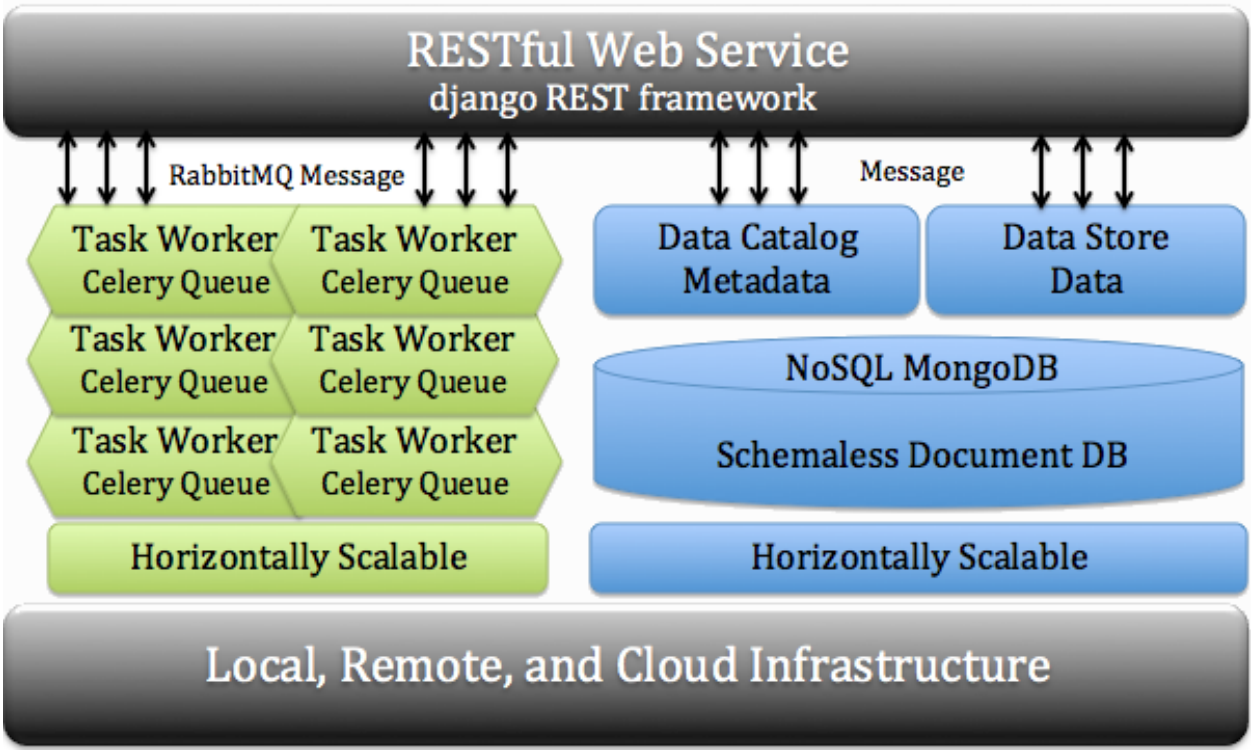
cyberCommons Framework

cyberCommons

Nov 16, 2021

INSTALLATION/CONFIGURATION

The cyberCommons Framework is a loosely coupled service-orientated reference architecture for distributed computing workflows. The framework is composed of a series of Docker contained services combined by a Python RESTful API. These containers in the reference architecture use MongoDB, RabbitMQ, Django RESTful and Celery to build a loosely coupled and horizontally scalable software stack. This reference stack can be used to manage data, catalog metadata, and register computational worker nodes with defined tasks. Computations can scale across a series of worker nodes on bare-metal or virtualized environments. The framework provides a flexible, accessible interface for distributed processing and data management from multiple environments including command-line, programming languages, and web and mobile applications.



The cyberCommons Framework currently deployed across a wide variety of environments.

1. University of Colorado Libraries at the University of Colorado Boulder.
2. University of Oklahoma Libraries at the University of Oklahoma.
3. US Congressional Hearings Search Engine
4. Latin Search Engine
5. Northern Arizona University EcoPAD is an ecological platform for data assimilation and forecasting in ecology.
6. The Oklahoma Biological Survey
7. The Earth Observation Modeling facility
8. The South Central Climate Sciences Center
9. The Oklahoma Water Survey

CHAPTER

ONE

TABLE OF CONTENTS

Contents:

1.1 Installation

The Cybercommons framework is a Django Rest Framework API. The API leverages MongoDB to provide a Catalog and Data Store for storing metadata and data within a JSON document database. The API also includes Celery which is an asynchronous task queue/jobs based on distributed message passing.

1.1.1 Requirements

- Docker
- Docker Compose
  - `pip install docker-compose`
- GNU Make or equivalent

1.1.2 Installation

1. Clone Repository

```
git clone https://github.com/cybercommons/cybercommons.git
```

2. Edit values within dc_config/cybercom_config.env
3. Copy secrets_template.env into secrets.env under the same folder and add required credentials into it.
4. Initialize database and generate internal SSL certs

```
make init
```

5. Build and Deploy on local system.

```
make build
make superuser
make run
```

6. Make Django's static content available. It only needs to be run once or after changing versions of Django.

```
make collectstatic
```

7. API running http://localhost
    - Admin credentials set from above `make superuser`
8. Shutdown cybercommons

```
make stop
```

1.1.3 cybercommons Installation on servers with a valid domain name.

1. Edit values within dc_config/cybercom_config.env[NGINX_HOST,NOTIFY_EMAIL,NGINX_TEMPLATE(These values must be set).
2. Copy secrets_template.env into secrets.env under the same folder and add required credentials into it.
3. Initialize database and generate internal SSL certs

```
make init
```

4. Initialize and Get TLS certificates from LetsEncrypt

```
make init_certbot
```

5. Build and Deploy on local system.

```
make build
make superuser
make run
```

6. Make Django's static content available. This only needs to be ran once or after changing versions of Django.

```
make collectstatic
```

7. API running https://{domain-name-of-server}
    - Admin credentials set from above `make superuser`
8. Shutdown cybercommons

```
make stop
```

1.1.4 TODO

1. Integration with Kubernetes

1.2 System Configuration

1.2.1 Configuration Files

The majority of configuration settings are stored in the following files:

- dc_config/cybercom_config.env
  - Used for general application settings and container versions
  - Configure Nginx to use Let's Encrypt
  - Configure MongoDB database name and Docker volume prefix
  - Set the ALLOWED_HOSTS setting - this must be updated if running on a publicly accessible server!
- dc_config/secrets.env (This should be copied from dc_config/secrets_template.env as a starting point)
  - !!! Once created, you should change the default credentials as they are not secure !!!
  - Used to store sensitive variables that should not be tracked in version control
  - Set MongoDB and RabbitMQ credentials
  - Configure email server connection
  - SSL configration
  - Configure Let's Encrypt reminder notification email address (NOTIFY_EMAIL)
- requirements.txt
  - Python requirements for the API / Django
- dc_config/images/celery/requirements.txt
  - Python requirements for the dockerized Celery container

It is recommended to copy dc_config/secrets_template.env to dc_config/secrets.env as a starting point. Once created, you should change the default credentials as they are not secure!

1.2.2 Generating SSL Keys and Where They are Stored

Rabbitmq and MongoDB are configured to use SSL certificates to secure their communications. By default, during the setup of cyberCommons, these certificates are configured to be valid for 365 days. This default can be changed by editing the CA_EXPIRE value in the dc_config/secrets.env file. Once the certificates expire, they will need to be regenerated by running `shell make initssl`

Generating SSL certificates

Self-signed certificates are automatically generated on first run for RabbitMQ and MongoDB. Generation of self-signed certificates for NGINX is currently not implemented. LetsEncrypt - refer to the LetsEncrypt section of the installation instructions.

Renewing SSL Certificates

1. Self-signed certificates can be updated by running the following command from the cyberCommons root directory:

```
$ make initssl
```

All remote Celery workers will need the new SSL client certificates to resume communications. See the section below on where these certificates are stored

1. LetsEncrypt certificates can be renewed by running the following from the cyberCommons root directory:

```
$ make renew_certbot
```

Follow LetsEncrypt's prompts

SSL Certificate Locations

1. Self-signed locations:
    - MongoDB
        - dc_config/ssl/backend/client/mongodb.pem
        - dc_config/ssl/backend/server/mongodb.pem
        - dc_config/ssl/testca/cacert.pem
    - RabbitMQ
        - dc_config/ssl/backend/client/key.pem
        - dc_config/ssl/backend/client/cert.pem
        - dc_config/ssl/backend/server/key.pem
        - dc_config/ssl/backend/server/cert.pem
        - dc_config/ssl/testca/cacert.pem
2. LetsEncrypt location:
    - NGINX
        - dc_config/ssl/nginx/letcencrypt/etc/live/*

1.2.3 Configure Email Backend

- Populate the Email Configuration section in dc_config/secrets.env. *The following is an example using gmail.*

```
EMAIL_BACKEND=django.core.mail.backends.smtp.EmailBackend
EMAIL_HOST=smtp.gmail.com
EMAIL_PORT=587
EMAIL_HOST_USER=username@gmail.com
EMAIL_HOST_PASSWORD=password
EMAIL_USE_TLS=True
```

Turn On Debug Mode for RESTful API

The Debug mode is turned off by default. If you need debug messages

1. Set DEBUG=True in dc_config/cybercom_config.py
2. Add host(s) to ALLOWED_HOSTS list if needed. See Django's documentation on the ALLOWED_HOSTS setting for more detail.

1.3 Install Remote Workers

cyberCommons can scale horizontally by allowing remote workers to take on tasks and execute them on remote systems. The following describes how to setup a remote Celery worker for use with cyberCommons. Celery is focused on real-time operation, but supports scheduling as well.

The execution units, called tasks, are executed concurrently on a single or more worker servers using multiprocessing, Eventlet, or gevent. Tasks can execute asynchronously (in the background) or synchronously (wait until ready).

1.3.1 Requirements

- PIP - Install
- Copies of client certificates and credentials to communicate with central cyberCommons server:
  - MongoDB
    - dc_config/ssl/backend/client/mongodb.pem
    - dc_config/ssl/testca/cacert.pem
  - RabbitMQ
    - dc_config/ssl/backend/client/key.pem
    - dc_config/ssl/backend/client/cert.pem
    - dc_config/ssl/testca/cacert.pem
- RabbitMQ and MongoDB ports are open by default:
  - RabbitMQ port 5671
  - MongoDB port 27017

1.3.2 Install Celery

1. Create virtual environment and activate

```
python -m venv virtpy
source virtpy/bin/activate
```

2. Install Celery

```
(virtpy) $ pip install Celery
```

1.3.3 Configuration

Get Config Files and Certificates

1. Download example celeryconfig.py and requirements.txt

```
wget https://raw.githubusercontent.com/cybercommons/cybercommons/master/docs/pages/files/
↪celeryconfig.py
```

1. Create SSL directory and copy cyberCommon's client certificates

```
mkdir ssl
cp mongodb.pem ssl/
cp key.pem ssl/
cp cert.pem ssl/
cp cacert.pem ssl/
```

1. Configure celeryconfig.py to point to client certificates and use corresponding credentials (values in this example between "<" and ">" need to be updated to match your cyberCommon's configuration. Do not include the "<" and ">" characters.)

```
broker_url = 'amqp://<username>:<password>@<broker_host>:<broker_port>/<broker_vhost>'
broker_use_ssl = {
    'keyfile': 'ssl/key.pem',
    'certfile': 'ssl/cert.pem',
    'ca_certs': 'ssl/cacert.pem',
    'cert_reqs': ssl.CERT_REQUIRED
}


result_backend = "mongodb://<username>:<password>@<mongo_host>:<mongo_port>/?ssl=true&
↪ssl_ca_certs=ssl/cacert.pem>&ssl_certfile=mongodb.pem>"

mongodb_backend_settings = {
    "database": "<application_short_name>",
    "taskmeta_collection": "tombstone"
}
```

Configure Tasks

1. Update requirements.txt to include desired libraries and task handlers.
2. Update celeryconfig.py to import task handlers that have been included in requirements file.

```
imports = ("cybercomq", "name_of_additional_task_handler_library", )
```

3. Install requirements

```
(virtpy) $ pip install -r requirements.txt
```

Launch Celery worker

1. Run in foreground. See Celery Worker Documentation for more information.

```
celery worker -Q remote -l INFO -n dev-hostname
```

1.4 RESTful API

1.4.1 Catalog and Data Store

The Catalog and Data Store are using the same logic and syntax for access and query language. The database which holds the information is MongoDB. MongoDB is a schemaless document noSQL database. The query language that the API deploys is the json representation of MongoDB.

API Return Data Structure

The API returns data in a consistent structure.

- count: number if result records returned
- meta: page, page_size, pages
- next and previous: urls to page through data
- results: list of records return from API

```
{
    "count": 1,
    "meta": {
        "page": 1,
        "page_size": 50,
        "pages": 1
    },
    "next": null,
    "previous": null,
    "results": [

    ]
}
```

URL Parameters

page_size:

The page_size returns the available records up to page_size. If more records exist, the next url value will be deployed.

```
?page_size=100
?page_size=0
```

If page_size=0 API will return all records.

page:

The page variable will move to the page requested. If the page does not exist the last page will be shown.

format:

1. api (Default) - Return type is HTML format
2. json - Return type is JSON format
3. jsonp - Return type is JSONP format
4. xml - Return type is xml format

```
?format=json
```

query:

The query url parameter is a JSON format query language. Please see below

Query Language

The API query language is based from the MongoDB pyhton query syntax.

Create Database and Collections

Create Database

```
View: /api/data_store/data/  HTTP Request: Post
Data: {"database":"mydata"}  Format: JSON
```

Delete Database

```
View: /api/data_store/data/  HTTP Request: Post
Data: {"action":"delete","database":"mydata"}  Format: JSON
```

Create Collection

```
View: /api/data_store/data/mydata  HTTP Request: Post
Data: {"collection":"mycollection"}  Format: JSON
```

Delete Collection

```
View: /api/data_store/data/mydata  HTTP Request: Post
Data: {"action":"delete","collection":"mycollection"}  Format: JSON
```

Filter Query

The following examples are on the collection view.

Filter Query

```
?query={"filter":{"tag":"content"}}

?query={"filter":{"tag":"content","tag2":"content"}}

# Return fields (projection: 0,1)

?query={"filter":{"tag":"content","tag2":"content"},"projection":{"tag":0}
```

Distinct Query

```
?distinct=tag,tag2
# Include query parameter
?distinct=tag&query={"filter":{"department":"Informatics"}}
```

MongoDb Aggregation

Please refer to MongoDB Documentation

```
?aggregate=[{"$match":{"status": "urgent"}},
  {"$group":{"_id":"$productName","sumQuantity":{"$sum":"$quantity"}}}]
```

Task Execution (celery)

The Celery Distributed Task Queue is integrated throught the RESTful API.

List of Available Tasks and Task History

```
URL: /api/queue/
Task History: /api/queue/usertasks/
```

Task Submission

```
    Example:
    URL /api/queue/run/cybercomq.tasks.tasks.add/
    Docstring: Very import to give users the description of task.
    Curl Example: Comand-line example with API token
```

Task HTML POST Data Requirement

```
    {
        "function": "cybercomq.tasks.tasks.add",
        "queue": "celery",
        "args": [],
        "kwargs": {},
        "tags": []
    }
```

function: task name queue: which queue to route the task args: [] List of argument kwargs: {} Keyword arguments tags: [] list of tags that will identify task run

Curl Command - Command-line Scripting

```
    curl -X POST --data-ascii '{"function":"cybercomq.tasks.tasks.add","queue":"celery",
↪"args":[],"kwargs":{  },"tags": []}' http://localhost/api/queue/run/cybercomq.tasks.
↪tasks.add/.json -H Content-Type:application/json -H 'Authorization: Token < authorized-
↪token > '
```

Python Script to Execute Script

```
    import requests,json

    headers ={'Content-Type':'application/json',"Authorization":"Token < authorized␣
↪token >"}
    data = {"function":"cybercomq.tasks.tasks.add","queue":"celery","args":[2,2],"kwargs
↪":{},"tags":["add"]}
    req=requests.post("http://localhost/api/queue/run/cybercomq.tasks.tasks.add/.json",
↪data=json.dumps(data),headers=headers)
    print(req.text)
```

Javascript JQuery $.postJSON

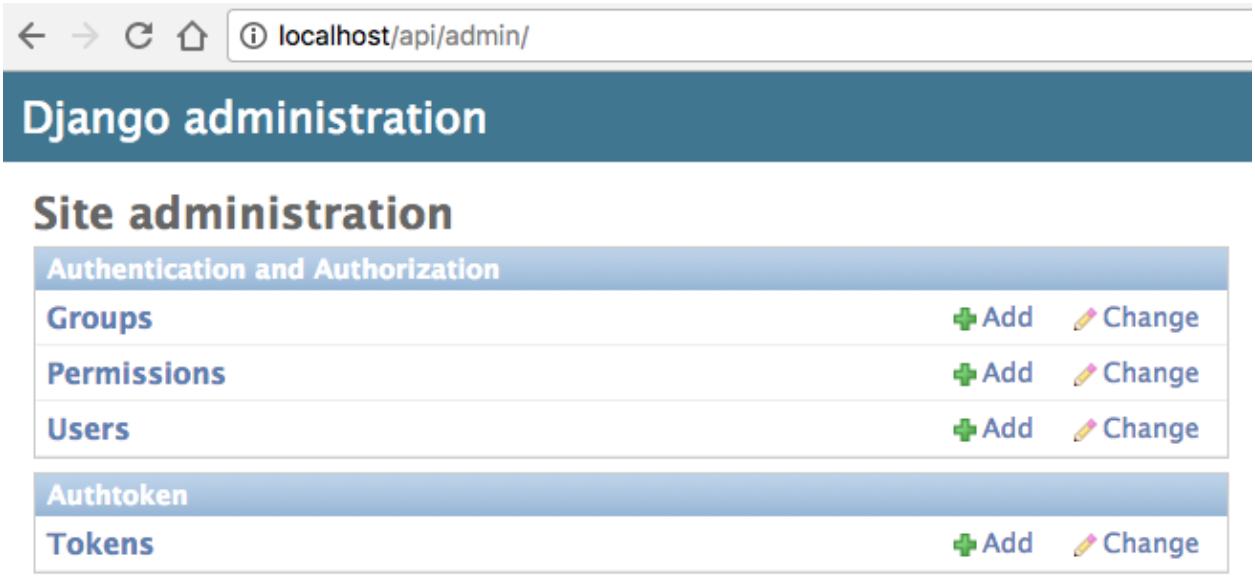
```
//postJSON is custom call for post to cybercommons api
$.postJSON = function(url, data, callback,fail) {
    return jQuery.ajax({
        'type': 'POST',
        'url': url,
        'contentType': 'application/json',
        'data': JSON.stringify(data),
        'dataType': 'json',
        'success': callback,
        'error':fail,
        'beforeSend':function(xhr, settings){
            xhr.setRequestHeader("X-CSRFToken", getCookie('csrftoken'));
        }
    });
}
```

1.5 Users and Persmissions

1.5.1 Django Admin Site

The Django admin comes with user and permissions functionality.

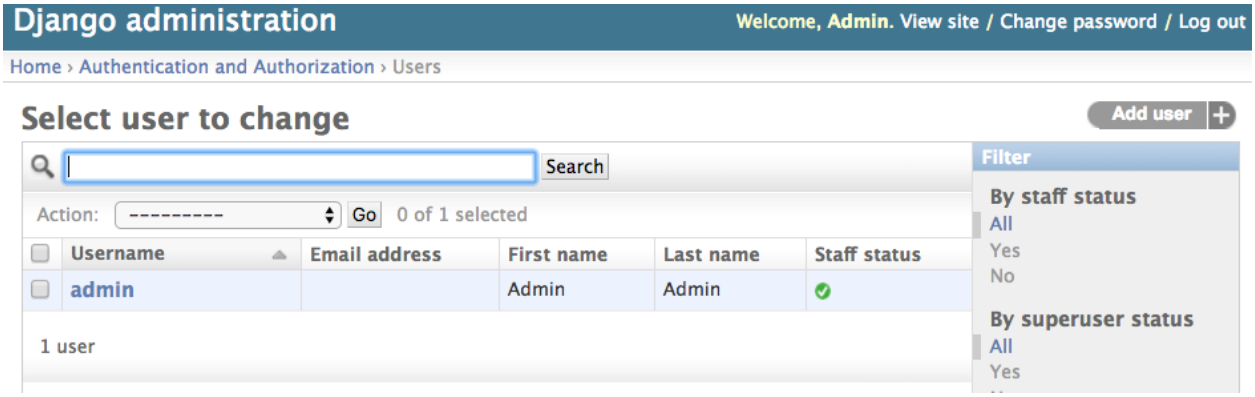
```
URL - /api/admin
```



User Creation

The users are stored locally and passwords are stored within the database. Django comes with many different modules to extend the authentication functionality.

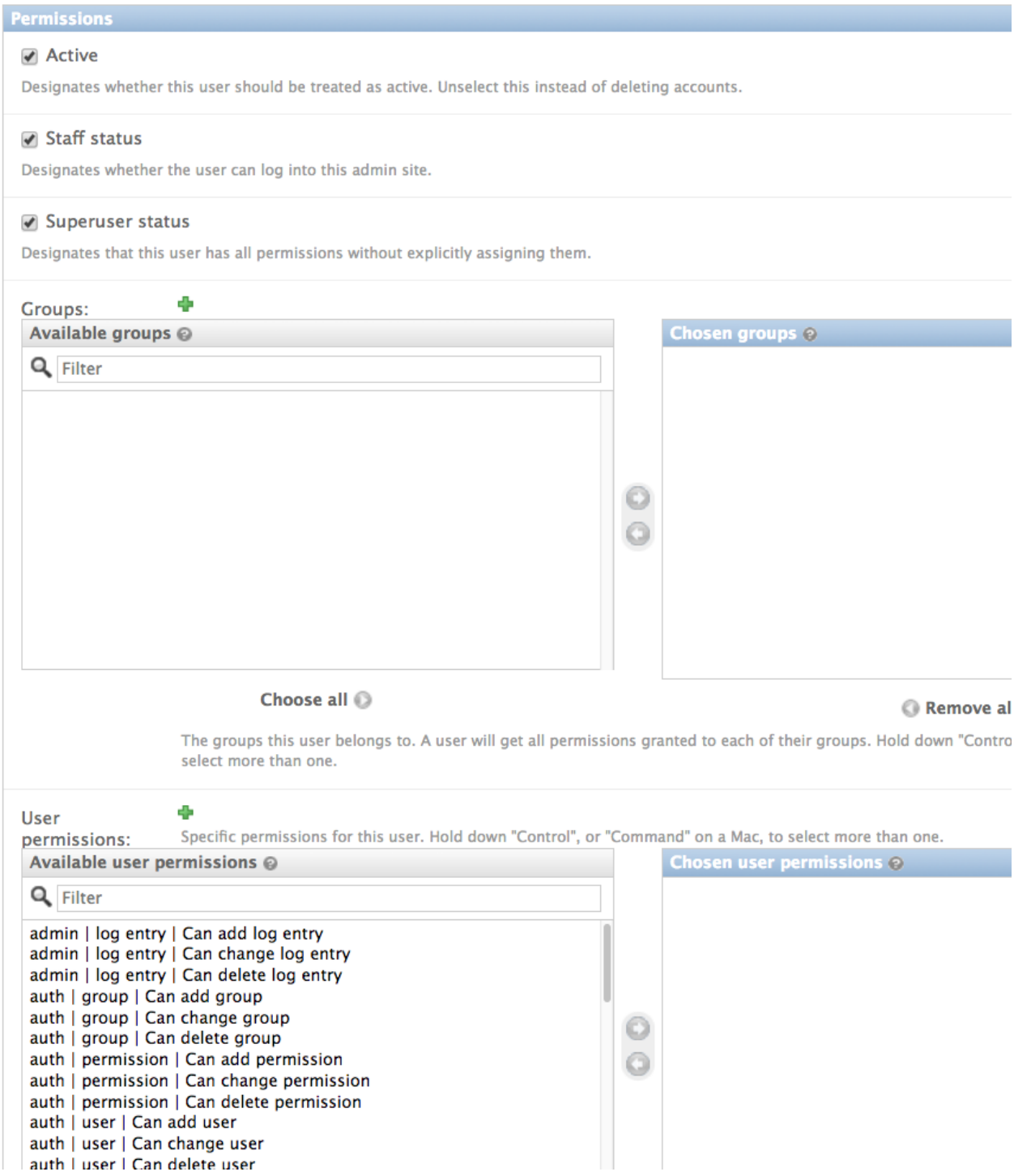
```
URL - /api/admin/auth/user/
```



Permissions

The cyberCommons RESTful api provides permissions and groups:

1. Data Catalog
    - Catalog Creation
        - Catalog Admin
        - Create Catalog Collections
    - Collection Permissions
        - Add Permissions
        - Update Permission
        - Safe Methods (Read) Permissions
2. Data Store
    - Catalog Creation
        - Data Store Admin
        - Create Database and Collections
    - Database and Collection Permissions
        - Add Permissions
        - Update Permission
        - Safe Methods (Read) Permissions


Permissions
Active
Designates whether this user should be treated as active. Unselect this instead of deleting accounts.
Staff status
Designates whether the user can log into this admin site.
Superuser status
Designates that this user has all permissions without explicitly assigning them.
Groups:
Available groups
Filter
Chosen groups
Choose all
Remove al
The groups this user belongs to. A user will get all permissions granted to each of their groups. Hold down "Contro
select more than one.
User permissions:
Specific permissions for this user. Hold down "Control", or "Command" on a Mac, to select more than one.
Available user permissions
Filter
admin | log entry | Can add log entry
admin | log entry | Can change log entry
admin | log entry | Can delete log entry
auth | group | Can add group
auth | group | Can change group
auth | group | Can delete group
auth | permission | Can add permission
auth | permission | Can change permission
auth | permission | Can delete permission
auth | user | Can add user
auth | user | Can change user
auth | user | Can delete user
Chosen user permissions

1.6 Help and Issue Reporting

1.6.1 Help

This documentation serves as the primary resource for help on the cyberCommons Framework.

1.6.2 Issue Reporting

- cyberCommons Framework

1.7 Contributors

The original cyberCommons framework was funded by the National Science Foundation(NSF) through the Oklahoma EPSCoR Track-II RII (EPS-0919466 grant. The grant focused on creating a cyberCommons, a powerful, integrated cyber environment for knowledge discovery and education across complex environmental phenomena. Specifically, the cyberCommons will integrate two frameworks— the science framework of data, models, analytics and narratives, and the cyberinfrastructure framework of hardware, software, collaboration environment and integration environment. The current cyberCommons platform has evolved and is used in production for research and automating workflows including:

1. University of Colorado Libraries
2. University of Oklahoma Libraries
3. Northern Arizona University
4. The Earth Observation Modeling facility
5. The South Central Climate Sciences Center

1.7.1 Informatics contributions

Original cyberCommons Team

```
* Cremeans, Brian
* Duckles, Jonah
* Stacy, Mark
```

Current cyberCommons Team

```
* Mark Stacy, Software Architect, University of Colorado Libraries
* Tyler Pearson, Director of Informatics, University of Okalahoma Libraries
```