where `func` is a member of this class.

commit()

Request commit of all metadata and files of a workspace

Parameters **wid** (*int*) – Workspace identifier

Returns

- *dict* – Workspace details
- *int* – HTTP response code

create(*, *user*, *token_info=None*)

Create a new workspace

Returns

- *dict* – Workspace details
- *int* – HTTP response code

delete(*, *wid*)

Request deletion of a workspace by id

Parameters **wid** (*int*) – Workspace identifier

Returns

- *dict* – Workspace details
- *int* – HTTP response code

details()

Get workspace details by id

Parameters **wid** (*int*) – Workspace identifier

Returns

- *dict* – Workspace details
- *int* – HTTP response code

fetch()

List workspaces

Returns

- *list* – List of Workspace details as a dictionaries
- *int* – HTTP response code

scan()
Request an update of the views of a workspace

Parameters **wid** (*int*) – Workspace identifier

Returns

- *dict* – Workspace details
- *int* – HTTP response code

quetzal.app.api.router.**auth**
alias of *quetzal.app.api.router.AuthRouter*

quetzal.app.api.router.**public**
alias of *quetzal.app.api.router.PublicRouter*

quetzal.app.api.router.**workspace**
alias of *quetzal.app.api.router.WorkspaceRouter*

quetzal.app.api.router.**workspace_file**
alias of *quetzal.app.api.router.WorkspaceFilesRouter*

quetzal.app.api.router.**workspace_query**
alias of *quetzal.app.api.router.WorkspaceQueryRouter*

quetzal.app.cli package

Submodules

quetzal.app.cli.data module

quetzal.app.cli.deployment module

quetzal.app.cli.users module

quetzal.app.cli.utils module

quetzal.app.helpers package

Submodules

quetzal.app.helpers.celery module

Improved version of the celery object of flask_celery_helper package

The original flask_celery.Celery object has a bug where the context is pushed during unit tests. This file extends and rewrites the related code to avoid this bug.

This bug has been documented on: https://github.com/Robpol86/Flask-Celery-Helper/issues/23

class quetzal.app.helpers.celery.**Celery**(*app=None*)
Bases: flask_celery.Celery

init_app(*app*)
Actual method to read celery settings from app configuration and initialize the celery instance.

Positional arguments: app – Flask application instance.

quetzal.app.helpers.celery.**_mockable_call**(*base*, *obj*, **args*, ***kwargs*)
Helper function to replace Task.__call__ for mockable tests

quetzal.app.helpers.celery.**log_task**(*task*, *level=20*, *limit=10*, *_logger=None*)
Log the ids of a task or chain of tasks in celery

quetzal.app.helpers.files module

quetzal.app.helpers.files.**get_readable_info**(*file_obj*)
Extract useful information from reading a file

This function calculates the md5sum and the file size in bytes from a file-like object. It does both operations at the same time, which means that there is no need to read the object twice.

After this function reads the file content, it will set the file pointer to its original position through *tell*.

Parameters **file_obj** (*file-like*) – File object. It needs the *read* and *tell* methods.

Returns **md5sum, size** – MD5 sum and size of the file object contents

Return type str, int

quetzal.app.helpers.files.**split_check_path**(*filepath*)

quetzal.app.helpers.google_api module

quetzal.app.helpers.google_api.**get_bucket**(*url*, ***, *client=None*)
Get a GCP bucket object from an URL

Parameters

- **url** (*str*) – URL of the bucket
- **client** (*google.storage.client.Client, optional*) – GCP client instance to use. If not set it uses *get_client()*.

Returns **bucket** – A bucket instance

Return type google.storage.bucket.Bucket

quetzal.app.helpers.google_api.**get_client**()
Create a GCP client built from the app configuration

The client is saved in the currrent application context and will be reused in any future call on this context.

quetzal.app.helpers.google_api.**get_data_bucket**(***, *client=None*)
Get Quetzal's data bucket

Parameters **client** (*google.storage.client.Client, optional*) – GCP client instance to use. If not set it uses *get_client()*.

Returns **bucket** – A bucket instance

Return type google.storage.bucket.Bucket

quetzal.app.helpers.google_api.**get_object**(*url*, *, *client=None*)

quetzal.app.helpers.pagination module

class quetzal.app.helpers.pagination.**CustomPagination**(**args*, ***kwargs*)
Bases: flask_sqlalchemy.Pagination

A specialization of flask_sqlalchemy pagination object

This specialization adds a utility method to produce a dictionary that conforms to Quetzal's pagination specification.

In addition to the original constructor parameters, this object takes a serializer method that converts whatever object type that the query from the paginate call generates and converts it to an object that must be JSON serializable. If not provided, the object is used as-is.

next(*error_out=False*)
Returns a Pagination object for the next page.

prev(*error_out=False*)
Returns a Pagination object for the previous page.

response_object()

quetzal.app.helpers.pagination.**paginate**(*queriable*, *, *page=None*, *per_page=None*, *error_out=True*, *max_per_page=None*, *serializer=None*)
Returns per_page items from page page.

This is a specialization of *flask_sqlalchemy.BaseQuery.paginate* with some custom modifications:

- It changes the original behavior to respond throw *APIException* instead of calling *abort*. The status code has also been changed to 400 instead of 404. Normally, this errors should not be reachable since connexion handles the input validation.
- In addition to handling regular *flask_sqlalchemy.BaseQuery* objects, it can also accept a cursor.
- In addition to these changes, this function returns a custom pagination object that provides a *response_object* method that can build a response according to Quetzal's paginated response specification.
- Uses keyword arguments to avoid incorrect arguments

The original docstring is as follows:

If page or per_page are None, they will be retrieved from the request query. If max_per_page is specified, per_page will be limited to that value. If there is no request or they aren't in the query, they default to 1 and 20 respectively.

When error_out is True (default), the following rules will cause a 404 response:

- No items are found and page is not 1.
- page is less than 1, or per_page is negative.
- page or per_page are not ints.

When error_out is False, page and per_page default to 1 and 20 respectively.

Returns a *CustomPagination* object.

quetzal.app.helpers.sql module

class quetzal.app.helpers.sql.**CreateTableAs**(*name*, *query*)
Bases: sqlalchemy.sql.base.Executable, sqlalchemy.sql.elements.ClauseElement

class quetzal.app.helpers.sql.**DropSchemaIfExists**(*name*, *cascade=False*)
Bases: sqlalchemy.sql.base.Executable, sqlalchemy.sql.elements.ClauseElement

class quetzal.app.helpers.sql.**GrantUsageOnSchema**(*schema*, *user*)
Bases: sqlalchemy.sql.base.Executable, sqlalchemy.sql.elements.ClauseElement

quetzal.app.helpers.sql.**print_sql**(*qs*)

quetzal.app.middleware package

Submodules

quetzal.app.middleware.debug module

quetzal.app.middleware.debug.**debug_request**()

quetzal.app.middleware.debug.**debug_response**(*response*)

quetzal.app.middleware.gdpr module

quetzal.app.middleware.gdpr.**gdpr_log_request**()

quetzal.app.middleware.headers module

class quetzal.app.middleware.headers.**HttpHostHeaderMiddleware**(*app*, *server=None*)
Bases: object

quetzal.app.redoc package

Submodules

quetzal.app.redoc.routes module

quetzal.app.redoc.routes.**redoc**()

Submodules

quetzal.app.background module

Background tasks

quetzal.app.background.**backup_logs**(*app*)

quetzal.app.background.**hello**()

quetzal.app.hacks module

Hacks needed to circumvent connexion validation

There is a bug on the connexion library concerning content negotiation and response validation. See https://github.com/zalando/connexion/issues/860

Until this issue is fixed, we need to find a way to avoid a false validation error when a requests sends an 'application/octet-stream' accept header when downloading files

class quetzal.app.hacks.**CustomResponseValidator**(*operation*, *mimetype*, *validator=None*)
Bases: connexion.decorators.response.ResponseValidator

validate_response_with_request(*request*, *data*, *status_code*, *headers*, *url*)

quetzal.app.models module

class quetzal.app.models.**ApiKey**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Model

class quetzal.app.models.**BaseMetadataKeys**
Bases: enum.Enum

Set of metadata keys that exist in the base metadata family

The base metadata family is completely managed by Quetzal; a user cannot set or change its values (with the exception of the value for the *path* or *filename* keys). This enumeration defines the set of keys that exist in this family.

CHECKSUM = 'checksum'
MD5 checksum of the file

DATE = 'date'
Date when this file was created.

FILENAME = 'filename'
Filename, without its path component.

ID = 'id'
Unique file identifier.

PATH = 'path'
Path component of the filename.

SIZE = 'size'
Size in bytes of the file.

STATE = 'state'
State of the file; see FileState.

URL = 'url'
URL where this file is stored.

class quetzal.app.models.**Family**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Model

Quetzal metadata family

In quetzal, metadata are organized in semantic groups that have a name and a version number. This is the definition of a metadata _family_. This class represents this definition. It is attached to a workspace, until the workspace is committed: at this point the family will be disassociated from the workspace to become *global* (available as public information).

id
Identifier and primary key of a family.

Type int

name
Name of the family.

Type str

version
Version of the family. Can be None during a workspace creation, and until its initialization, to express the *latest* available version.

Type int

description
Human-readable description of the family and its contents, documentation, and any other useful comment.

Type str

fk_workspace_id
Reference to the workspace that uses this family. When None, it means that this family and all its associated metadata is public.

Type int

Extra attributes

metadata_set All Metadata entries associated to this family.

increment()
Create a new family with the same name but next version number

The new family will be associated to the same workspace.

class quetzal.app.models.**FileState**
Bases: enum.Enum

State of a Quetzal file

Quetzal files have a status, saved in their *base* metadata under the *state* key. It can only have the values defined in this enumeration.

DELETED = 'deleted'
File has been deleted.

Deleted files will have their metadata cleared when the workspace is committed.

If it was an already committed file, its contents will not be removed from the global data storage directory or bucket, but its metadata will be cleared. If it was a file that was not committed yet, it will be erased from its workspace data directory or bucket.

Deleted files are not considered in queries.

READY = 'ready'
File is ready

It has been uploaded, it can be downloaded, its metadata can be changed and when its workspace is committed, it will be moved to the global data storage directory or bucket.

TEMPORARY = 'temporary'
File is ready but temporary

Like `READY`, but this file will not be considered when the workspace is committed. That is, it will not be copied to the global data storage directory or bucket.

class quetzal.app.models.**Metadata**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Model

Quetzal unstructured metadata

Quetzal defines metadata as a dictionary associated with a family. Families define the semantic organization and versioning of metadata, while this class gathers all the metadata key and values in a dictionary, represented as a JSON object.

id
Identifier and primary key of a metadata entry.

Type int

id_file
Unique identifier of a file as a UUID number version 4. This identifier is also present and must be the same as the *id* entry in the *json* member.

Type uuid.UUID

json
A json representation of metadata. Keys are metadata names and values are the related values. It may be a nested object if needed.

Type dict

Extra attributes

family The related `Family` associated to this metadata.

static get_latest(*file_id*, *family*)
Retrieve the latest metadata of a file under a particular family

static get_latest_global(*file_id=None*, *family_name=None*)
Retrieve the latest metadata of a file under a particular family

to_dict()
Return a dictionary representation of the metadata

Used to conform to the metadata details object on the OpenAPI specification.

Returns Dictionary representation of this object.

Return type dict

update(*json*)
Update the underlying json metadata with the values of a new one

This function takes the current json saved in this metadata object and updates it (like `dict.update`) with the new values found in the *json* input parameter. This does not remove any key; it adds new keys or changes any existing one.

Since SQLAlchemy does not detect changes on a JSONB column unless a new object is assigned to it, this function creates a new dictionary and replaces the previous one.

Changes still need to be committed through a DB session object.

Parameters **json** (*dict*) – A new metadata object that will update over the existing one

Returns

Return type self

class quetzal.app.models.**MetadataQuery**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Model

Query for metadata on Quetzal

Queries on Quetzal are temporarily saved as objects. This was initially thought as a mechanism for easier and faster paginations, to avoid verifying that a query is valid every time and possibly to compile these queries if needed.

id
Identifier and primary key of a query.

Type int

dialect
Dialect used on this query.

Type QueryDialect

code
String representation of the query. May change in the future.

Type str

fk_workspace_id
Reference to the `Workspace` where this query is applied. If `None`, the query is applied on the global, committed metadata.

Type int

fk_user_id
Reference to the `User` who created this query.

Type int

static get_or_404(*qid*)
Get a workspace by id or raise an APIException

static get_or_create(*dialect*, *code*, *workspace*, *owner*)
Retrieve a query by its fields or create a new one

to_dict(*results=None*)
Create a dict representation of the query and its results

Used to conform to the OpenAPI specification of the paginable query results

Parameters **results** (*dict*) – Results as a paginable object.

Returns Dictionary representation of this object.

Return type dict

class quetzal.app.models.**QueryDialect**
Bases: enum.Enum

Query dialects supported by Quetzal

class quetzal.app.models.**Role**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Model

Authorization management role on Quetzal

Quetzal operations are protected by an authorization system based on roles. A user may have one to many roles; a role defines what operations the associated users can do.

Note that the *n to n* relationship of roles and users is implemented through the `roles_users_table`.

id
Identifier and primary key of a role.

Type int

name
Unique name of the role.

Type str

description
Human-readable description of the role.

Type str

Extra attributes

users Set of users associated with this role. This attribute is defined through a backref in `User`.

class quetzal.app.models.**User**(***kwargs*)
Bases: flask_login.mixins.UserMixin, sqlalchemy.ext.declarative.api.Model

Quetzal user

Almost all operations on Quetzal can only be done with an authenticated user. This model defines the internal information that Quetzal needs for bookeeping its users, permissions, emails, etc.

id
Identifier and primary key of a user.

Type int

username
Unique string identifier of a user (e.g. admin, alice, bob).

Type str

email
Unique e-mail address of a user.

Type str

password_hash
Internal representation of the user password with salt.

Type str

token
Unique, temporary authorization token.

Type str

token_expiration
Expiration date of autorization token.

Type datetime

active
Whether this user is active (and consequently can perform operations) or not.

Type bool

Extra attributes

roles Set of `Roles` associated with this user.

workspaces Set of `Workspaces` owned by this user.

queries Set of `Queries` created by this user.

check_password(*password*)
Check if a password is correct.

Parameters **password** (*str*) – The password to verify against the hash-salted stored password.

Returns `True` when the provided password matches the hash-salted stored one.

Return type bool

static check_token(*token*)
Retrieve a user by token

No user will be returned when the token is expired or does not exist.

Parameters **token** (*str*) – Authorization token.

Returns **user** – User with the provided token, or `None` when either the token was not found or it was expired.

Return type `User`

get_token(*expires_in=3600*)
Create or retrieve an authorization token

When a user already has an authorization token, it returns it.

If there is no authorization token or the existing authorization token for this user is expired, this function will create a new one as a random string.

The changes on this instance are not propagated to the database (this must be done by the caller), but this instance added to the current database session.

Parameters **expires_in** (*int*) – Expiration time, in seconds from the current date, used when creating a new token.

Returns The authorization token

Return type str

property is_active
Property accessor for *active*.

Needed to conform to the flask_login.UserMixin interface.

revoke_token()
Revoke the authorization token

The changes on this instance are not propagated to the database (this must be done by the caller), but this instance added to the current database session.

set_password(*password*)
Change the password of this user.

This function set and store the new password as a salt-hashed string.

The changes on this instance are not propagated to the database (this must be done by the caller), but this instance added to the current database session.

Parameters **password** (*str*) – The new password.

class quetzal.app.models.**Workspace**(***kwargs*)
Bases: sqlalchemy.ext.declarative.api.Model

Quetzal workspace

In Quetzal, all operations on files and metadata are *sandboxed* in workspaces. Workspaces define the exact metadata families and versions, which in turn provides a snapshot of what files and metadata are available. This is the base of the reproducibility of dataset in Quetzal and the traceability of the data changes.

Workspaces also provide a storage directory or bucket where the user can upload new and temporary data files.

id
Identifier and primary key of a workspace.

Type int

name
Short name for a workspace. Unique together with the owner's username.

Type str

_state
State of the workspace. Do not use directly, use its property accessors.

Type WorkspaceState

description
Human-readable description of the workspace, its purpose, and any other useful comment.

Type str

creation_date
Date when the workspace was created.

Type datetime

temporary
When True, Quetzal will know that this workspace is intended for temporary operations and may be deleted automatically when not used for a while. When False, only its owner may delete it.

Type bool

data_url
URL to the data directory or bucket where new files associated to this workspace will be saved.

Type str

pg_schema_name
Used when creating structured views of the structured metadata, this schema name is the postgresql schema where temporary tables exists with a copy of the unstructured metadata.

Type str

fk_user_id
Owner of this workspace as a foreign key to a `User`.

Type int

fk_last_metadata_id
Reference to the most recent `Metadata` object that has been committed at the time when this workspace was created. This permits to have a reference to which global metadata entries should be taken into account when determining the metadata in this workspace.

Type int

Extra attributes

families Set of `Families` (including its version) used for this workspace.

queries Set of `Queries` created on this workspace.

property can_change_metadata
Returns `True` when metadata can be changed on the current workspace state

get_base_family()
Get the base family instance associated with this workspace

get_current_metadata()
Get the metadata that has been added or modified in this workspace

In contrast to `get_previous_metadata()`, this function only retrieves the metadata that has been changed on this workspace after its creation.

get_metadata()
Get a union of the previous and new metadata of this workspace

This function uses a combination of the results of `get_previous_metadata()` and `get_current_metadata()` to obtain the merged version of both. This represents the definitive metadata of each file, regardless of changes before or after the creation of this workspace.

static get_or_404(*wid*)
Get a workspace by id or raise a *quetzal.app.api.exceptions.ObjectNotFoundException*

get_previous_metadata()
Get the global metadata of this workspace

The global metadata is the metadata that already has been committed, but it must also have a version value that is under the values declared for this workspace.

make_schema_name()
Generate a unique schema name for its internal structured metadata views

property state
Property accessor for the workspace state

to_dict()
Return a dictionary representation of the workspace

This is used in particular to adhere to the OpenAPI specification of workspace details objects.

Returns Dictionary representation of this object.

Return type dict

class quetzal.app.models.**WorkspaceState**
Bases: enum.Enum

Status of a workspace.

Workspaces in Quetzal have a state that defines what operations can be performed on them. This addresses the need for long-running tasks that modify the workspace, such as initialization, committing, deleting, etc.

The transitions from one state to another is defined on this enumeration on the `transitions()` function. The following diagram illustrates the possible state transitions:

The verification of state transitions is implemented in the `quetzal.app.models.Workspace.state` property setter function.

COMMITTING = 'committing'
The workspace is committing its files and metadata.

The workpace will remain on this state until the committing routine finishes. No operation is possible until then.

CONFLICT = 'conflict'
The workspace detected a conflict during its commit routine.

The workpace will remain on this state until the administrator fixes this situation. No operation is possible.

DELETED = 'deleted'
The workspace has been deleted.

The instance of the workspace remains in database for bookeeping, but there is no operation possible with it at this point.

DELETING = 'deleting'
The workspace is deleting its files and itself.

The workpace will remain on this state until the deleting routine finishes. No operation is possible.

INITIALIZING = 'initializing'
The workspace has just been created.

The workspace will remain on this state until the initialization routine finishes. No operation is possible until then.

INVALID = 'invalid'
The workspace has encountered an unexpected error.

The workpace will remain on this state until the administrator fixes this situation. No operation is possible.

READY = 'ready'
The workspace is ready.

The workspace can now be scanned, updated, committed or deleted. Files can be uploaded to it and their metadata can be changed.

SCANNING = 'scanning'
The workspace is updating its internal views.

The workpace will remain on this state until the scanning routine finishes. No operation is possible until then.

UPDATING = 'updating'
The workspace is updating its metadata version definition.

The workpace will remain on this state until the updating routine finishes. No operation is possible until then.

quetzal.app.models.**roles_users_table = Table('roles_users', MetaData(bind=None), Column('fl**
Auxiliary table associating users and roles

quetzal.app.routes module

quetzal.app.routes.**favicon**()

quetzal.app.routes.**health**()

quetzal.app.routes.**index**()

quetzal.app.security module

class quetzal.app.security.**CommitWorkspacePermission**(*workspace_id*)
Bases: flask_principal.Permission

class quetzal.app.security.**ReadWorkspacePermission**(*workspace_id*)
Bases: flask_principal.Permission

quetzal.app.security.**WorkspaceNeed**
alias of quetzal.app.security.workspace_need

class quetzal.app.security.**WriteWorkspacePermission**(*workspace_id*)
Bases: flask_principal.Permission

quetzal.app.security.**load_identity**(*sender*, *identity*)

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

q

INDEX

Symbols

A

B

C

D

O

P

Q

R

S

T

U

V

W