Omicron Server Documentation

Release 0.1.1

Michal Kononenko, Chris Yoo, Shahab Akmal

February 22, 2016

Contents:

CHAPTER 1

Installation

1.1 Dependencies

Package dependencies are outlined in the "requirements.txt" folder. In order to install all dependencies, run

```
$ pip install -r requirements.txt
```

from the project's root directory.

1.1.1 Windows Users and Python Packages

The following packages may have trouble installing on Windows

- psycopg2

Due to the fact that some Python packages depend on compiled C code to run properly (Numpy. being a good example), users running Windows may get an error when installing the following packages. This error usually states that pip is unable to find a file named `vcvarsall.bat`. In order to resolve this, it is recommended that you go to the UC Irvine Repo. to find pre-compiled `.whl` binaries for these packages. Running `pip` on these wheels should install them on your machine. Keep in mind that these are **UNOFFICIAL** binaries, and are distributed **AS-IS**.

1.2 Running the Code

To run the code, run

```
$ python run_server.py
```

from the project's root directory. Flask includes a built-in threaded web server that will start up on `localhost:5000` by default. In order to change these parameters, set the `IP_ADDRESS` and `PORT` environment variables on your command line to the desired values

On Linux (specifically bash), this can be done using

```
$ EXPORT IP_ADDRESS=127.0.0.1
```

On Windows, this can be done using

```
> SET IP_ADDRESS=127.0.0.1
```

1.3 Running Tests

This project is tested using Python's built-in unittest. for writing unit tests. In order to run all tests, run

```
$ nosetests
```

from your command line. To run unit tests from a particular directory, `cd` into that directory, and run `nosetests` from it. `nosetests` determines what is a test by the following criteria

- The name of the method must start with `test_`
- **The class in which the method is located must inherit from** `unittest.TestCase` somewhere along its inheritance hierarchy
- The module in which the method is located must start with `test_`

CHAPTER 2

A Brief Concepts Summary

Here is a brief explanation of some of the concepts powering this API, and a discussion on what was intended to be achieved using certain technologies. If you are a web developer, or have experience writing web applications, feel free to skip over this chapter. The purpose of this chapter is to give newcomers a brief introduction into some of the terms and concepts used in general web development, and to ensure that we are all on the same page when we discuss this project.

If there is anything to which this document doesn't do justice, or you feel like information on an important concept is missing, that's a bug. Please report it here.

2.1 Representational State Transfer (REST)

2.1.1 The Role of an API

The goal of any Application Programming Interface (API) is to expose the functionality of a software component, defined as some coherent block of code, to the outside world in a way that makes sense. A good API will make it easy for programmers to take our code, and stich it together with other APIs in order to build applications.

The majority of the codebase of OmicronServer is devoted to presenting our code for managing projects, experiments, and users on the Omicron System to the outside world. In addition, it presents our data model to interested clients.

The problem becomes non-trivial as our API needs to be online, it needs to be language-agnostic (clients can be using several different programming languages), and we need to be able to control who has access to the data that our API manages. To solve this problem, we have chosen to implement a REST API

2.1.2 Enter REST

REST is an architectural style for writing APIs, where a client machine can interact and invoke functionality on a remote server. It requires the following six criteria

- It must follow a client-server model
- It must be stateless
- Responses must be cacheable
- The system must be layered
- Code should be available on demand
- The API should provide a uniform interface to users

As you can see, these requirements are *very* general. And of course, the combination of a nifty idea with vague standards results in a *lot* of arguing in the blogosphere, as can be seen here, here, here, here, and, well you get the point.

I recommend reading these articles, and listening to this talk by Miguel Grinberg from PyCon 2015 where he discusses his take on REST, and also by joining the discussion on the GitHub Repository to make our API more RESTFul.

2.1.3 HTTP and REST: Why They Go So Well Together

REST doesn't require that you use a particular protocol to communicate with other machines, but the Hypertext Transfer Protocol (HTTP) has become the ideal protocol on which most well-known APIs are based. This is because in addition to its ubiquity as the protocol of the World Wide Web, HTTP helps create a uniform interface for our API by packaging all API calls in requests and packaging all return values in responses.

HTTP requests consist of the following

- **A web address and a port to which the request is being sent**
  - **If a text web address is specified, then the Domain Name Service** (DNS) is used to resolve the address behind the covers
  - **The port to which the request is being sent, as an integer from** 0 to 65535.
  - **The HTTP Method to be executed on the address. `GET` is a good** example
- **A request header containing a set of key-value pairs containing request** metadata (data about the request), and data required to make the request that doesn't involve the actual request data. This data is returned if a `HEAD` request is made.
  - **An example of a header is `content-type: application/json`, which** specifies that the response should be returned in the JSON format, as opposed to, for example, XML
  - **Another important header that you will encounter is** `Authorization`, which contains authentication information
- **A request body consisting of the data to be sent to the browser. The** format of the body is specified by the `content-type` header.

An HTTP response contains all these as well, but it also contains a status code, communicating whether the message succeeded or failed, and in what way. This is why websites will display `404` if you go to a non-existent page on their sites.

So what does this have to do with REST? Think of a REST API as a very simple web site, without any of the buttons, styling, forms, or anything that would disrupt the machine readability of the site. There is only text. If you want to interact with the site, you as the programmer will have to know to where you have to send your request, and, in the case of `POST` and `PATCH` requests, what is expected of you in the request body. It may sound like a lot, but the advantage of using HTTP is that libraries exist in a multitude of programming languages for making requests, and for parsing JSON. Also, it provides you with a fantastic tool for debugging REST APIs, your web browser

Note: For Chrome users, I recommend getting the Postman extension, which provides an easy-to-use interface for making requests to REST APIs.

2.2 Git and GitHub

The OmicronServer GitHub repository is the single source of truth for working production-ready code in this project. If your code isn't in version control, it doesn't exist. There's no way to track code changes, no way to merge your code with working code in a transactional way, no easy way to see what you want to change in the project codebase, and you're one disk failure away from losing everything. Version control solves all these problems, and, for the purposes of this project, git solves them the best.

2.2.1 Git: Distributed Version Control

Initially developed by Linus Torvalds in 2005 for Linux kernel development, git is a free and open-source version control system optimized for non-linear development workflows and synchronization of code across multiple repositories. Your local copy of a git repository is treated no differently from a git repository on a remote server, i.e. GitHub. This makes it a very powerful program for version control, but also presents a bit of a learning curve for newcomers.

2.2.2 GitHub: A Hub For Git

GitHub is a super-awesome website that works both as a central repository for git projects, and provides web apps for simple project management. It is by far the largest code host in the world for open-source projects, and is nice enough to host open-source projects for free. In addition to providing all the utilities of git in a nice GUI, it also offers some nice features including

- **Pull Requests: When you feel your branch is ready to be merged into** `master` (let's say you fixed the bug you were trying to fix), you can open up a pull request, which is a way for you to ask the owner of a repo to pull one of your branches (hence the name) and merge it into their `master` branch. This opens up a lovely window where you can see line-for-line exactly what they intend to change, offers an opportunity for TravisCI to check your code, and lets contributors comment on your code. They can even reference the lines they think can be improved. Pull requests, therefore, serve an important role as the place where code review occurs.
- **Issues: GitHub also lets you track issues with code. These aren't** just bug reports, but can also be enhancements, questions to ask the contributors, or any discussion thread that you feel is relevant to the code in this repository. Issues can be opened, closed (indicating that they are solved), assigned to people, and referenced in other issues and pull requests, making them powerful tools for project management and request specifications. If you want to see a particular feature in this code, or if you would like to report a bug, please open an issue here.
- **Milestones: Milestones are nothing more than collections of issues** that may be attached to a particular due date. This lets us set project deadlines, and establish project scope for releases. Milestones also come with a nifty percentage bar that lets contributors know how far along work has progressed towards meeting a particular milestone. This is how project scope will be tracked, at least for now.

2.3 Relational Databases and You

REST APIs need to be stateless, meaning that after a request is processed, the application must return to the same state that it was in before the request started. We can't, for example, open up a session for a user when they authenticate, store the fact that the session was opened, and close the session when the user logs out. It also means that, at least in production, we can't store our data in our app. So where can we put this data if not in the app? The answer: a database.

Cuz you know I'm all about that 'base, database, no treble!

- Meghan Trainor if she was a web developer

In production, the API will be pulling its data from PostgreSQL. This is a relational database, meaning that it stores data as a bunch of cross-indexed tables. This gives us the following benefits

- **Relations do not set either of their partners as first-class citizens.** Do projects belong to users or do users belong to projects? Relational databases don't care.

- **Relational databases can enforce constraints on our data by mandating** that a value in a column must match one of the values in another table, or that a value in a column must be unique. This prevents us from, for example, creating a project without a user as its owner.
- **Relational databases are transactional, meaning any update or delete** operations can be done in such a way that the database always moves from one allowed state to another allowed state.

CHAPTER 3

Technologies Used in This Project

This is a brief rundown of key third-party libraries and programs that this API depends on. Each system is summarized below, a guide is provided on how to use it for this project, and useful links are also provided for additional resources.

3.1 Libraries

3.1.1 Flask

Flask bills itself as a "micro-framework" for web development in Python. Essentially, it is a set of functions and objects used by the code here to interact with Werkzeug, processing HTTP requests and returning HTTP responses. You give Flask a URL, and Flask maps those HTTP requests into function calls.

Compared to other Python web frameworks, notably Django, Flask is a lot more flexible to use, and behaves more like a library than a framework. For instance, a simple "Hello World" server clocks in at 7 lines of Python code.

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World'

if __name__ == '__main__':
    app.run()
```

This program will start a server, and return 'Hello World' if a `GET` request is sent to `localhost:5000`. Note the use of the decorator to decorate our route, and the fact that the `run` method is inside an `if __name__ == '__main__'` block.

Flask defines routes in terms of decorators, but for this API, we mixed this up a little bit with the next library on our list.

For more information, I would like to refer you to Miguel Grinberg's excellent 18-part Flask Mega-Tutorial on what Flask is capable of, and to some of his other posts on REST APIs, including

- Designing a RESTful API in Flask
- RESTful Authentication with Flask
- Designing a RESTful API using Flask-RESTful

3.1.2 Flask-RESTful

Flask RESTFul is an extension to Flask that provides a very useful abstraction to vanilla Flask by letting us use classes instead of functions in order to write endpoints. In order to implement it, the following is required

- The class must inherit from `flask_restful.Resource`
- **An object of type `flask_restful.Api` must be created and bound** to an object of type `flask.Flask`
- **The class must be registered with the API using** `flask_restful.Api.add_resource()`

What does this offer us?

- **Each subclass of `flask_restful.Resource` has methods available** corresponding to each HTTP verb (`GET`, `POST`, `PUT`, etc.). Endpoints are now mapped to objects in a one-to-one relationship
- **Code that helps the object process HTTP requests now travels with the** request-processing business logic, if it's not being used anywhere else. This provides some nice separation of concerns.
- **Our entire API can now have a common URL prefix, and all our resources** are registered in one place, creating a pseudo-routing table in our code. Neat!

An example of a simple REST API created with Flask RESTFul is given below

```
from flask import Flask
from flask_restful import Api, Resource

app = Flask(__name__)
api = Api(app, prefix='api/v1')

class HelloWorld(Resource):

    def get():
        return 'Hello World'

    def post():
        # Post handling logic here
        return 'Request posted!'

api.add_resource(HelloWorld, '/hello', endpoint='hello)

if __name__ == '__main__':
    app.run()
```

I think you'll agree that this is a lot lighter-weight than something like DjangoRESTFramework

3.1.3 SQLAlchemy

R

3.2 Git and GitHub

3.2.1 Git: Distributed Version Control

The following section provides a brief introduction to some git commands that you may need to use. For a more comprehensive tutorial, check out the official documentation for git here.

- **`git clone <url>` will take a repository located at `<url>`, create** a new folder to hold your code, and will download the `master` branch of the repository into this new folder.
- **`git branch <name>` will create a new branch named `<name>` in your** local repository. Git branches are a series of snapshots of your code repository that show some linear development of code. For example, I could create a branch called "Issue14BugFix", and I would develop in this branch anything that I needed to fix whatever "Issue 14" is. Then, when I felt the issue was fixed, I would merge this branch with the "master" branch.

Calling just `git branch` will give you a list of local git branches, and will highlight the branch you are currently in.

Note: The `master` branch is special. It is the first branch created with each repository, and represents the main line of development in the code. Only users with write access to a repository can write to the `master` branch, and it is generally considered bad practice to develop in `master`. Use `git branch` to make a new branch and develop in that instead.

- **`git checkout <branch>` is how you switch between git branches. Any** file tracked by git will be changed to the version stored in that branch. Any file not tracked by git will stay the way it is.
- **`git add <file>` is how you add files to git to be tracked. If you** specify a directory, `git add` will be run recursively over every file and subdirectory in the directory. `git add .` is a common command, and will add every file to git in your current directory, provided you are in a git repository (there is a `.git` folder somewhere between your current directory and your system root directory). Git will not add any file whose name matches a string in the `.gitignore` file. This is by design. I will have more to say about this later, but `.gitignore`'s job is to tell git what files are not source code so that they aren't tracked.
- **`git commit` is the most important command. When you hit the save** button on a file, you write it to disk, but git can overwrite saved files if you `git checkout` to another branch. `git commit` writes your code changes to a branch. Branches are nothing more than sets of `git commit``s. ``git commit` is atomic, meaning that either all the changed files are committed, or none of the changed files are committed. adding an `-a` to `git commit` will make git run `git add .` before committing, and adding an `-m <message>` flag will allow you to insert a brief (up to 80 characters) message stating what you did in the commit. (Relevant XKCD). If you don't add a message in the command, you will be prompted for a message.

The following is a list of other useful git commands, expect documentation for them soon

- `git diff`
- `git rm`
- `git remote add`
- `git push`
- `git pull`
- `git merge`

3.3 Travis CI

Travis CI, short for Continuous Integration, is a hosted web service that has only one job, take any code changes from GitHub, build the changes, run the tests, and report on whether the tests passed or not. The advantage of this is that since tests are run remotely on a “clean” server, we can get some confidence that the server will run in any environment. Furthermore, the server builds all pull requests, and so it lets us catch any test failures during the pull request.

The build parameters for Travis are described in the `.travis.yml` file.

More information about Travis can be found **here_**.

3.4 Coveralls

Coveralls, like Travis CI, is a free service for open source projects, except instead of running tests to see if they pass, this service measures code coverage. This is the percentage of lines of code in the project that are actually hit during testing. This should be as high as possible.

3.5 Heroku

Heroku will serve as the development machine for this project. The server is hosted at omicronserver.herokuapp.com. This machine will pull the master branch from GitHub and deploy it using instructions given to it in `Procfile`. For more information, check out Heroku’s Documentation.

Warning: The dev machine on Heroku will shut down after 30 minutes of inactivity, and can only be up for a combined total of 18 hours per day, as part of the free usage tier for Heroku hosting. This machine should not be used for production, or any serious performance testing.

3.6 SQLAlchemy

As awesome as relational databases are, there is a problem. There are cases where relations and objects simply don’t get along as data structures. To quote SQLAlchemy’s Documentation

> SQL databases behave less like object collections the more size and performance start to matter; object collections behave less like tables and rows the more abstraction starts to matter.

SQLAlchemy is not the solution to the object-relational impedance_mismatch, but it tries to alleviate some problems associated with working with relational databases in object-oriented programming languages. Namely

- **For small queries (i.e. queries returning little data**), SQLAlchemy can take the data returned from executing SQL code against the database (a blob of tables and columns), and map those return values to a list of objects of the type specified by the SQLAlchemy query. For instance

```
with sessionmaker() as session:
    session.query(User).all()
```

  will return a list of objects of type `User`, and allow further processing.

CHAPTER 4

Web API

4.1 Introduction

Done properly, end users will never have to interact with this project at all! The functionality in this server will be exposed through an HTTP REST API. This module provides documentation for all the HTTP requests that will be handled by this API (application programming interface).

4.2 URL Endpoints

POST /api/v1/projects

Create a new project

Example Request

```
HTTP/1.1
Content-Type: application/json

{
    "name": "NMR Project",
    "description": "NMR Project Description",
    "owner": "mkononen"
}
```

Example Response

```
HTTP/1.1 201 CREATED
Content-Type: application/json

{
    "name": "NMR Project",
    "description": "NMR Project Description",
    "owner": {
        "username": "mkononen",
        "email": "my@email.com"
    }
}
```

Status Codes

- 201 Created – Project successfully created

- 400 Bad Request – Unable to create project due to malformed JSON

Parameters

- **session** (*ContextManagedSession*) – The database session to be used for making the request

Return A Flask response

Rtype flask.Response

GET /api/v1/projects

Returns the list of projects accessible to the user

Example Response

```
HTTP/1.1 200 OK
Content-type: application/json
page: 1
items_per_page: 1
Count: 1

{
    "projects": [
        {
            "name": "NMR Project",
            "description": "This is a project",
            "date_created": "2015-01-01T12:00:00",
            "owner": {
                "username": "mkononen"
                "email": "my@email.com"
            }
        }
    ]
}
```

Status Codes

- 200 OK – Request completed without errors
- 400 Bad Request – Bad request, occurred due to malformed JSON or due to the fact that the user was not found

Parameters

- **pag_args** (*PaginationArgs*) – A named tuple injected into the function's arguments by the `@restful_pagination()` decorator, containing parsed parameters for pagination

Return A flask response object containing the required data to be displayed

POST /api/v1/users

Create a new user

Example Request

```
HTTP/1.1
Content-Type: application/json

{
    "username": "scott",
    "password": "tiger",
    "email": "scott@tiger.com"
}
```

Example Response

```
HTTP/1.1 201 CREATED
Content-Type: application/json

{
    "username": "scott",
    "email": "scott@tiger.com
}
```

Status Codes

- 201 Created – The new user has been created successfully
- 400 Bad Request – The request could not be completed due to poor JSON
- 422 Unprocessable Entity – The request was correct, but the user could not be created due to the fact that a user with that username already exists

Parameters

- **`session`** (*`Session`*) – The database session that will be used to make the request

`GET /api/v1/users`
Process a GET request for the /users endpoint

Example request

```
GET /api/v1/users HTTP/1.1
Host: example.com
Content-Type: application/json
```

Example response

```
Vary: Accept
Content-Type: application/json

.. include:: /schemas/users/examples/get.json
```

Query Parameters

- **`contains`** – A string specifying what substring should be contained in the username. Default is `''`
- **`starts_with`** – The substring that the username should start with. Default is `''`.
- **`page`** – The number of the page to be displayed. Defaults to `1`.
- **`items_per_page`** – The number of items to be displayed on this page of the results. Default is `1000` items.

Status Codes

- 200 OK – no error

Parameters

- **`session`** (*`Session`*) – The database session used to make the request

- **pag_args** (*PaginationArgs*) – The pagination arguments generated by the `@restful_pagination()` decorator, injected into the function at runtime

Return A flask response object with the search request parsed

POST /api/v1/token

Generate a user's auth token from the user in Flask's `Flask.g` object, which acts as an object repository unique to each request. Expects an Authorization header with Basic Auth.

Example Request

```
POST /api/v1/token HTTP/1.1
Host: example.com
Content-Type: application/json
Authorization: B12kS1l2jS1=
```

Example Response

```
Content-Type: application/json

{
    "token": "a409a362-d733-11e5-b625-7e14f79230d0",
    "expiration_date": "2015-01-01T12:00:00"
}
```

Return A Flask response object with the token jsonified into ASCII

DELETE /api/v1/token

Revoke the current token for the user that has just authenticated, or the user with username given by a query parameter, allowed only if the user is an Administrator

GET /index

GET /

Base URL to confirm that the API actually works. Eventually, this endpoint will serve the OmicronClient. JavaScript UI to users.

Example Response

```
GET / HTTP/1.1
Content-Type: application/json

Hello World!
```

Return Hello, world!

Rtype str

GET /api/v1/users/(*username_or_id*)

Returns information for a given user

Example Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "username": "scott"
    "email": "scott@tiger.com"
    "projects": [
        {
            "name": "NMR Experiment 1",
            "description": "Measure this thing in NMR",
            "date_created": "2015-01-01T12:00:00Z",
            "owner": {
                "username": "scott",
                "email": "scott@tiger.com"
            }
        }
    ]
}
```

Parameters

- **session** (*Session*) – The database session to be used in the endpoint
- **or str username_or_id** (*int*) – The username or user_id of the user for which data needs to be retrieved

DELETE /api/v1projects/(*project_name_or_id*)

Delete a project

Status Codes

- 200 OK – The project was deleted successfully
- 404 Not Found – Unable to find the project to delete

Parameters

- **or str project_name_or_id** (*int*) – The project to delete

GET /api/v1projects/(*project_name_or_id*)

Returns the details for a given project

Example Response

```
HTTP/1.1 200 OK

{
    "name": "NMR Project",
    "description": "NMR Project Description",
    "owner": {
        "username": "mkononen",
        "email": "my@email.com"
    }
}
```

Status Codes

- 200 OK – The Request completed successfully
- 404 Not Found – The request project could not be found

Parameters

- **or str project_name_or_id** (*int*) – The id or name of the project to retrieve

Return A flask response object containing the required data

Rtype `flask.Response`

GET /static/(**path:** *filename*)

Function used internally to send static files from the static folder to the browser.

New in version 0.5.

API Reference

5.1 Contents

5.1.1 API Documentation

Contains documentation pulled from all modules in Omicron Server **api_server**

Defines the flask app which will run our HTTP application. This also creates a flask-restful API object, which will serve as the router to the objects in `api_views`.

`api_server.`**`create_token`**`(`**args*, ***kwargs*`)`

Generate a user's auth token from the user in Flask's `Flask.g` object, which acts as an object repository unique to each request. Expects an Authorization header with Basic Auth.

Example Request

```
POST /api/v1/token HTTP/1.1
Host: example.com
Content-Type: application/json
Authorization: B12kS1l2jS1=
```

Example Response

```
Content-Type: application/json

{
    "token": "a409a362-d733-11e5-b625-7e14f79230d0",
    "expiration_date": "2015-01-01T12:00:00"
}
```

Returns A Flask response object with the token jsonified into ASCII

`api_server.`**`hello_world`**`()`

Base URL to confirm that the API actually works. Eventually, this endpoint will serve the OmicronClient. JavaScript UI to users.

Example Response

```
GET / HTTP/1.1
Content-Type: application/json
```

```
        Hello World!
```

Returns Hello, world!
Return type str

`api_server.`**`revoke_token`**(**args*, ***kwargs*)
Revoke the current token for the user that has just authenticated, or the user with username given by a query parameter, allowed only if the user is an Administrator

authentication

Contains declarations for HTTP authorization, needed to be put here to avoid a circular import where *`api_server`* imports `api_views.users.UserContainer`, but `api_views.users.UserContaner` requires an `auth` object declared in *`api_server`*

`auth.`**`_verify_user`**(*username*, *password*)
Check if the username matches the user. If it does, write the user to `Flask.g()` and return `True`, else return `False` :param str username: The name of the user to validate :param str password: The password to validate :return: `True` if the user authenticated and `False` if not

`auth.`**`verify_password`**(*username_or_token*, *password=None*)
Callback function for `flask.ext.httpauth.HTTPBasicAuth.verify_password()`, used to verify both username and password authentication, as well as authentication tokens.

In order to authenticate, the user must use base64 encoding, encode a string of the form `username:password`, and submit the encoded string in the request's Authorization. header.

Alternatively, the user can encode their token in base64, and submit this in their Authorization. header. In this case, the incoming password will be `None`.

Warning: Basic Authentication, unless done over SSL. **IS NOT A SECURE FORM** ** OF AUTHENTICATION**, as **ANYONE** can intercept an HTTP request, and decode the information in the Authorization header. This will be solved in two ways

- Any production deployment of this API will be done using SSL
- **HMAC-SHA256. authentication will be supported, although this is**
  currently out of scope for the Christmas Release of this API

Parameters

- **`username_or_token`** (*`str`*) – The username or token of the user attempting to authenticate into the API
- **`password`** (*`str`*) – The password or token to be used to authenticate into the API. If no password is supplied, this value will be `None`.

Returns True if the password or token is correct, False if otherwise
Rtype bool

config

Contains global config parameters for the API

class `config.`**`Config`**(*conf_dict={'READTHEDOCS_PROJECT': 'omicron-server', 'READTHEDOCS': 'True', 'APPDIR': '/app', 'DEBIAN_FRONTEND': 'noninteractive', 'OLDPWD': '/', 'HOSTNAME': 'version-latest-of-omicronserver-1769447', 'PWD': '/home/docs/checkouts/readthedocs.org/user_builds/omicron-server/checkouts/latest/docs/source', 'BIN_PATH': '/home/docs/checkouts/readthedocs.org/user_builds/omicron-server/envs/latest/bin', 'READTHEDOCS_VERSION': 'latest', 'PATH': '/home/docs/checkouts/readthedocs.org/user_builds/omicron-server/envs/latest/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin', 'HOME': '/home/docs'}*)

Contains configuration parameters and methods for receiving values from environment variables.

For global configuration, system environment variables have priority, followed by values in this object, then followed by the value in a component's configuration file.

`__init__`(*conf_dict={'READTHEDOCS_PROJECT': 'omicron-server', 'READTHEDOCS': 'True', 'APPDIR': '/app', 'DEBIAN_FRONTEND': 'noninteractive', 'OLDPWD': '/', 'HOSTNAME': 'version-latest-of-omicronserver-1769447', 'PWD': '/home/docs/checkouts/readthedocs.org/user_builds/omicron-server/checkouts/latest/docs/source', 'BIN_PATH': '/home/docs/checkouts/readthedocs.org/user_builds/omicron-server/envs/latest/bin', 'READTHEDOCS_VERSION': 'latest', 'PATH': '/home/docs/checkouts/readthedocs.org/user_builds/omicron-server/envs/latest/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin', 'HOME': '/home/docs'}*)

Instantiate config parameters from environment variables

Parameters **`conf_dict`** – The dictionary or dictionary-like object from which configuration parameters should be pulled, defaults to `os.environ`. This is overwritten for testing

`__weakref__`

list of weak references to the object (if defined)

decorators

Contains utilities to decorate endpoints with additional functionality

Decorators

A decorator is a function that takes in a function as an argument, and returns a function. In Python, a typical decorator can be used as follows

```python
class UserContainer(Resource):
    @staticmethod
    def parse_search_params(params):
        pass

    @restful_pagination()
    def get(self, pag_args):
        pass
```

The use of the `@` symbol is syntactic sugar for

```
    parse_search_params = staticmethod(parse_search_params)
```

In order to provide arguments to the decorator, as in the case of `@restful_pagination`, another level of indirection is needed. `restful_pagination` is not a decorator in and of itself, but it returns a decorator that then decorates a particular function.

Note: Due to *Sphinx Autodoc <http://sphinx-doc.org/ext/autodoc.html>*'s documentation generator, the `__name__` and `__doc__` property of the function to be decorated must be assigned to the `__name__` and `__doc__` of the decorator. See `tests.unit.test_decorators.TestRestfulPagination` for an example of a unit test that tests this behaviour

class `decorators.`**`PaginationArgs`**(*page*, *items_per_page*, *offset*)

`__getnewargs__`()
Return self as a plain tuple. Used by copy and pickle.

`__getstate__`()
Exclude the OrderedDict from pickling

static `__new__`(*_cls*, *page*, *items_per_page*, *offset*)
Create new instance of PaginationArgs(page, items_per_page, offset)

`__repr__`()
Return a nicely formatted representation string

`_asdict`()
Return a new OrderedDict which maps field names to their values

classmethod `_make`(*iterable*, *new=<built-in method __new__ of type object at 0x9192c0>*, *len=<built-in function len>*)
Make a new PaginationArgs object from a sequence or iterable

`_replace`(*_self*, ***kwds*)
Return a new PaginationArgs object replacing specified fields with new values

`items_per_page`
Alias for field number 1

`offset`
Alias for field number 2

`page`
Alias for field number 0

`decorators.`**`restful_pagination`**(*default_items_per_page=1000*)
Wraps a RESTful getter, extracting the arguments `page` and `items_per_page` from the URL query parameter. These arguments are then parsed into integers, and returned as a named tuple. consisting of the following variables

- `page`: The number of the page to be displayed
- **`items_per_page`: The number of items to be displayed on each** page
- **`offset`: The offset (item number of the first item on this** page)

The *PaginationArgs* tuple is then injected into the decorated function as a keyword argument (kwarg.), in a similar way to the `@mock.patch` decorator. A usage pattern for this decorator could be

```
        @restful_pagination()
        def paginated_input(pagination_args):
            with sessionmaker() as session:
                session.query(Something).limit(
                    pagination_args.limit
                ).offset(
                    pagination_args.offset
                ).all()
```

Parameters **`default_items_per_page`** – The number of items that should be displayed by default. This is to prevent a query with, for example, 300,000 results loading a large amount of data into memory.

Returns A decorator with the default arguments per page configured

Return type function

JSON Schema Parser

Contains utilities for loading, parsing, and validating inputs against a JSON schema

exception `json_schema_parser.`**`BadJsonSchemaError`**
Thrown if the JSON Schema presented is not a valid draft 3 or draft 4 schema

`__weakref__`
list of weak references to the object (if defined)

exception `json_schema_parser.`**`FileNotFoundError`**
Thrown if the parser is unable to find the file

`__weakref__`
list of weak references to the object (if defined)

class `json_schema_parser.`**`JsonSchemaValidator`**(*path*)
Contains utilities for validating JSONSchema from a file

`__init__`(*path*)
Initialize the schema :param path: the path from which the file should be pulled

`__weakref__`
list of weak references to the object (if defined)

static **`_read_schema_from_file`**(*path*)
Read the file containing the JSON Schema and return it as a json-loaded dictionary :return:

`validate_dict`(*dict_to_validate*)
Validates a dictionary against `self.json_dict`. :param dict dict_to_validate: The dictionary to check against `self.json_dict`, representing the base JSON Schema to validate.

Returns True if the dictionary is allowed by the schema and false if not. The False return also returns a string showing the reason why validation failed.

5.1.2 Database API

Contains documentation for `database`, which contains all the model classes for allowing the API to interact with the database

Database Models

Contains documentation for all database model classes **Projects**

Contains model classes related to managing projects on the Omicron server

class `database.models.projects.`**`Project`**(*project_name*, *description*, *date_created=datetime.datetime(2016*, *2*, *22*, *2*, *44*, *22*, *609803)*, *owner=None*)

Models a project for a given user. The project constructor requires the following

Variables

- **`project_name`** (*`str`*) – The name of the project
- **`description`** (*`str`*) – A description of the project
- **`date_created`** (*`datetime.datetime`*) – The date at which the project was created, defaults to the current datetime
- **`owner`** (`User`) – The user who owns this project

`__init__`(*project_name*, *description*, *date_created=datetime.datetime(2016*, *2*, *22*, *2*, *44*, *22*, *609803)*, *owner=None*)

Instantiates the variables described above

`date_created_isoformat`

Returns the project date created as an ISO 8601 - compliant string :return: The date string :rtype: str

`get`

Returns a dictionary representing a summary of the project. :return: A summary of the project :rtype: dict

Users

Contains all model classes relevant to management of users

class `database.models.users.`**`Administrator`**(*username*, *password*, *email*, *date_created=datetime.datetime(2016*, *2*, *22*, *2*, *44*, *22*, *613849)*)

Represents a "superuser" type. The administrator will be able to oversee all projects, revoke anyone's token, and approve new users into the system, when user approval is complete.

class `database.models.users.`**`Token`**(*token_string*, *expiration_date=None*, *owner=None*)

Contains methods for manipulating user tokens.

static **`hash_token`**(*token_string*)

Takes a token string and hashes it using SHA256.

`revoke`()

Expire the token by setting the expiration date equal to the current date

`verify_token`(*token*)

Checks if the provided token string matches the token hash in the DB, and that the token is not expired :param str token: The token to verify :return: True if the token is valid, False if not

class `database.models.users.`**`User`**(*username*, *password*, *email*, *date_created=datetime.datetime(2016*, *2*, *22*, *2*, *44*, *22*, *613849)*)

Base class for a User.

`__eq__`(*other*)

Parameters **`other`** (`User`) – The user against which to compare

`__ne__`(*other*)

Check if two users are not equal :param other:

classmethod `from_session`(*username*, *session*)

Provides an alternate "constructor" by using the supplied session and returning the user matching the given username. The method also asserts that a user was returned by the query. If it did not happen, something horrible has happened.

Parameters

- **`username`** (`str`) – The username of the user to get
- **`session`** (`ContextManagedSession`) – The session to use for retrieving the user

Returns The user to be retrieved

`generate_auth_token`(*expiration=600*, *session=ContextManagedSession(bind=Engine(sqlite:////home/docs/che server/checkouts/latest/test_db.sqlite3), expire_on_commit=True)*)

Generates a token for the user. The user's token is a UUID 1, randomly generated in this function. A *`Token`* is created with this randomly-generated UUID, the UUID token is hashed, and stored in the database.

> **Warning:** The string representation of the generated token is returned only once, and is not recoverable from the data stored in the database. The returned token is also a proxy for the user's password, so treat it as such.

Parameters

- **`expiration`** (`int`) – The lifetime of the token in seconds.
- **`session`** (`ContextManagedSession`) – The database session with which this method will interact, in order to produce a token. By default, this is a `ContextManagedSession` that will point to `conf.DATABASE_ENGINE`, but for the purposes of unit testing, it can be repointed.

Returns A tuple containing the newly-created authentication token, and the expiration date of the new token. The expiration date is an object of type `Datetime`

Return type tuple(str, Datetime)

static `hash_password`(*password*)

Hash the user's password :param str password: The password to hash :return:

`verify_auth_token`(*token_string*)

Loads the user's current token, and uses that token to check whether the supplied token string is correct

Parameters **`token_string`** (`str`) – The token to validate

Returns `True` if the token is valid and `False` if not

`verify_password`(*password*)

Verify the user's password :param str password: The password to verify :return: True if the password is correct, else False :rtype: bool

Database Management API

Contains API documentation for the database schema, session management, and database version management

ContextManagedSession

Contains classes related to session management with the DB.

class `database.sessions.`**`ContextManagedSession`**(*bind=None*, *autoflush=True*, *expire_on_commit=True*, *_enable_transaction_accounting=True*, *autocommit=False*, *twophase=False*, *weak_identity_map=True*, *binds=None*, *extension=None*, *info=None*, *query_cls=<class 'sqlalchemy.orm.query.Query'>*)

An extension to `sqlalchemy.orm.Session` that allows the session to be run using a `with` statement, committing all changes on an error-free exit from the context manager.

This class also allows deep copies of itself to be injected dynamically into function arguments, when employed as a decorator.

Note: In order to make the best use of *`ContextManagedSession`*, it is recommended that this class is first instantiated in a module-level scope, with its `bind` (i.e. the SQLAlchemy engine or connection to which the session is bound), declared in as global a scope as possible. In particular, engines should be declared as globally as possible in order to maximize the efficiency of SQLAlchemy's connection pooling features.

The module-level instantiation is then used as a master from which copies are created as needed.

** Example **

The following example shows how to use the session as both a context manager and a decorator

```
import engine # import the DB engine from somewhere else if needed

session = ContextManagedSession(bind=engine)

@session()
def do_something(session):
    session.query(something)

def do_something_context_managed():

    with session() as new_session:
        new_session.query(something)
```

`__call__`(*f=None*)

Returns itself, provided that the argument `f` is not a callable function. (i.e., it does not have a `__call__` method defined)

Parameters **f** (*function or None*) – The function to be decorated. If no function is provided, then f is `None`.

Returns A deep copy of this session, or a decorator function that can then inject arguments dynamically into the function to be decorated.

Return type `ContextManagedSession` or a function

`__enter__`()

Magic method that opens a new context for the session, returning a deep copy of the session to use in the new context.

Returns A deep copy of `self`

Return type `ContextManagedSession`

`__exit__`(*exc_type*, *exc_val*, *_*)

Magic method that is responsible for exiting the context created in `ContextManagedSession.__enter__()`; running the required clean-up logic in order to flush changes made in the session to the database.

If there are no exceptions thrown in the `ContextManagedSession`'s context, it is assumed that the code inside the context completed successfully. In this case, the changes made to the database in this session will be committed.

If an exception was thrown in this context, then the database will be rolled back to the state before any logic inside the context manager will be executed.

The arguments taken into this method are the standard arguments taken into an `__exit__` method.

The last argument _ is used as a placeholder for the stack trace of the exception thrown in the context. This parameter is currently not used in this method, as re-throwing an exception prepends the stack trace of the exception prior to re-throw into the exception. Python is awesome this way :)!

Parameters

- **`exc_type`** (*`type`*) – The class of exception that was thrown in the context. This is `None` if no exception was thrown
- **`exc_val`** (*`exc_type`*) – The particular instance of the exception that was thrown during execution in the context. After the session is rolled back, this exception is re-thrown.

`__repr__`()

Returns a string representation of an instance of `ContextManagedSession` useful for debugging.

Returns A string representing useful data about an instance of this class

Return type str

`_decorator`(*f*)

This method decorates a function f with the session to be run. This is done by invoking the `ContextManagedSession`'s context manager to open and clean up a new session, and running the decorated function in this context. The session is then appended to the decorated function's argument list, and the function is then run.

For reconciling method names and docstring for documentation purposes, and to avoid namespace collisions in the server's routing table, the `__name__` and `__doc__` properties of the decorated function are assigned to the decorator.

Parameters **`f`** (*`function`*) – The function to decorate

Returns A function that decorates the function to be decorated

Return type function

copy()

Returns a new *ContextManagedSession* with the same namespace as `self`

Versioning

Contains tools for versioning databases, and for managing database upgrades. This code is currently not implemented in the dev version of OmicronServer, but will eventually be used as part of the server's command line interface.

This code was adapted from Miguel Grinberg's Flask mega-tutorial.

class `database.versioning.`**DatabaseManager**(*metadata=MetaData(bind=None), database_url=None, migrate_repo='/home/docs/checkouts/readthedocs.org/user_builds/om server/checkouts/latest/db_versioning_repo', api=<module 'migrate.versioning.api' from '/home/docs/checkouts/readthedocs.org/user_builds/omicron-server/envs/latest/local/lib/python2.7/site-packages/migrate/versioning/api.pyc'>, engine=Engine(sqlite:////home/docs/checkouts/readthedocs.org/user_ server/checkouts/latest/test_db.sqlite3)*)

Wraps methods for managing database versions. The constructor requires the following

Variables

- **metadata** – An object of type `sqlalchemy.MetaData`, containing information about the database schema. Defaults to the metadata object located in `database.models.schema`.
- **database_url** – The URL [RFC 3986] of the database to be upgraded or downgraded. Defaults to the database URL given in *config*, or read from command line environment variables.
- **migrate_repo** – An absolute path to the directory in which database migration scripts are stored. Defaults to the value given in *config*.
- **api** – The sqlalchemy-migrate API that will be used to perform the migration operations. This is overwritten for testability. Defaults to the API in `migrate.versioning`. See the Flask mega-tutorial for more details

__init__(*metadata=MetaData(bind=None), database_url=None, migrate_repo='/home/docs/checkouts/readthedocs.org/user_builds/omicron-server/checkouts/latest/db_versioning_repo', api=<module 'migrate.versioning.api' from '/home/docs/checkouts/readthedocs.org/user_builds/omicron-server/envs/latest/local/lib/python2.7/site-packages/migrate/versioning/api.pyc'>, engine=Engine(sqlite:////home/docs/checkouts/readthedocs.org/user_builds/omicron-server/checkouts/latest/test_db.sqlite3)*)

Instantiates the variables listed above :raises: DatabaseNotReferencedError if the Database-Manager is created

without a database url or a database engine

__repr__()

Returns a representation of the Database Manager useful for debugging :return:

`__weakref__`
list of weak references to the object (if defined)

`create_db``(`*engine=None*`)`
Create a database at `self.database_url`

Parameters **`engine`** – The SQLAlchemy engine which will be used to create the database

`create_migration_script``()`
Creates a new migration script for the database. The migration script is a python script located at `self.migrate_repo`. The target version of the script is listed on the file name. Each migration script must contain the following two functions.

`upgrade` describes how to move from the previous version to the version written in the file name.

`downgrade` describes how to downgrade the database from the version written on the file name to the previous version.

Warning: It is recommended that each migration script is reviewed prior to use, ESPECIALLY IN PRODUCTION. Automatically-generated migration scripts have known issues with migrations, particularly with queries involving `ALTER COLUMN` queries. In such situations, the migration script can easily `DROP` the old column and create a new one. Care should be taken when running migrations.

`downgrade_db``()`
Downgrade the DB from the current version to the decremented previous version.

`upgrade_db``()`
Upgrade the database to the most current version in the migrate repository, by running `upgrade` in all the migration scripts from the database's version up to the current version.

`version`
Return the version of the database using the SQLAlchemy-migrate API

exception `database.versioning.`**`DatabaseNotReferencedError`**
Thrown if :class:database.versioning.DatabaseManager‘ is instantiated without an engine or a database url.

`__weakref__`
list of weak references to the object (if defined)

Schema

Contains the database schema to be implemented or queried, from which models in `database.models` will be populated. This describes the database to be built on server initialization. This is the most current version of the database schema.

5.1.3 API Views

Schema-defined Resource

class `views.schema_defined_resource.`**`SchemaDefinedResource`**
Abstract class that defines an API view with a schema built into it. This class is capable of serving its schema and

is_schema_draft3
Checks if `self.schema` corresponds to the Draft 3 JSON Schema

Note: At the time of writing, the latest draft of JSON Schema is Draft 4. Refrain from using Draft 3 schemas in this API wherever possible

Returns True if the schema conforms to draft 3, else false

Return type bool

schema
Return the JSON schema of the view as a dictionary

validate(*dict_to_validate*, *source_dict=None*)
Checks that the supplied dictionary matches the JSON Schema in another dictionary. If no `source_dict` is provided, the method will attempt to validate the validation dictionary against *attr:self.schema*.

Parameters

- **dict_to_validate** (*dict*) – The dictionary representing the JSON against the JSON Schema to be validated
- **source_dict** – The dictionary representing the JSON Schema against which the incoming JSON will be compared

Returns A tuple containing a boolean corresponding to whether the schema validated or not, and a message. If the schema validated successfully, the message will be `"success"`. If not, then the message will correspond to the reason why the schema did not successfully validate

Projects

class `views.projects.`**ProjectContainer**
Maps the /projects endpoint

get(**args*, ***kwargs*)
Returns the list of projects accessible to the user

Example Response

```
HTTP/1.1 200 OK
Content-type: application/json
page: 1
items_per_page: 1
Count: 1

{
    "projects": [
        {
            "name": "NMR Project",
            "description": "This is a project",
            "date_created": "2015-01-01T12:00:00",
            "owner": {
                "username": "mkononen"
                "email": "my@email.com"
            }
```

```
                }
            ]
        }
```

Statuscode 200 Request completed without errors

Statuscode 400 Bad request, occurred due to malformed JSON or due to the fact that the user was not found

Parameters **`pag_args`** (`PaginationArgs`) – A named tuple injected into the function's arguments by the `@restful_pagination()` decorator, containing parsed parameters for pagination

Returns A flask response object containing the required data to be displayed

`post`(**args*, ***kwargs*)

Create a new project

Example Request

```
HTTP/1.1
Content-Type: application/json

{
    "name": "NMR Project",
    "description": "NMR Project Description",
    "owner": "mkononen"
}
```

Example Response

```
HTTP/1.1 201 CREATED
Content-Type: application/json

{
    "name": "NMR Project",
    "description": "NMR Project Description",
    "owner": {
        "username": "mkononen",
        "email": "my@email.com"
    }
}
```

Statuscode 201 Project successfully created

Statuscode 400 Unable to create project due to malformed JSON

Parameters **`session`** (`ContextManagedSession`) – The database session to be used for making the request

Returns A Flask response

Return type flask.Response

class `views.projects.`**`Projects`**

Maps the "/projects/<project_id>" endpoint

exception **`ProjectNotFoundError`**

Thrown if `__getitem__` cannot find the required project

`__weakref__`
list of weak references to the object (if defined)

`Projects.`**`__delitem__`**`(`**args*, ***kwargs*`)`
Retrieve the required project name or ID corresponding r

Parameters

- **`session`** (`ContextManagedSession`) – The database session that this method will use to communicate with the database
- **`project_name_or_id`** (`str`) – The project name or ID that will be used as the key to find the project

`Projects.`**`__getitem__`**`(`**args*, ***kwargs*`)`
Return the project corresponding to the name or ID

Parameters

- **`session`** (`ContextManagedSession`) – The database session to be used for making the request. This is injected into the method using the `@database_session()` decorator
- **`project_name_or_id`** (`str`) – The project name or the project ID

Returns The project model class from the database

Return type *database.models.projects.Project*

Raises `Projects.ProjectNotFoundError` if the project is not found in the DB

`Projects.`**`delete`**`(`**args*, ***kwargs*`)`
Delete a project

Statuscode 200 The project was deleted successfully

Statuscode 404 Unable to find the project to delete

Parameters or str project_name_or_id (`int`) – The project to delete

`Projects.`**`get`**`(`**args*, ***kwargs*`)`
Returns the details for a given project

Example Response

```
HTTP/1.1 200 OK

{
    "name": "NMR Project",
    "description": "NMR Project Description",
    "owner": {
        "username": "mkononen",
        "email": "my@email.com"
    }
}
```

Statuscode 200 The Request completed successfully

Statuscode 404 The request project could not be found

Parameters or str project_name_or_id (`int`) – The id or name of the project to retrieve

Returns A flask response object containing the required data

Return type `flask.Response`

Users

Contains views for the `/users` endpoint.

class `views.users.`**`UserContainer`**

Maps the /users endpoint's `GET` and `POST` requests, allowing API consumers to create new users

`get`(**args*, ***kwargs*)

Process a GET request for the /users endpoint

Example request

```
GET /api/v1/users HTTP/1.1
Host: example.com
Content-Type: application/json
```

Example response

```
Vary: Accept
Content-Type: application/json

.. include:: /schemas/users/examples/get.json
```

Query contains A string specifying what substring should be contained in the username. Default is `''`

Query starts_with The substring that the username should start with. Default is `''`.

Query page The number of the page to be displayed. Defaults to `1`.

Query items_per_page The number of items to be displayed on this page of the results. Default is `1000` items.

Statuscode 200 no error

Parameters

- **`session`** (*`Session`*) – The database session used to make the request
- **`pag_args`** (`PaginationArgs`) – The pagination arguments generated by the `@restful_pagination()` decorator, injected into the function at runtime

Returns A flask response object with the search request parsed

static `parse_search_query_params`(*request*)

This method enables text search over the results returned by a `GET` request on the `/users` endpoint. It parses query parameters from `Flask.request`, a container for the HTTP request sent into the method. Currently, the query parameters parsed here are

•starts_with: A string stating what the username should begin with

•contains: A string specifiying what the username should contain

•ends_with: A string stating what the username should end with

Parameters **`request`** – The flask request from which query parameters need to be retrieved

Returns A `%` delineated search string ready for insertion as a parameter into a SQL or SQLAlchemy query language's `LIKE` clause

post (**args*, ***kwargs*)
Create a new user

Example Request

```
HTTP/1.1
Content-Type: application/json

{
    "username": "scott",
    "password": "tiger",
    "email": "scott@tiger.com"
}
```

Example Response

```
HTTP/1.1 201 CREATED
Content-Type: application/json

{
    "username": "scott",
    "email": "scott@tiger.com
}
```

Statuscode 201 The new user has been created successfully

Statuscode 400 The request could not be completed due to poor JSON

Statuscode 422 The request was correct, but the user could not be created due to the fact that a user with that username already exists

Parameters **session** (*Session*) – The database session that will be used to make the request

class views.users.**UserView**
Maps the `/users/<username>` endpoint

get (**args*, ***kwargs*)
Returns information for a given user

Example Response

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "username": "scott"
    "email": "scott@tiger.com"
    "projects": [
        {
            "name": "NMR Experiment 1",
            "description": "Measure this thing in NMR",
            "date_created": "2015-01-01T12:00:00Z",
            "owner": {
                "username": "scott",
                "email": "scott@tiger.com"
            }
```

```
                        }
                    ]
                }
```

Parameters

- **`session`** (*`Session`*) – The database session to be used in the endpoint
- **`or str username_or_id`** (*`int`*) – The username or user_id of the user for which data needs to be retrieved

CHAPTER 6

Tests

Contains documentation for integration and unit tests for OmicronServer.

Contents:

6.1 Integration Tests

The purpose of these tests is to test server functionality without mocking. These tests are not so much about attaining code coverage, as they are about ensuring that OmicronServer "plays nice" with other components.

6.1.1 Test Database Round-Trip

Tests that the server can successfully write and read from the database in a "Round Trip"

class `tests.integration.test_database_roundtrip.`**`TestDatabaseRoundTrip`**(*methodName='runTest'*)

Tests that SQLAlchemy is successfully able to connect to the database, store an object in the DB, and retrieve it successfully.

`test_write`()

Tests the round trip

6.1.2 Test Auth

Contains integration tests for *`auth`*, performing authentication without stubbing out any functionality.

class `tests.integration.test_auth.`**`TestAuth`**(*methodName='runTest'*)

Base class for testing *`auth`*

classmethod **`setUpClass`**()

Set up the test database, and put the server into a request context requiring authentication.

classmethod **`tearDownClass`**()

Clean up a test suite by dropping all tables in the test database.

class `tests.integration.test_auth.`**`TestVerifyPassword`**(*methodName='runTest'*)

Tests the *`auth.verify_password()`* callback

`test_verify_correct_uname_bad_pwd`()

Tests that the callback returns `False` if the correct username but an incorrect password are supplied.

test_verify_incorrect_username_correct_pwd()
Tests that the callback returns `False` if an incorrect username but correct password are supplied.

test_verify_password_correct_credentials()
Tests that the callback returns `True` if the correct username and password are provided.

class `tests.integration.test_auth.`**TestVerifyToken**(*methodName='runTest'*)
Tests `auth._verify_token()`

setUp()
Starts up the server, and requests an authentication token using the correct login credentials.

test_token_auth()
Tests that the `verify_password` callback can authenticate a user when the correct token is provided.

test_token_auth_bad_token()
Tests that the user cannot be authenticated if an incorrect token is provided.

6.2 Unit Tests

Contains standalone unit tests for the Omicron Server. These tests take advantage of mocking via unittest.mock to ensure high code coverage, and to isolate the components of OmicronServer apart from themselves and outside components. These tests test code, and only code.

6.2.1 Top-Level Modules

Contains documentation for unit tests in top-level modules in the unit testing library

API Server

Contains unit tests for *`api_server`*

class `tests.unit.test_api_server.`**TestAPIServer**(*methodName='runTest'*)
Base class for unit tests in *`api_server`*

classmethod setUpClass()
Sets up basic parameters for testing in *`api_server`*.

classmethod tearDownClass()
Tear down the tests

class `tests.unit.test_api_server.`**TestGetAuthToken**(*methodName='runTest'*)
Tests *`api_server.create_token()`*

setUp()
Set up the tests by creating a mock token, and assigning it as a return value to `self.user`, which is assigned to `g.user`.

class `tests.unit.test_api_server.`**TestHelloWorld**(*methodName='runTest'*)
Tests *`api_server.hello_world()`*

test_hello_world()
Tests that a request to the server's root endpoint returns 200, indicating that the server has set up and is running successfully.

class `tests.unit.test_api_server.`**`TestRevokeToken`**`(`*methodName='runTest'*`)`

Tests *`api_server.revoke_token()`*

Auth

Contains unit tests for *`auth`*

Config

Decorators

JSON Schema Parser

Contains unit tests for *`json_schema_parser`*

CHAPTER 7

Introduction

OmicronServer forms the back end of the Omicron Project, which aims to automate a large part of laboratory processes and record-keeping.

7.1 Audience

This document is intended for developers coming from a non-Python background to understand how this software works, and how to contribute additional functionality to the software.

In addition, this document is intended for reading by potential consumers of this API, in order to understand what each HTTP request in this API does.

CHAPTER 8

Readme

Welcome to OmicronServer

OmicronServer is the back end for Omicron Web Services, a project that aims to automate much of the process engineering and experimentation

9.1 Badges

9.2 Installation

A stable version of **OmicronServer** has not yet been released. TravisCI builds are run on all branches and pull requests with every commit, but features may be lacking in these builds. See the issues and project milestones for a timeline of future releases.

```
$ git clone https://github.com/MichalKononenko/OmicronServer.git
```

9.2.1 Packages and Dependencies

Dependencies are outlined in `requirements.txt`. In order to install dependencies, run

```
pip install -r requirements.txt
```

TravisCI uses this file to install all its dependencies, and Heroku uses this to identify **OmicronServer** as a Python project.

Since Travis runs on Linux, it has the advantage of using `gcc` to compile some of the packages listed here. your machine may not have this luxury, and so you may have to find binaries for the psycopg2 package, in order to connect to PostgreSQL

9.2.2 Supported Versions

This project is currently supported for Python versions

- 2.7
- 3.4
- 3.5

9.2.3 Supported Databases

OmicronServer currently supports PostgreSQL and SQLite.

9.3 Running the Program

To run the server, run

```
$ python run_server.py
```

from the command line. By default, the server will run on `localhost:5000`. The address can be specified by setting the `IP_ADDRESS` and `PORT` environment variables in your command line.

9.3.1 Environment Variables

The file `config.py` will adjust its settings depending on the value of several environment variables. These are

- `IP_ADDRESS`: The IP address at which the server is to be hosted
- `PORT`: The port number at which the server is to be hosted
- `BASE_DIRECTORY`: The base directory where `run_server.py` is kept. By default, this is the current directory of the `config.py` file
- **`TOKEN_SECRET_KEY`: The secret key used as a salt to generate authentication tokens for the user. Tokens are not stored**
  the back end, but are generated from user data, and the value of `TOKEN_SECRET_KEY`. If this value is changed, the user's token will no longer work.
- `DATABASE_URL`: The URL at which the database sits. This is the database to be used by the server
- `DEBUG`: If `TRUE`, then stack traces will be displayed when **in the browser** if the server throws a 500 error
- `STATE`: A generic flag for specifying between `DEV`, `CI`, and `PROD` machines. Currently not wired to anything
- **`LOGFILE`: The file to which the log should be written. If this is not** defined, the application log will be
  written to `'sys.stdout'`.

9.3.2 Command Line Parsing

Unfortunately, the values above must be set as environment variables as OmicronServer does not currently support parsing these arguments in the command line

9.4 License

This project and all files in this repository are licensed under the GNU General Public License (GPL) version 3. A copy of this license can be found in the `LICENSE` file

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index