authnzerver

Release 0.post.dev87

Jan 29, 2022

Documentation

Authnzerver is a tiny authentication (authn) and authorization (authz) server implemented in Python and the Tornado web framework.

I wrote it to help with the login/logout/signup flows for the Light Curve Collection Server and extracted much of the code from there. It builds on the auth bits there and is eventually meant to replace them. It can do the following things:

- Handle user sign-ups, logins, logouts, and locks/unlocks.
- Handle user email verification, password changes, forgotten password processes, and editing user properties.
- Handle API key issuance and verification.
- Handle access and rate-limit checks for arbitrary schemes of user roles, permissions, and target items. There is a default scheme of permissions and user roles, originally from the LCC-Server where this code was extracted from. A custom permissions policy can be specified as JSON.

Authnzerver talks to a frontend server over HTTP. Communications are secured with symmetric encryption using the cryptography package's Fernet scheme, so you'll need a pre-shared key that both Authnzerver and your frontend server know.

See *the HTTP API docs* for details on how to call Authnzerver from a frontend service.

CHAPTER 1

Installation

1.1 Installing with pip

Install authnzerver (preferably in a virtualenv):

```
(venv)$ pip install authnzerver
```

1.2 Installing the latest version from Github

To install the latest version (may be unstable at times):

```
$ git clone https://github.com/waqasbhatti/authnzerver
$ cd authnzerver
$ python setup.py install
$ # or use pip install . to install requirements automatically
$ # or use pip install -e . to install in develop mode along with requirements
```

1.3 Installing the container from Docker Hub

Pull the image:

```
docker pull waqasbhatti/authnzerver:latest
```

CHAPTER 2

Using the server

See *the server configuration and usage docs* on how to configure the server with environment variables or command-line options, and run it either as a Docker container or as script executable from the Python package.

2.1 Quick start

If you have authnzerver installed as a Python package in an activated virtualenv:

```
authnzrv --autosetup --basedir=$(PWD)
```

If you're running it as a Docker container:

```
docker run -p 13431:13431 -v $(PWD):/home/authnzerver/basedir \
  --rm -it waqasbhatti/authnzerver:latest \
  --autosetup --basedir=/home/authnzerver/basedir
```

2.1.1 Running the server

This page describes how to configure and launch the server.

Configuring the authnzerver

The server is configurable via either environment variables or command-line options. You can also specify *how* to load these items if special handling is required. See the *Retrieving configuration items* section below for details.

To set a command line option, use `--option=value`.

To use environment variables for configuration, add them to the shell environment before the server starts, or add them to an `.env` file and provide its location with `--envfile` command-line parameter.

At a minimum, you must provide:

- a random pre-shared secret key as an environmental variable: `AUTHNZERVER_SECRETKEY` or as a command-line option: `--secret`.
- a random salt value for hashing personally identifiable information in the authnzerver logs as an environmental variable: `AUTHNZERVER_PIISALT` or as a command-line option: `--piisalt`.
- an SQLAlchemy database URL to indicate where the local authentication DB is. This should be in the form discussed in the SQLAlchemy docs as an environmental variable: `AUTHNZERVER_AUTHDB` or as a command-line option: `--authdb`.

If none of these required items are set, the authnzerver will not start.

To set up these required items in an interactive manner, provide a single command-line option `--autosetup`. The server will prompt you for admin credentials during start up, generate the pre-shared secret key and random salt, and initialize an authentication database at the SQLAlchemy URL you provide. Autogenerated defaults for these values can be used by hitting Enter at all the prompts.

Note: If you'd like to use PostgreSQL with authnzerver, make sure to also install the psycopg2 package so SQLAlchemy can talk to the database. Similarly, for MariaDB or MySQL, install a MySQL compatible package, for example, PyMySQL. Both of these packages are already included in the authnzerver Docker container, but are optional when authnzerver is installed as a Python package.

Setting the administrator password

Authnzerver sets up an account with a role of **superuser** when it first initializes its authentication database. If you use `--autosetup`, you will be asked for an email address and password to use for this account.

If you start the server directly, giving it all the required environment variables to do so (`AUTHNZERVER_SECRETKEY`, `AUTHNZERVER_PIISALT`, and `AUTHNZERVER_AUTHDB`), you will not be asked for admin account credentials. To provide these credentials in this case, use two more environment variables: `AUTHNZERVER_ADMIN_EMAIL`, and `AUTHNZERVER_ADMIN_PASSWORD`.

If these admin user credentials are not provided, a default admin user email address and random password will be generated and written to a file called `.authnzerver-admin-credentials` in server's base directory (by default: the directory where it starts from).

Warning: If you're running Authnzerver as a Docker container, the generated admin credentials file will be in the `/home/authnzerver/basedir` directory inside the container. Make sure to copy this file over to your host machine if you want to save it, since the container filesystem is ephemeral.

List of all configuration items

cmdline: `--allowedhosts`, env: `AUTHNZERVER_ALLOWEDHOSTS`

The allowed HTTP request header "Host" values that the server will respond to. Separate values with semicolons. Specifying these helps prevent DNS-rebinding attacks. (*default:* `'localhost;127.0.0.1'`)

cmdline: None, env: `AUTHNZERVER_ADMIN_EMAIL`

The email address to use for the auto-generated admin user with a role of **superuser** upon first startup of the server. If this is not provided, and you did not use `--autosetup` as a command line argument either, a default email address will be generated and used.

cmdline: None, env: `AUTHNZERVER_ADMIN_PASSWORD`

The password to use for the auto-generated admin user with a role of **superuser** upon first startup of the server. If this is not provided, and you did not use `--autosetup` as a command line argument either, a random password will be generated and used.

cmdline: `--authdb`, env: `AUTHNZERVER_AUTHDB`

An SQLAlchemy database URL to indicate where the local authentication DB is. This should be in the form discussed in the SQLAlchemy docs.

cmdline: `--autosetup`, env: None

If this is True, will automatically generate an SQLite authentication database in the basedir, copy over `default-permissions-model.json` and `confvars.py` to the basedir for easy customization, and finally, generate the communications secret file and the PII salt file. (*default:* False).

cmdline: `--basedir`, env: `AUTHNZERVER_BASEDIR`

The base directory containing secret files and the auth DB. (*default:* directory the server is launched in)

cmdline: `--confvars`, env: None

Path to the file containing the configuration variables needed by the server and how to load them. (*default:* the `confvars.py` file shipped with authnzerver)

cmdline: `--debugmode`, env: `AUTHNZERVER_DEBUGMODE`

If set to `1`, will enable an `/echo` endpoint for debugging purposes. (*default:* 0)

cmdline: `--emailpass`, env: `AUTHNZERVER_EMAILPASS`

The password to use for login to the email server.

cmdline: `--emailport`, env: `AUTHNZERVER_EMAILPORT`

The SMTP port of the email server to use. (*default:* 25)

cmdline: `--emailsender`, env: `AUTHNZERVER_EMAILSENDER`

The account name and email address that the authnzerver will send from. (*default:* `Authnzerver <authnzerver@localhost>`)

cmdline: `--emailserver`, env: `AUTHNZERVER_EMAILSERVER`

The address of the email server to use. (*default:* `localhost`)

cmdline: `--emailuser`, env: `AUTHNZERVER_EMAILUSER`

The username to use for login to the email server. (*default*: user running the authnzrv executable)

cmdline: `--envfile`, env: None

Path to a file containing environ variables for testing/development.

cmdline: `--listen`, env: `AUTHNZERVER_LISTEN`

Bind to this address and serve content. (*default:* `127.0.0.1`)

cmdline: `--passpolicy`, env: `AUTHNZERVER_PASSPOLICY`

This sets the password policy enforced by the server. This includes:

1. the minimum number of characters required in the password
2. the maximum allowed string similarity (out of 100) between the password and unsafe items like the server's domain name, the user's own email address, or their full name
3. the maximum number of times any single character can appear in the password as a fraction of the total number of characters in the password
4. the minimum number of matches required against the Have I Been Pwned compromised passwords database.

This parameter is specified as key:value pairs separated by a semicolon. (*default:* `min_pass_length:12; max_unsafe_similarity:50; max_char_frequency:0.3; min_pwned_matches:25`)

cmdline: `--ratelimits`, env: `AUTHNZERVER_RATELIMITS`

This sets the rate limit policy for authnzerver actions. This parameter is specified as key:value pairs separated by a semicolon. Specify values for all actions (tied to the IP address of the frontend server's client) in the `ipaddr` key, user-tied actions (based on email/user_id/IP address) in the `user` key, session-tied actions (based on session_token/IP address) in the `session` key, and apikey-tied actions (based on session_token/IP address) in the `apikey` key. The `burst` key indicates how many requests will be allowed to come in before rate-limits start being enforced.

All values are in units of max requests allowed per minute. Set this parameter to the string 'none' to turn off rate-limiting entirely.

(*default:* `ipaddr:720; user:480; session:600; apikey:720; burst:150`).

Some individual API actions are more aggressively rate-limited per IP address by the authnzerver. Currently, these include (all values in requests/minute):

```
AGGRESSIVE_RATE_LIMITS = {
  "user-new": 5,
  "user-login": 10,
  "user-logout": 10,
  "user-edit": 10,
  "user-resetpass": 5,
  "user-changepass": 5,
  "user-sendemail-signup": 2,       # also rate-limited per email address
  "user-sendemail-forgotpass": 2,   # also rate-limited per email address
  "user-set-emailsent": 2,          # also rate-limited per email address
  "apikey-new": 30,
  "apikey-new-nosession": 30,
  "apikey-refresh-nosession": 30,
}
```

You may also override the rate-limit for an individual API action by specifying it as a key-value pair in this configuration variable. For example, to set a custom rate limit of 20 requests/minute for the `user-login` action, add `user-login:20` to the `ratelimits` configuration variable string.

cmdline: `--permissions`, env: `AUTHNZERVER_PERMISSIONS`

The JSON file containing the permissions model the server will enforce. (*default:* the permissions model JSON shipped with authnzerver)

cmdline: `--piisalt`, env: `AUTHNZERVER_PIISALT`

A random value used as a salt when SHA256 hashing personally identifiable information (PII), such as user IDs and session tokens, etc. for authnzerver logs.

cmdline: `--port`, env: `PORT` or `AUTHNZERVER_PORT`

Run the server on this TCP port. (*default:* 13431)

cmdline: `--secret`, env: `AUTHNZERVER_SECRET`

The shared secret key used to secure communications between authnzerver and any frontend servers.

cmdline: `--sessionexpiry`, env: `AUTHNZERVER_SESSIONEXPIRY`

This sets the session-expiry time in days. (*default:* 30)

cmdline: `--tlscertfile`, env: `AUTHNZERVER_TLSCERTFILE`

The TLS certificate to use. If this is provided along with the certificate key in the `--tlscertkey` option, the server will start in TLS-enabled mode.

cmdline: `--tlscertkey`, env: `AUTHNZERVER_TLSCERTKEY`

The TLS certificate's key to use. If this is provided along with the certificate in the `--tlscertfile` option, the server will start in TLS-enabled mode.

cmdline: `--userlocktime`, env: `AUTHNZERVER_USERLOCKTIME`

This sets the lockout time in seconds for failed user logins that exceed the maximum number of failed login tries. (*default:* 3600)

cmdline: `--userlocktries`, env: `AUTHNZERVER_USERLOCKTRIES`

This sets the maximum number of failed logins per user that triggers a temporary lock on their account. (*default:* 10)

cmdline: `--workers`, env: `AUTHNZERVER_WORKERS`

The number of background workers to use when processing requests. (*default:* 4)

Retrieving configuration items

The `confvars.py` file contains all the configuration items required by the authnzerver and also defines how to retrieve them. If you run `--autosetup`, this file will be copied to the base directory you specify. Running the authnzerver with a `--confvars=/path/to/authnzerver/basedir/confvars.py` can be used to override the default config retrieval methods used by authnzerver.

YOU MUST NOT STORE ANY SECRETS IN THIS FILE. It only defines which variables in the environment or command-line parameters to use when retrieving secrets and other config items, as well as methods of retrieving them.

Let's walk through some examples of customizing retrieval of a config parameter: the secret shared key that secures communications between authnzerver and a frontend webserver.

Open up the `confvars.py` file in your authnzerver base directory. Here's the `secret` entry in the main CONF dict:

```
'secret':{
    'env':'%s_SECRET' % ENVPREFIX,
    'cmdline':'secret',
    'type':str,
    'default':None,
    'help':('The shared secret key used to secure '
            'communications between authnzerver and any frontend servers.'),
    'readable_from_file':'string',
    'postprocess_value':None,
}
```

This means the server will look at an environmental variable called `AUTHNZERVER_SECRET`, falling back to the value provided in the `--secret` command line option. The `readable_from_file` key tells the server how to handle the value it retrieved from either of these two sources.

To indicate that the retrieved value is to be used directly, set `"readable_from_file" = False`.

To indicate that the retrieved value can either be: (i) used directly or, (ii) may be a path to a file and the actual value of the `secret` item is a string to be read from that file, set `"readable_from_file" = "string"`.

To indicate that the retrieved value is a URL and the authnzerver must fetch the actual secret from this URL, set:

```
"readable_from_file" = ("http",
                        {'method':'get',
                         'headers':{header dict},
                         'data':{param dict},
                         'timeout':5.0},
                         'string')
```

Finally, you can also tell the server to fetch a JSON and pick out a key in the JSON. See the docstring for *authnzerver.confload.get_conf_item()* for more details on the various ways to retrieve the actual item pointed to by the config variable key.

To make this example more concrete, if the authnzerver `secret` was stored as a GCP Secrets Manager item, you'd set some environmental variables like so:

```
GCP_SECMAN_URL=https://secretmanager.googleapis.com/v1/projects/abcproj/secrets/abc/
↪versions/z:access
GCP_AUTH_TOKEN=some-secret-token
```

Then change the `secret` dict item in CONF dict below to:

```
'secret':{
    'env':'GCP_SECMAN_URL',
    'cmdline':'secret',
    'type':str,
    'default':None,
    'help':('The shared secret key used to secure '
            'communications between authnzerver and any frontend servers.'),
    'readable_from_file':see below,
    'postprocess_value':'custom_decode.py::custom_b64decode',
}
```

The `readable_from_file` key would be set to something like:

```
"readable_from_file" = ("http",
                        {"method":"get",
                         "headers":{"Authorization":"Bearer [[GCP_AUTH_TOKEN]]",
                                    "Content-Type":"application/json",
                                    "x-goog-user-project": "abcproj"},
                         "data":None,
                         "timeout":5.0},
                        'json',
                        "payload.data")
```

This would then load the authnzerver `secret` directly from the Secrets Manager.

Notice that we used a path to a Python module and function for the `postprocess_value` key. This is because GCP's Secrets Manager base-64 encodes the data you put into it and we need to post-process the value we get back from the stored item's URL. This module looks like:

```
import base64

def custom_b64decode(input):
    return base64.b64decode(input.encode('utf-8')).decode('utf-8')
```

The function above will base-64 decode the value returned from the Secrets Manager and finally give us the `secret` value we need.

Launching the authnzerver

Running the executable from the Python package

After you've installed the `authnzerver` package from PyPI (preferably in an already-activated virtualenv), there will be an `authnzrv` executable in your path.

`authnzrv --help` will list all options available. See the section above for details on configuring the server with either environment variables or command-line options.

If you want to run authnzerver as a systemd service, there's an example systemd service file available, along with an environment conf file that can be used to set it up.

Running with Docker and docker-compose

See below for an example docker-compose.yml snippet to include authnzerver as a service.

```
volumes:
  authnzerver_basedir:

services:
  authnzerver:
    image: waqasbhatti/authnzerver:latest
    expose: [13431]
    volumes:
      - authnzerver_basedir:/home/authnzerver/basedir
    # optional health check
    healthcheck:
      test: ["CMD-SHELL", "curl --silent --fail http://localhost:13431/health || exit␣
↪1"]
      interval: 10s
      timeout: 5s
      retries: 3
    environment:
      AUTHNZERVER_ALLOWEDHOSTS: authnzerver;localhost
      AUTHNZERVER_AUTHDB: "sqlite:////home/authnzerver/basedir/.authdb.sqlite"
      AUTHNZERVER_BASEDIR: "/home/authnzerver/basedir"
      AUTHNZERVER_DEBUGMODE: 0
      AUTHNZERVER_LISTEN: "0.0.0.0"
      AUTHNZERVER_PORT: 13431
      AUTHNZERVER_SECRET:
      AUTHNZERVER_PIISALT:
      AUTHNZERVER_SESSIONEXPIRY: 30
      AUTHNZERVER_USERLOCKTRIES: 10
      AUTHNZERVER_USERLOCKTIME: 3600
      AUTHNZERVER_PASSPOLICY: "min_pass_length:12;max_unsafe_similarity:50;max_char_
↪frequency:0.3;min_pwned_matches:25"
      AUTHNZERVER_WORKERS: 4
      AUTHNZERVER_EMAILSERVER: "localhost"
      AUTHNZERVER_EMAILPORT: 25
      AUTHNZERVER_EMAILUSER: "authnzerver"
      AUTHNZERVER_EMAILPASS:
      AUTHNZERVER_EMAILSENDER: "Authnzerver <authnzerver@localhost>"
      AUTHNZERVER_TLSCERTFILE:
      AUTHNZERVER_TLSCERTKEY:
      AUTHNZERVER_RATELIMITS: "ipaddr:720; user:480; session:600; apikey:720;␣
↪burst:150"
```

Some things to note about the snippet:

First, we're using an SQLite auth DB in the mounted authnzerver base directory. Another database can be specified here by using the appropriate SQLAlchemy database URL. On every start up, the authnzerver will recreate its database tables only if these don't exist already.

Note: For an example docker-compose file using PostgreSQL as the auth database, see example-docker-compose-postgres.yml in the authnzerver Github repository.

Next, the required `AUTHNZERVER_SECRET` and `AUTHNZERVER_PIISALT` environment variables are passed in from the host environment. Set these in your docker-compose `.env` file or in another manner as appropriate. Make sure to use strong random values here, for example:

```
python3 -c "import secrets, base64; [print('AUTHNZERVER_%s=\"%s\"' % (x, base64.
↪urlsafe_b64encode(secrets.token_bytes()).decode('utf-8'))) for x in ('SECRET',
↪'PIISALT')]"
```

Note that we're setting the listen address for the authnzerver to `0.0.0.0` so it can listen to requests on its container's external network interface.

Finally, we're setting the `AUTHNZERVER_ALLOWEDHOSTS` environment variable to include the DNS name of the container service generated by docker-compose: `authnzerver`, as well as `localhost`. The former allows requests from within the docker-compose network (i.e. other containers relying on authnzerver) to work correctly by using `http://authnzerver:13431` as the URL for the authnzerver. The latter lets the Docker and docker-compose health checks work correctly since these use cURL installed inside the container itself to ping the server periodically.

Running with Docker in development mode with auto-setup

First, pull the container from Docker Hub:

```
docker pull waqasbhatti/authnzerver:latest
```

Run it with the `--autosetup` option to set up a base directory, the auth DB, and the envfile. The commands below set up an empty base directory on your Docker host, mount it into the container as a volume, then tell authnzerver to use it for its base directory.

```
mkdir authnzerver-basedir
cd authnzerver-basedir
docker run -p 13431:13431 -v $(PWD):/home/authnzerver/basedir \
  --rm -it waqasbhatti/authnzerver:latest \
  --autosetup --basedir=/home/authnzerver/basedir
```

This will start an interactive session where you can set your auth DB and initial admin credentials:

```
[W 200625 17:42:21 autosetup:105] Enter a valid SQLAlchemy database URL to use for␣
↪the auth DB.
If you leave this blank and hit Enter, an SQLite auth DB
will be created in the base directory: /home/authnzerver/basedir
Auth DB URL [default: auto generated]:
[W 200625 17:42:23 autosetup:116] Enter the path to the permissions policy JSON file␣
↪to use.
If you leave this blank and hit Enter, the default permissions
policy JSON shipped with authnzerver will be used: /home/authnzerver/authnzerver/
↪default-permissions-model.json
```

(continues on next page)

(continued from previous page)

```
Permission JSON path [default: included permissions JSON]:
[W 200625 17:42:25 autosetup:134] No existing authentication DB was found, making a␣
↪new SQLite DB in authnzerver basedir: /home/authnzerver/basedir/.authdb.sqlite

Admin email address [default: authnzerver@localhost]:
Admin password [default: randomly generated]:
[W 200625 17:42:27 autosetup:214] Generated random admin password, credentials␣
↪written to: /home/authnzerver/basedir/.authnzerver-admin-credentials

[I 200625 17:42:27 autosetup:220] Generating server secret tokens...
[I 200625 17:42:27 autosetup:236] Generating server PII random salt...
[I 200625 17:42:27 autosetup:252] Copying default-permissions-model.json to basedir: /
↪home/authnzerver/basedir
[I 200625 17:42:27 autosetup:260] Copying confvars.py to basedir: /home/authnzerver/
↪basedir
[I 200625 17:42:27 autosetup:271] Generating an envfile: /home/authnzerver/basedir/.
↪env
[W 200625 17:42:27 main:216] Auto-setup complete, exiting...
[W 200625 17:42:27 main:219] Environment variables needed for the authnzerver to␣
↪start have been written to:

    /home/authnzerver/basedir/.env

    Edit this file as appropriate or add these environment variables to the shell␣
↪environment.
[W 200625 17:42:27 main:226] To run the authnzerver with this env file, your selected␣
↪auth DB, and the auto-setup generated secrets files in your selected authnzerver␣
↪basedir, start authnzerver with the following command:

    authnzrv --basedir="/home/authnzerver/basedir" --confvars="/home/authnzerver/
↪basedir/confvars.py" --envfile="/home/authnzerver/basedir/.env"
```

Edit the `.env` file that was created in your Docker host's authnzerver base directory. In particular, you want to set `AUTHNZERVER_LISTEN` variable to `0.0.0.0` for running authnzerver as a Docker container.

Start up authnzerver, using the command-line hints provided in autosetup:

```
docker run -p 13431:13431 -v $(PWD):/home/authnzerver/basedir \
  --rm -it waqasbhatti/authnzerver:latest \
  --confvars="/home/authnzerver/basedir/confvars.py" \
  --envfile="/home/authnzerver/basedir/.env"
```

If you do not want to use the envfile (e.g. in production), add the variables in it to your environment (e.g. in docker-compose) before launching the container, then use:

```
docker run -p 13431:13431 -v $(PWD):/home/authnzerver/basedir \
  --rm -it waqasbhatti/authnzerver:latest \
  --confvars="/home/authnzerver/basedir/confvars.py"
```

2.1.2 Permissions policy enforcement

This page describes the permissions model used by the Authnzerver.

Note: These docs are a work in progress, missing sections below will be filled in soon.

Permissions policy definition

Roles

Items

Actions

Visibilities

Limits

Item Policy

Role Policy

Required components

Using a permissions policy

The default permissions policy

2.1.3 Authnzerver HTTP API

This page describes the API of the Authnzerver.

All requests are composed of a Python dict containing request parameters. This is encoded to JSON, encrypted with the pre-shared key, base64-encoded, and then POSTed to the Authnzerver's / endpoint. The response is a base64-encoded string that must be base64-decoded, decrypted, and deserialized from JSON into a dict.

A request is of the form:

```
{'request': one of the request names below,
 'body': a dict containing the arguments for the request,
 'reqid': any integer or string (preferably a UUID) used to keep track
          of the request flow,
 'client_ipaddr': the IP address of the frontend server's
                  client to use in rate-limiting}
```

A response, when decrypted and deserialized to a dict, is of the form:

```
{'success': True or False,
 'response': dict containing the response items based on the request,
 'messages': a list of str containing informative/warning/error messages,
 'reqid': returns the same request ID provided to the authnzerver}
```

If the 'reqid' item in the authnzervr response dict doesn't match what you sent to the authnzerver, the response from the authnzerver MUST be rejected. A good way to populate 'reqid' in the authnzerver request dict is to generate one right at the beginning of the frontend server's HTTP response cycle, so it can remain the same between several authnzerver action requests. This allows you to track all authnzerver actions pertaining to a single response processing cycle of the frontend server.

The 'messages' item contains a list of messages that MAY be shared with a client of the frontend user to inform them about the state of the action request.

If the requested action failed, a 'failure_response' item will be present in the response dict. This SHOULD NOT be shared with a client of the frontend server because it likely contains the exact reason why something went wrong. Use these only to decide what to do on the frontend server.

If you share the 'failure_reason' item with a client of the frontend user, this may lead to an information leak when ambiguity is better, e.g. in the case of a password check, you usually don't want to disclose if the user account doesn't exist or the password was incorrect, instead responding with something like "Sorry, that username/password combination didn't work. Please try again."

The sections below describe the various available action request types, how to construct the `body` dict, and what to expect in the `response` dict.

Using the API Client

An authnzerver HTTP API client is available in the authnzerver Python package's *`authnzerver.client`* module. This contains a class that handles the encryption/decryption for the request-response cycle and includes both synchronous and asyncio-compatible request methods.

Example

```python
import asyncio
import base64
import secrets
import os

from authnzerver.client import Authnzerver


# this is the secret key shared between the authnzerver and our client
# best stored as an environment variable
SHARED_SECRET_KEY = os.environ.get("AUTHNZERVER_SECRET", None)

# if you haven't generated a secret key for authnzerver yet,
# here's how to do it
SHARED_SECRET_KEY = (
    base64.urlsafe_b64encode(secrets.token_bytes()).decode('utf-8')
)

# make a new authnzerver client object
client = Authnzerver(authnzerver_url="http://localhost:13431",
                     authnzerver_secret=SHARED_SECRET_KEY)

# fire a synchronous request
response = client.request("user-new",
                          {"email": "hello@test.org",
                           "password": "super-strong-password",
                           "full_name": "Test User",
                           "client_ipaddr": "1.2.3.4"})

# check if the request was successful
print(response.success)

# look at the response dict
print(response.response)
```

(continues on next page)

(continued from previous page)

```
# look at the messages that can be passed on to an end-user
print(response.messages)

# look at the failure_reason that should be used internally only
print(response.failure_reason)

# look at the headers of the response
print(response.headers)

# look at the HTTP status code of the response -- useful for HTTP 401
# or HTTP 429 responses from the authnzerver
print(response.status_code)

#
# the same request in an asynchronous style -- using asyncio.run
#

# a runner function to demonstrate await syntax
async def run_request():
    return await client.async_request("user-new",
                                      {"email": "hello2@test.org",
                                       "password": "superb-strong-password",
                                       "full_name": "Test User 2",
                                       "client_ipaddr": "1.2.3.4"})
# execute the asynchronous request
async_response = asyncio.run(run_request())
```

Constructing API Requests manually

Request example

```
import json
from base64 import b64encode
import random
from cryptography.fernet import Fernet
import requests

FERNET_KEY = "SHARED_SECRET_KEY"

def encrypt_request(request_dict, fernetkey):
    '''
    This encrypts the outgoing request to authnzerver.

    '''

    frn = Fernet(fernetkey)
    json_bytes = json.dumps(request_dict).encode()
    json_encrypted_bytes = frn.encrypt(json_bytes)
    request_base64 = b64encode(json_encrypted_bytes)
    return request_base64


# generate random request ID
reqid = random.randint(0,10000)
```

(continues on next page)

(continued from previous page)

```
# this is the request that will be sent to the authnzerver
req = {'request': request_type,
       'body': request_body,
       'reqid': reqid,
       'client_ipaddr': '1.1.1.1'}

# encrypt the request
encrypted_request = encrypt_request(req, FERNET_KEY)

# send the request and get the response
response = requests.post('http://127.0.0.1:13431', data=encrypted_request)
```

Response example

```
import json
from base64 import b64decode
from cryptography.fernet import Fernet, InvalidToken

FERNET_KEY = "SHARED_SECRET_KEY"

def decrypt_response(response_base64, fernetkey):
    '''
    This decrypts the incoming response from authnzerver.

    '''

    frn = Fernet(fernetkey)

    try:

        response_bytes = b64decode(response_base64)
        decrypted = frn.decrypt(response_bytes)
        return json.loads(decrypted)

    except InvalidToken:

        print('invalid response could not be decrypted')
        return None

    except Exception as e:

        print('could not understand incoming response')
        return None


# decrypt the response
decrypted_response_dict = decrypt_response(response.text, FERNET_KEY)
```

Session handling

`session-new`: Create a new session

Requires the following `body` items in a request:

- `ip_address` (str): the IP address of the client
- `user_agent` (str): the user agent of the client
- `user_id` (int): a user ID associated with the client
- `expires` (int): the number of days after which the token is invalid
- `extra_info_json` (dict): a dict containing arbitrary session associated information

Returns a `response` with the following items if successful:

- `session_token` (str): a session token suitable for use in a session cookie
- `expires` (str): a UTC datetime in ISO format indicating when the session expires

`session-exists`: Get info about an existing session

Requires the following `body` items in a request:

- `session_token` (str): the session token to check

Returns a `response` with the following items if successful:

- `session_info` (dict): a dict containing session info if it exists, None otherwise

`session-delete`: Delete a session

Requires the following `body` items in a request:

- `session_token` (str): the session token to delete

Returns a `response` with the following items:

- None. Check the `success` item in the returned dict.

`session-delete-userid`: Delete all sessions for a user ID

Requires the following `body` items in a request:

- `session_token` (str): the current session token
- `user_id` (int): a user ID associated with the client
- `keep_current_session` (bool): whether to keep the currently logged-in session

Returns a `response` with the following items:

- None. Check the `success` item in the returned dict.

`user-login`: Perform a user login action

Requires the following `body` items in a request:

- `session_token` (str): the session token associated with the `user_id`

- `email` (str): the email address associated with the `user_id`
- `password` (str): the password associated with the `user_id`

Returns a `response` with the following items if successful:

- `user_id` (int): a user ID associated with the logged-in user or None if login failed.
- `user_role` (str): the user's role.

`user-logout`: Perform a user logout action

Requires the following `body` items in a request:

- `user_id` (int): a user ID associated with the logged-in user or None if login failed.
- `session_token` (str): the session token associated with the `user_id`

Returns a `response` with the following items if successful:

- `user_id` (int): a user ID associated with the logged-in user or None if logout failed.

`user-passcheck`: Perform a user password check (requires an existing session)

Requires the following `body` items in a request:

- `session_token` (str): the session token associated with the `user_id`
- `password` (str): the password associated with the `user_id`

Returns a `response` with the following items if successful:

- `user_id` (int): a user ID associated with the logged-in user or None if password check failed.
- `user_role` (str): the user's role.

`user-passcheck-nosession`: Perform a user password check (without an existing session)

Requires the following `body` items in a request:

- `email` (str): the email address associated with the `user_id`
- `password` (str): the password associated with the `user_id`

Returns a `response` with the following items if successful:

- `user_id` (int): a user ID associated with the logged-in user or None if password check failed.
- `user_role` (str): the user's role.

User handling

`user-new`: Create a new user

Requires the following `body` items in a request:

- `full_name` (str): the user's full name
- `email` (str): the user's email address

- `password` (str): the user's password

Optional parameters are:

- `extra_info` (dict): arbitrary key-val items to store for this user. This is a good place to store user metadata like their organization, their avatar URL, their full address, etc.
- `verify_retry_wait` (int, default 6, minimum 1): The amount of time in hours the user must wait to retry a failed sign-up attempt. This situation arises if the user didn't get to their verification token email in time and it expired, or they sent back the incorrect token. The user must then wait for *verify_retry_wait* hours before they can try to sign up for an account again and get a new verification token via email.
- `system_id` (str): A (preferably random) string to use as the unique system ID for this user. A system ID is safer to use outside of the frontend/authnzerver system (e.g. by Javascript clients) than the `user_id` value, which is an integer primary key. If this is not provided, a UUIDv4 will be generated and used for the system ID.

Returns a `response` with the following items if successful:

- `user_email` (str): the user's email address
- `user_id` (int): the user's integer user ID (primary key in the `users` DB table)
- `system_id` (str): the user's system ID
- `send_verification` (bool): whether or not an email for user signup verification should be sent to this user. If the user has signed up already, but has not verified their account email address and *verify_retry_wait* hours have not yet passed, `send_verificiation` will be False.

`user-delete`: Delete an existing user

Requires the following `body` items in a request:

- `email` (str): the email address of the user
- `user_id` (int): the user ID of the user
- `password` (str): the password of the user to confirm account deletion if the user initiates this request themselves. optional if request was initiated by a superuser.

Returns a `response` with the following items if successful:

- `user_id` (str): the user ID of the just deleted user
- `email` (str): the email address of the just deleted user

`user-list`: List all users' or a single user's properties

Requires the following `body` items in a request:

- `user_id` (int): the user ID of the user to look up. If None, will list all users.

  Returns a `response` with the following items if successful:

- `user_info` (list of dicts): a list containing all user info as a dict per user. Each dict has the following items of information as dict keys: `user_id`, `system_id`, `full_name`, `email`, `is_active`, `created_on`, `user_role`, `last_login_try`, `last_login_success`, `extra_info`.

`user-lookup-email`: Look up a user's info given their email address

Requires the following `body` items in a request:

- `email` (str): the email address of the user to look up.

  Returns a `response` with the following items if successful:

- `user_info` (dict): a dict with the following items of information for the user as dict keys: `user_id`, `system_id`, `full_name`, `email`, `is_active`, `created_on`, `user_role`, `last_login_try`, `last_login_success`, `extra_info`.

`user-lookup-match`: Look up users by matching on a property

Requires the following `body` items in a request:

- `by` (str): the property to look up users by. This must be one of the following: `user_id`, `system_id`, `full_name`, `email`, `is_active`, `created_on`, `user_role`, `last_login_try`, `last_login_success`, `extra_info`.
- `match` (str or dict): the value to match against the stored value of the property. If this is a dict, then `by` must be equal to `extra_info`. The dict must be of the form `{'key':'value'}` to match one of the JSON items in the `extra_info` column of the `users` table.

Returns a `response` with the following items if successful:

- `user_info` (list): a list of dicts with the following items of information for each user as dict keys: `user_id`, `system_id`, `full_name`, `email`, `is_active`, `created_on`, `user_role`, `last_login_try`, `last_login_success`, `extra_info`.

`user-edit`: Edit a user's properties

Requires the following `body` items in a request:

- `user_id` (int): the user ID of the user initiating this request
- `user_role` (str): the role of the user initiating this request
- `session_token` (str): the session token of the user initiating this request
- `target_userid` (int): the user ID that will be the subject of this request
- `update_dict` (dict): the items to update. Keys that can be updated by all authenticated users are: `full_name`, `email`. Additional keys that can be updated by superusers only are: `is_active`, `user_role`.

Returns a `response` with the following items if successful:

- `user_info` (dict): dict containing the user's updated information

`user-lock`: Toggle a lock out for an existing user

Requires the following `body` items in a request:

- `user_id` (int): the user ID initiating this request
- `user_role` (str): the role of the user initiating this request
- `session_token` (str): the session token of the user initiating this request
- `target_userid` (int): the user ID of the subject of this request

- `action` (str): either `unlock` or `lock`

Returns a `response` with the following items if successful:

- `user_info` (dict): a dict with user info related to current lock and account status.

This request can only be initiated by users with the `superuser` role.

Password handling

`user-changepass`: Change an existing user's password

Requires the following `body` items in a request:

- `user_id` (int): the integer user ID of the user
- `session_token` (str): the current session token of the user
- `full_name` (str): the full name of the user
- `email` (str): the email address of the user
- `current_password` (str): the current password that will be changed
- `new_password` (str): the new password that will be used from now on

Returns a `response` with the following items if successful:

- `user_id` (int): the user ID of the user
- `email` (str): the email address of the user

`user-changepass-nosession`: Change an existing user's password (no session required)

Requires the following `body` items in a request:

- `user_id` (int): the integer user ID of the user
- `full_name` (str): the full name of the user
- `email` (str): the email address of the user
- `current_password` (str): the current password that will be changed
- `new_password` (str): the new password that will be used from now on

Returns a `response` with the following items if successful:

- `user_id` (int): the user ID of the user
- `email` (str): the email address of the user

`user-resetpass`: Reset a user's password

Requires the following `body` items in a request:

- `email_address` (str): the email address of the user whose password will be reset
- `new_password` (str): the new password provided by the user
- `session_token` (str): the session token of the session initiating the request

Returns a `response` with the following items:

- None, check the `success` key to see if the request succeeded.

Note that this API action deletes all of the user's existing sessions to make them log in again with the new password.

`user-resetpass-nosession`: Reset a user's password (no session required)

Requires the following `body` items in a request:

- `email_address` (str): the email address of the user whose password will be reset
- `new_password` (str): the new password provided by the user
- `required_active` (bool): if True, the user's *is_active* column value in the DB is required to be True. If False, the user's *is_active* column value in the DB is required to be False. Use this to require a specific user lock-out state before the password is reset. For example, if you always lock out users after their password-reset email token is verified and before they've entered a new password, set *required_active* to False.

Returns a `response` with the following items:

- None, check the `success` key to see if the request succeeded.

`user-validatepass`: Validate the user's password to see if it's insecure

Requires the following `body` items in a request:

- `password` (str): the password to validate
- `email` (str): the user's email address
- `full_name` (str): the user's full name

Optional items include:

- `min_pass_length` (int, default: 12): the minimum allowed password length in number of characters
- `max_unsafe_similarity` (int, default: 30): the maximum allowed string similarity (normalized to 100) between the user's password and their email address, their name, or the server's domain name.

Returns a `response` with the following items:

- `success` (bool): whether the password is OK.
- `messages` (str): any messages for the end-user that explain why their password was rejected if it was.

Authorization actions

These actions depend on a permissions policy that can be specified when the authnzerver starts up. This is a JSON file describing the roles, items, actions, item visibilities, and finally, the appropriate access rules and limits for each role. An example is the default-permissions-model.json shipped with the authnzerver package. If you don't specify a policy JSON as an environment variable or as a command line option, this default policy will be used.

`user-check-access`: Check if the specified user can access a specified item

Requires the following `body` items in a request:

- `user_id` (int): the user ID of the user attempting access.
- `user_role` (str): the role of the user attempting access.

- `action` (str): the action being checked.
- `target_name` (str): the item that the action is going to be applied to.
- `target_owner` (int): the user ID of the item's owner.
- `target_visibility` (str): the visibility of the item being accessed.
- `target_sharedwith` (str): a CSV list of user IDs that the item is shared with.

Returns a `response` with the following items if successful:

- None, check the value of `success`. `True` indicates the access was successfully granted, `False` indicates otherwise.

`user-check-limit`: Check if the specified user is over a specified limit

Requires the following `body` items in a request:

- `user_id` (int): the user ID of the user being checked for limit overage.
- `user_role` (str): the role of the user being checked.
- `limit_name` (str): the name of the limit to be checked.
- `value_to_check` (float, int): the amount to be checked against the limit value.

Returns a `response` with the following items if successful:

- None, check the value of `success`. `True` indicates the user is under the specified limit, `False` indicates otherwise.

Email actions

`user-sendemail-signup`: Send a verification email to a new user

Requires the following `body` items in a request:

- `email_address` (str): the email address of the new user
- `session_token` (str): the session token of the session initiating this request
- `created_info` (dict): the dict returned from the `user-new` request
- `server_name` (str): a name associated with the frontend server initiating the request (used in the email sent to the user)
- `server_baseurl` (str): the base URL of the frontend server initiating the request (used in the email sent to the user).
- `account_verify_url` (str): the URL fragment of the account verification endpoint on the frontend server initiating the request (used in the email sent to the user).
- `verification_token` (str): a time-stamped verification token generated by the frontend (this will be used as the verification token in the email text)
- `verification_expiry` (int): number of seconds after which the verification token will expire.

Returns a `response` with the following items if successful:

- `user_id` (int): the user ID of the user the email was sent to
- `email_address` (str): the email address the email was sent to

- `emailverify_sent_datetime` (str): the UTC datetime the email was sent on in ISO format

`user-sendemail-forgotpass`: Send a verification email to a user who forgot their password

Requires the following `body` items in a request:

- `email_address` (str): the email address of the new user
- `session_token` (str): the session token of the session initiating this request
- `created_info` (dict): the dict returned from the `user-new` request
- `server_name` (str): a name associated with the frontend server initiating the request (used in the email sent to the user)
- `server_baseurl` (str): the base URL of the frontend server initiating the request (used in the email sent to the user).
- `password_forgot_url` (str): the URL fragment of the forgot-password process initiation endpoint on the frontend server initiating the request (used in the email sent to the user).
- `verification_token` (str): a time-stamped verification token generated by the frontend (this will be used as the verification token in the email text)
- `verification_expiry` (int): number of seconds after which the verification token will expire.

Returns a `response` with the following items if successful:

- `user_id` (int): the user ID of the user the email was sent to
- `email_address` (str): the email address the email was sent to
- `emailforgotpass_sent_datetime` (str): the UTC datetime the email was sent on in ISO format

`user-set-emailverified`: Set the "verified" flag for a user completing sign-up

Requires the following `body` items in a request:

- `email` (str): the email address of the new user that has completed sign-up and the verification token challenge.

Returns a `response` with the following items if successful:

- `user_id` (int): the user ID of the newly signed-up user the email was sent to
- `user_role` (str): the user role of the newly signed-up user
- `is_active` (bool): True if the user is successfully tagged as verified.
- `emailverify_sent_datetime` (str): the UTC datetime the email was sent on in ISO format

`user-set-emailsent`: Set the sent datetime for a user sign-up or forgot-pass email

When some other way of emailing the user, external to authnzerver, is used to notify them about a signup verification or a forgot-password challenge, use this API call to set the corresponding time at which the emails were sent. This lets it do the right thing if someone tries to sign up for an account with the same email address later.

Requires the following `body` items in a request:

- `email` (str): the email address of the new user that has completed sign-up and the verification token challenge.
- `email_type` (str): either "signup" or "forgotpass".

Returns a `response` with the following items if successful:

- `user_id` (int): the user ID of the newly signed-up user the email was sent to
- `user_role` (str): the user role of the newly signed-up user
- `is_active` (bool): True if the user is successfully tagged as verified.
- `emailverify_sent_datetime` (str): the UTC datetime the email was sent on in ISO format
- `emailforgotpass_sent_datetime` (str): the UTC datetime the email was sent on in ISO format

API key actions

`apikey-new`: Create a new API key tied to a user ID, role, and existing user session

Requires the following `body` items in a request:

- `issuer` (str): the entity that will be designated as the API key issuer
- `audience` (str): the service this API key is being issued for (usually the host name of the frontend server)
- `subject` (list of str or str): the specific API endpoint(s) this API key is being issued for (usually a list of URIs for specific service endpoints)
- `apiversion` (int): the version of the API this key is valid for
- `expires_days` (int): the number of days that the API key will be valid for
- `not_valid_before` (int): the number of seconds after the current UTC time required before the API key becomes valid
- `user_id` (int): the user ID of the user that this API key is tied to
- `user_role` (str): the role of the user that this API key is tied to
- `ip_address` (str): the IP address that this API key is tied to
- `user_agent` (str): the user agent of the user creating the API key
- `session_token` (str): the session token of the user requesting this API key

Returns a `response` with the following items if successful:

- `apikey` (str): the API key information dict dumped to a JSON string
- `expires` (str): a UTC datetime in ISO format indicating when the API key expires

`apikey-verify`: Verify a session-tied API key's user ID, role, expiry, and token

Requires the following `body` items in a request:

- `apikey_dict` (dict): the decrypted and validated API key information dict from the frontend.
- `user_id` (int): the user ID of the user that this API key is tied to
- `user_role` (str): the role of the user that this API key is tied to

Returns a `response` with the following items:

- None, check the value of `success` to see if the API key is valid

`apikey-revoke`: Revoke a previously issued session-tied API key

Requires the following `body` items in a request:

- `apikey_dict` (dict): the decrypted and validated API key information dict from the frontend.
- `user_id` (int): the user ID of the target user whose API key is being revoked
- `user_role` (str): the role of the user that this API key is tied to

Returns a `response` with the following items:

- None, check the value of `success` to see if the API key revocation was successful

`apikey-new-nosession`: Create a new API key tied to a user ID, role, and IP address

See *`authnzerver.actions.apikey_nosession`* for notes on how to use no-session API keys.

Requires the following `body` items in a request:

- `issuer` (str): the entity that will be designated as the API key issuer
- `audience` (str): the service this API key is being issued for (usually the host name of the frontend server or the API service)
- `subject` (list of str or str): the specific API endpoint(s) this API key is being issued for (usually a list of URIs for specific service endpoints)
- `apiversion` (int): the version of the API this key is valid for
- `expires_seconds` (int): the number of seconds that the API key will be valid for
- `not_valid_before` (int): the number of seconds after the current UTC time required before the API key becomes valid
- `refresh_expires` (int): the number of seconds that the refresh token will be valid for
- `refresh_nbf` (int): the number of seconds after the current UTC time required before the refresh token become valid
- `user_id` (int): the user ID of the user that this API key is tied to
- `user_role` (str): the role of the user that this API key is tied to
- `ip_address` (str): the IP address that this API key is tied to

Returns a `response` with the following items if successful:

- `apikey` (str): the API key information dict dumped to a JSON string
- `expires` (str): a UTC datetime in ISO format indicating when the API key expires
- `refresh_token` (str): a refresh token to use when asking for a refreshed API key
- `refresh_token_expires` (str): a UTC datetime in ISO format indicating when the refresh token expires

`apikey-verify-nosession`: Verify a no-session API key's user ID, role, expiry, and token

Requires the following `body` items in a request:

- `apikey_dict` (dict): the decrypted and validated API key information dict from the frontend.
- `user_id` (int): the user ID of the user that this API key is tied to

- `user_role` (str): the role of the user that this API key is tied to

Returns a `response` with the following items:

- None, check the value of `success` to see if the API key is valid

`apikey-revoke-nosession`: Revoke a previously issued no-session API key

Requires the following `body` items in a request:

- `apikey_dict` (dict): the decrypted and validated API key information dict from the frontend.
- `user_id` (int): the user ID of the target user whose API key is being revoked
- `user_role` (str): the role of the user that this API key is tied to

Returns a `response` with the following items:

- None, check the value of `success` to see if the API key revocation was successful

`apikey-revokeall-nosession`: Revoke all previously issued no-session API keys

Requires the following `body` items in a request:

- `apikey_dict` (dict): the decrypted and validated API key information dict from the frontend. A valid and unexpired API no-session is required to validate the all-keys revocation request.
- `user_id` (int): the user ID of the target user whose API key is being revoked
- `user_role` (str): the role of the user that this API key is tied to

Returns a `response` with the following items:

- `deleted_keys`, the number of API keys that were revoked for this user after this action.

`apikey-refresh-nosession`: Refresh a previously issued no-session API key

Requires the following `body` items in a request:

- `apikey_dict` (dict): the decrypted and validated API key information dict from the frontend.
- `user_id` (int): the user ID of the target user whose API key is being revoked
- `user_role` (str): the role of the user that this API key is tied to
- `refresh_token` (str): the refresh token of this API key
- `ip_address` (str): the current IP address of the user
- `expires_seconds` (int): the number of seconds that the API key will be valid for
- `not_valid_before` (int): the number of seconds after the current UTC time required before the API key becomes valid
- `refresh_expires` (int): the number of seconds that the refresh token will be valid for
- `refresh_nbf` (int): the number of seconds after the current UTC time required before the refresh token become valid

Returns a `response` with the following items:

- `apikey` (str): the API key information dict dumped to a JSON string

- `expires` (str): a UTC datetime in ISO format indicating when the API key expires
- `refresh_token` (str): a new refresh token to use when asking for a refreshed API key
- `refresh_token_expires` (str): a UTC datetime in ISO format indicating when the refresh token expires

Internal actions

These are actions that are meant only for internal use of a frontend server. Invoking these actions MUST NOT accept any direct end-user input or pass it on to the authnzerver because no permissions are checked.

`internal-user-edit`: Edit a user's information

Requires the following `body` items in a request:

- `target_userid` (int): the user ID to update
- `update_dict` (dict): a dict containing arbitrary user associated information to edit existing values in the columns of the users table.

  The `update_dict` cannot contain the following fields: user_id, system_id, password, emailverify_sent_datetime, emailforgotpass_sent_datetime, emailchangepass_sent_datetime, last_login_success, last_login_try, failed_login_tries, created_on. These are tracked in other action functions and should not be changed directly. This helps keep the user database consistent.

  If `extra_info` is one of the items in `update_dict`, the `extra_info` JSON field in the database will be updated with the dict in `update_dict['extra_info']`. To delete an item from the database `extra_info` JSON field, pass in the special value of `"__delete__"` in `update_dict['extra_info']` for that item.

Returns a `response` with the following items if successful:

- `user_info` (dict): all user information with the updates included.

`internal-session-edit`: Edit an existing user session

Requires the following `body` items in a request:

- `target_session_token` (str): the session token to update
- `update_dict` (dict): a dict containing arbitrary session associated information to add to, edit existing items, or delete items from the `extra_info_json` column of the sessions table. The `extra_info_json` field in the database will be updated with the info in `update_dict`. To delete an item from `extra_info_json`, pass in the special value of `"__delete__"` in `update_dict` for that item.

Returns a `response` with the following items if successful:

- `session_info` (dict): all session related information with the updates included.

`internal-user-lock`: Lock/unlock a user

Requires the following `body` items in a request:

- `target_userid` (int): the user ID to lock/unlock
- `action` (str): the action to perform, one of: {'unlock','lock'}

Returns a `response` with the following items if successful:

- `user_info` (dict): user information including the current state of the `is_active` database column

`internal-user-delete`: Delete a user

Requires the following `body` items in a request:

- `target_userid` (int): the user ID to delete

Returns a `response` with the following items if successful:

- `user_id` (int): user ID of the user that was deleted `is_active` database column

2.1.4 authnzerver

authnzerver package

Subpackages

authnzerver.actions package

This contains functions to drive auth actions.

Submodules

authnzerver.actions.access module

This contains functions to apply access control.

`authnzerver.actions.access.`**`check_user_access`**(*payload: dict*, *raiseonfail: bool = False*, *override_permissions_json: str = None*, *override_authdb_path: str = None*, *config: types.SimpleNamespace = None*) → dict

Checks for user access to a specified item based on a permissions policy.

Parameters

- **`payload`** (*`dict`*) – This is the input payload dict. Required items:
  - user_id: int
  - user_role: str
  - action: str
  - target_name: str
  - target_owner: int
  - target_visibility: str
  - target_sharedwith: str

  In addition to these items received from an authnzerver client, the payload must also include the following keys (usually added in by a wrapping function):

  - reqid: int or str
  - pii_salt: str

- **raiseonfail** (*bool*) – If True, will raise an Exception if something goes wrong.
- **override_permissions_json** (*str or None*) – If given as a str, is the alternative path to the permissions JSON to load and use for this request. Normally, the path to the permissions JSON has already been specified as a process-local variable by the main authnzerver start up routines. If you want to use some other permissions model JSON (e.g. for testing), provide that here.

  Note that we load the permissions JSON from disk every time we need to take a decision. This might be a bit slower, but allows for much faster policy changes by just changing the permissions JSON file and not having to restart the authnzerver.
- **override_authdb_path** (*str or None*) – If given as a str, is the alternative path to the auth DB.
- **config** (*SimpleNamespace object or None*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns

The dict returned is of the form:

```
{'success': True or False,
 'messages': list of str messages if any}
```

Return type dict

authnzerver.actions.access.**check_user_limit**(*payload: dict*, *raiseonfail: bool = False*, *override_permissions_json: str = None*, *override_authdb_path: str = None*, *config: types.SimpleNamespace = None*) → dict

Applies a specified limit to an item based on a permissions policy.

Parameters

- **payload** (*dict*) – This is the input payload dict. Required items:
  - user_id: int
  - user_role: str
  - limit_name: str
  - value_to_check: any

  In addition to these items received from an authnzerver client, the payload must also include the following keys (usually added in by a wrapping function):
  - reqid: int or str
  - pii_salt: str
- **raiseonfail** (*bool*) – If True, will raise an Exception if something goes wrong.
- **override_permissions_json** (*str or None*) – If given as a str, is the alternative path to the permissions JSON to load and use for this request. Normally, the path to the permissions JSON has already been specified as a process-local variable by the main authnzerver start up routines. If you want to use some other permissions model JSON (e.g. for testing), provide that here.

  Note that we load the permissions JSON from disk every time we need to take a decision. This might be a bit slower, but allows for much faster policy changes by just changing the permissions JSON file and not having to restart the authnzerver.

- **override_authdb_path** (*str or None*) – If given as a str, is the alternative path to the auth DB.
- **config** (*SimpleNamespace object or None*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns

The dict returned is of the form:

```
{'success': True or False,
 'messages': list of str messages if any}
```

Return type dict

authnzerver.actions.admin module

This contains functions to drive admin related actions (listing users, editing users, change user roles).

`authnzerver.actions.admin.`**`edit_user`**(*payload: dict, raiseonfail: bool = False, override_permissions_json: str = None, override_authdb_path: str = None, config=None*) → dict

This edits users.

FIXME: add permissions checks to this instead of relying on a frontend to filter out users who aren't allowed to perform this action.

Parameters

- **payload** (*dict*) – This is the input payload dict. Required items:
  - user_id: int, user ID of an admin user or == target_userid
  - user_role: str, == 'superuser' or == target_userid user_role
  - session_token: str, session token of admin or target_userid token
  - target_userid: int, the user to edit
  - update_dict: dict, the update dict

  In addition to these items received from an authnzerver client, the payload must also include the following keys (usually added in by a wrapping function):

  - reqid: int or str
  - pii_salt: str

  Only these items can be edited:

  ```
  {'full_name', 'email',      <- by user and superuser
   'is_active','user_role', 'email_verified'}  <- by superuser only
  ```

  User IDs 2 and 3 are reserved for the system-wide anonymous and locked users respectively, and can't be edited.

- **raiseonfail** (*bool*) – If True, will raise an Exception if something goes wrong.
- **override_permissions_json** (*str or None*) – If given as a str, is the alternative path to the permissions JSON to load and use for this request. Normally, the path to the

permissions JSON has already been specified as a process-local variable by the main authnzerver start up routines. If you want to use some other permissions model JSON (e.g. for testing), provide that here.

Note that we load the permissions JSON from disk every time we need to take a decision. This might be a bit slower, but allows for much faster policy changes by just changing the permissions JSON file and not having to restart the authnzerver.

- **override_authdb_path** (*str or None*) – If given as a str, is the alternative path to the auth DB.
- **config** (*SimpleNamespace object or None*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns

The dict returned is of the form:

```
{'success': True or False,
 'user_info': dict, with new user info,
 'messages': list of str messages if any}
```

Return type dict

`authnzerver.actions.admin.`**`get_user_by_email`**(*payload: dict*, *raiseonfail: bool = False*, *override_authdb_path: str = None*, *config: types.SimpleNamespace = None*) → dict

This gets a user's information using their email address.

FIXME: add permissions checks to this instead of relying on a frontend to filter out users who aren't allowed to perform this action.

Parameters

- **payload** (*dict*) – This is the input payload dict. Required items:
  - email: str

  In addition to these items received from an authnzerver client, the payload must also include the following keys (usually added in by a wrapping function):
  - reqid: int or str
  - pii_salt: str
- **raiseonfail** (*bool*) – If True, will raise an Exception if something goes wrong.
- **override_authdb_path** (*str or None*) – If given as a str, is the alternative path to the auth DB.
- **config** (*SimpleNamespace object or None*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns

The dict returned is of the form:

```
{'success': True or False,
 'user_info': a user info dict,
 'messages': list of str messages if any}
```

The user info dict will contain the following items:

```
{'user_id','system_id', 'full_name', 'email',
 'is_active','created_on','user_role',
 'last_login_try','last_login_success'}
```

Return type dict

`authnzerver.actions.admin.`**`internal_edit_user`**(*payload: dict*, *raiseonfail: bool = False*, *override_authdb_path: str = None*, *config: types.SimpleNamespace = None*) → dict

Handles editing users. Meant for use internally in a frontend server.

Parameters

- **`payload`** (*`dict`*) – The input payload dict. Required items:
  - target_userid: int, the user to edit
  - update_dict: dict, the changes to make, with each key being a column value to change in the *users* table.

  update_dict cannot contain the following fields: user_id, system_id, password, emailverify_sent_datetime, emailforgotpass_sent_datetime, emailchangepass_sent_datetime, last_login_success, last_login_try, failed_login_tries, created_on, and last_updated. These are tracked in other action functions and should not be changed directly.

  If *update_dict* contains the *extra_info* field, this JSON field in the database will be updated with the info in *extra_info*. To delete an item from *extra_info*, pass in the special value of "__delete__" in *extra_info* for that item.

  In addition to these items received from an authnzerver client, the payload must also include the following keys (usually added in by a wrapping function):
  - reqid: int or str
  - pii_salt: str
- **`raiseonfail`** (*`bool`*) – If True, and something goes wrong, this will raise an Exception instead of returning normally with a failure condition.
- **`override_authdb_path`** (*`str or None`*) – The SQLAlchemy database URL to use if not using the default auth DB.
- **`config`** (*`SimpleNamespace object or None`*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns Returns a dict containing the new user information.

Return type dict

`authnzerver.actions.admin.`**`internal_toggle_user_lock`**(*payload: dict*, *raiseonfail: bool = False*, *override_authdb_path: str = None*, *config: types.SimpleNamespace = None*) → dict

Locks/unlocks user accounts.

This version of the function should only be run internally (i.e. not called by a client). The use-case is automatically locking user accounts if there are too many incorrect password attempts. The lock can be permanent or temporary.

Parameters

- **payload** (*dict*) – This is the input payload dict. Required items:
  - target_userid: int, the user to lock/unlock
  - action: str {'unlock','lock'}

  In addition to these items received from an authnzerver client, the payload must also include the following keys (usually added in by a wrapping function):
  - reqid: int or str
  - pii_salt: str
- **raiseonfail** (*bool*) – If True, will raise an Exception if something goes wrong.
- **override_authdb_path** (*str or None*) – If given as a str, is the alternative path to the auth DB.
- **config** (*SimpleNamespace object or None*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns

The dict returned is of the form:

```
{'success': True or False,
 'user_info': dict, with new user info,
 'messages': list of str messages if any}
```

Return type dict

`authnzerver.actions.admin.`**`list_users`**(*payload: dict, raiseonfail: bool = False, override_authdb_path: str = None, override_permissions_json: str = None, config: types.SimpleNamespace = None*) → dict

This lists users.

FIXME: add permissions checks to this instead of relying on a frontend to filter out users who aren't allowed to perform this action.

Parameters

- **payload** (*dict*) – This is the input payload dict. Required items:
  - user_id: int or None. If None, all users will be returned

  In addition to these items received from an authnzerver client, the payload must also include the following keys (usually added in by a wrapping function):
  - reqid: int or str
  - pii_salt: str
- **raiseonfail** (*bool*) – If True, will raise an Exception if something goes wrong.
- **override_authdb_path** (*str or None*) – If given as a str, is the alternative path to the auth DB.
- **override_permissions_json** (*str or None*) – If given as a str, is the alternative path to the permissions JSON to use.
- **config** (*SimpleNamespace object or None*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns

The dict returned is of the form:

```
{'success': True or False,
 'user_info': list of dicts, one per user,
 'messages': list of str messages if any}
```

The dicts per user will contain the following items:

```
{'user_id','full_name', 'email',
 'is_active','created_on','user_role',
 'last_login_try','last_login_success'}
```

Return type dict

`authnzerver.actions.admin.`**`lookup_users`**(*payload: dict*, *raiseonfail: bool = False*, *override_authdb_path: str = None*, *config: types.SimpleNamespace = None*) → dict

This looks up users by a given property.

FIXME: add permissions checks to this instead of relying on a frontend to filter out users who aren't allowed to perform this action.

Valid properties are all the columns in the users table, except for the password column.

Parameters

- **`payload`** (*`dict`*) – This is the input payload dict. Required items:
  - by (str): the property column to use to look up the user by
  - match (object): the required value of the property. Note that in most cases, this will be coerced to a string to compare it to the database value.

  If by == 'extra_info', then match must be a dict of the form:

  {'extra_info_key': extra_info_value}

  to match one or more keys inside the extra_info JSON column to the specified value.

  In addition to these items received from an authnzerver client, the payload must also include the following keys (usually added in by a wrapping function):

  - reqid: int or str
  - pii_salt: str
- **`raiseonfail`** (*`bool`*) – If True, will raise an Exception if something goes wrong.
- **`override_authdb_path`** (*`str or None`*) – If given as a str, is the alternative path to the auth DB.
- **`config`** (*`SimpleNamespace object or None`*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns

The dict returned is of the form:

```
{'success': True or False,
 'user_info': a user info dict,
 'messages': list of str messages if any}
```

The user info dict will contain the following items:

```
{'user_id','system_id', 'full_name', 'email',
 'is_active','created_on','user_role',
 'last_login_try','last_login_success'}
```

Return type dict

`authnzerver.actions.admin.`**`toggle_user_lock`**(*payload: dict*, *raiseonfail: bool = False*, *override_authdb_path: str = None*, *config: types.SimpleNamespace = None*) → dict

Locks/unlocks user accounts.

Can only be run by superusers and is suitable for use when called from a frontend.

Parameters

- **`payload`** (*`dict`*) – This is the input payload dict. Required items:
  - user_id: int, user ID of a superuser
  - user_role: str, == 'superuser'
  - session_token: str, session token of superuser
  - target_userid: int, the user to lock/unlock
  - action: str {'unlock','lock'}

  In addition to these items received from an authnzerver client, the payload must also include the following keys (usually added in by a wrapping function):
  - reqid: int or str
  - pii_salt: str
- **`raiseonfail`** (*`bool`*) – If True, will raise an Exception if something goes wrong.
- **`override_authdb_path`** (*`str or None`*) – If given as a str, is the alternative path to the auth DB.
- **`config`** (*`SimpleNamespace object or None`*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns

The dict returned is of the form:

```
{'success': True or False,
 'user_info': dict, with new user info,
 'messages': list of str messages if any}
```

Return type dict

`authnzerver.actions.admin.`**`user_info_columns`**(*table: sqlalchemy.sql.expression.table*) → Tuple

Returns the column expression for all required info retrieved by a user lookup.

table is the users SQLAlchemy table object. Required to preserve type information for the columns.

authnzerver.actions.apikey module

This contains functions to drive API key related auth actions.

`authnzerver.actions.apikey.`**`issue_apikey`**(*payload: dict*, *raiseonfail: bool = False*, *override_authdb_path: str = None*, *override_permissions_json: str = None*, *config: types.SimpleNamespace = None*) → dict

Issues a new API key.

Parameters

- **`payload`** (*`dict`*) – The payload dict must have the following keys:
  - issuer: str, the entity that will be designated as the API key issuer
  - audience: str, the service this API key is being issued for
  - subject: str, the specific API endpoint API key is being issued for
  - apiversion: int or str, the API version that the API key is valid for
  - expires_days: int, the number of days after which the API key will expire
  - not_valid_before: float or int, the amount of seconds after utcnow() when the API key becomes valid
  - user_id: int, the user ID of the user requesting the API key
  - user_role: str, the user role of the user requesting the API key
  - ip_address: str, the IP address to tie the API key to
  - user_agent: str, the browser user agent requesting the API key
  - session_token: str, the session token of the user requesting the API key
- **`raiseonfail`** (*`bool`*) – If True, will raise an Exception if something goes wrong.
- **`override_authdb_path`** (*`str or None`*) – If given as a str, is the alternative path to the auth DB.
- **`override_permissions_json`** (*`str or None`*) – If given as a str, is the alternative path to the permissions JSON to use. This is used to check if the user_id is allowed to actually request an API key.
- **`config`** (*`SimpleNamespace object or None`*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns

The dict returned is of the form:

```
{'success': True or False,
 'apikey': apikey dict,
 'expires': expiry datetime in ISO format,
 'messages': list of str messages if any}
```

Return type dict

Notes

API keys are tied to an IP address and client header combination.

This function will return a dict with all the API key information. This entire dict should be serialized to JSON, encrypted and time-stamp signed by the frontend as the final "API key", and finally sent back to the client.

`authnzerver.actions.apikey.`**`revoke_apikey`**(*payload: dict*, *raiseonfail: bool = False*, *override_authdb_path: str = None*, *override_permissions_json: str = None*, *config: types.SimpleNamespace = None*) → dict

Revokes an API key.

Parameters

- **`payload`** (*`dict`*) – This dict contains the following keys:
  - apikey_dict: the decrypted and verified API key info dict from the frontend.
  - user_id: the user ID of the person revoking this key. Only superusers or staff can revoke an API key that doesn't belong to them.
  - user_role: the user ID of the person revoking this key. Only superusers or staff can revoke an API key that doesn't belong to them.
- **`raiseonfail`** (*`bool`*) – If True, will raise an Exception if something goes wrong.
- **`override_authdb_path`** (*`str or None`*) – If given as a str, is the alternative path to the auth DB.
- **`override_permissions_json`** (*`str or None`*) – If given as a str, is the alternative path to the permissions JSON to use. This is used to check if the user_id is allowed to actually revoke ("delete") an API key.
- **`config`** (*`SimpleNamespace object or None`*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns

The dict returned is of the form:

```
{'success': True if API key was revoked and False otherwise,
 'messages': list of str messages if any}
```

Return type dict

`authnzerver.actions.apikey.`**`verify_apikey`**(*payload: dict*, *raiseonfail: bool = False*, *override_authdb_path: str = None*, *override_permissions_json: str = None*, *config: types.SimpleNamespace = None*) → dict

Checks if an API key is valid.

Parameters

- **`payload`** (*`dict`*) – This dict contains a single key:
  - apikey_dict: the decrypted and verified API key info dict from the frontend.
  - user_id: the user ID of the person wanting to verify this key.
  - user_role: the user role of the person wanting to verify this key.
- **`raiseonfail`** (*`bool`*) – If True, will raise an Exception if something goes wrong.

- **override_authdb_path** (*str or None*) – If given as a str, is the alternative path to the auth DB.
- **override_permissions_json** (*str or None*) – If given as a str, is the alternative path to the permissions JSON to use. This is used to check if the user_id is allowed to actually verify ("read") an API key.
- **config** (*SimpleNamespace object or None*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns

The dict returned is of the form:

```
{'success': True if API key is OK and False otherwise,
 'messages': list of str messages if any}
```

Return type dict

authnzerver.actions.apikey_nosession module

This contains functions to drive API key related auth actions.

API keys generated by this module do not require existing user sessions, so are useful for backend API services. They are shorter-lived than API keys tied to sessions, so come with a refresh token, which can be used to fetch a new no-session API key.

The workflow to use here is:

1. Hit the login endpoint. Make sure the endpoint uses XSRF protection. For Tornado generated pages, this is fairly easy: use `xsrf_form_html()` in a template to add a form field for the token. An AJAX call can pick this up from the form and send it in the POST request as the `_xsrf` parameter. For an SPA, this is more difficult. For a Tornado app, the static HTML of the SPA endpoint can be served using a `StaticFileHandler` that makes sure to set the `_xsrf` cookie (session-only scoped) by calling `self.xsrf_cookie` in the `initialize()` or `prepare()` method (this sets the cookie as a side-effect). That way, any response back from the SPA endpoint will have the `_xsrf` cookie (this is not HttpOnly), and the POST request can then read the cookie and send it back as verification in the request header or POST request params.
2. After login is verified, run the `issue_apikey()` function below. This returns an API key and a refresh token. The API key should have an expiry no longer than 15 minutes (or about the time needed to process a single API call). The refresh token has a longer expiry time (no longer than 24 hours or maybe the actual session lifetime) to allow for the user coming back and having to fetch a new API key. The refresh token is effectively a password, and the authnzerver stores it as such, generating a random 32-byte token, then using Argon2-ID to hash and store it in the DB. To verify it, we run the Argon2-ID hash as we do for passwords.
3. For SPAs, send back the API key and its expiry date in the response body, and set the refresh token as an HttpOnly, Secure cookie, with the TTL set to the expiry date of the refresh token. For API-only calls, the refresh token will have to be sent in the JSON response body. HTTPS is required in all cases.
4. The client can then use the no-session API key as normal until it expires. The verification of the API key itself can take place statelessly if it is decrypted correctly, has not expired, and the the claims in the decrypted key dict match the API endpoint's requirements. If further verification is required, the frontend can call `verify_apikey()`, which will check the API key against the one stored in the DB.
5. A bit before the API key expires, the client can hit an refresh-api-key endpoint on the frontend that is dedicated to refreshing the API key. The refresh token is presented in the cookie to the endpoint so the refresh request can be authenticated.

6. Use the `refresh_apikey()` function to refresh the API key. The refresh token presented is verified, and if that passes, a new API key + its expiry is sent back to the client, and a NEW refresh token MUST ALSO be set as an HttpOnly cookie if the client is an SPA (send back the refresh token in the response body if not).
7. To enforce logout or account deletion or lock, the API key and the refresh token are deleted from the `apikeys_nosession` table by the `revoke_apikey()` or `revoke_all_apikeys()` function below. The refresh token cookie MUST also be deleted from the client if it hits the logout/delete endpoint successfully.

References

- https://pragmaticwebsecurity.com/cheatsheets.html
- https://hasura.io/blog/best-practices-of-using-jwt-with-graphql/
- https://levelup.gitconnected.com/secure-jwts-with-backend-for-frontend-9b7611ad2afb

`authnzerver.actions.apikey_nosession.`**`issue_apikey`**(*payload: dict*, *raiseonfail: bool = False*, *override_authdb_path: str = None*, *override_permissions_json: str = None*, *config: types.SimpleNamespace = None*) → dict

Issues a new API key.

This version does not require a session.

Parameters

- **`payload`** (*`dict`*) – The payload dict must have the following keys:
  - issuer: str, the entity that will be designated as the API key issuer
  - audience: str, the service this API key is being issued for
  - subject: str, the specific API endpoint API key is being issued for
  - apiversion: int or str, the API version that the API key is valid for
  - expires_seconds: int, the number of seconds after which the API key expires
  - not_valid_before: float or int, the amount of seconds after utcnow() when the API key becomes valid
  - user_id: int, the user ID of the user requesting the API key
  - user_role: str, the user role of the user requesting the API key
  - ip_address: str, the IP address to tie the API key to
  - refresh_expires: int, the number of seconds after which the API key's refresh token expires
  - refresh_nbf: float or int, the amount of seconds after utcnow() after which the refresh token becomes valid
- **`raiseonfail`** (*`bool`*) – If True, will raise an Exception if something goes wrong.
- **`override_authdb_path`** (*`str or None`*) – If given as a str, is the alternative path to the auth DB.
- **`override_permissions_json`** (*`str or None`*) – If given as a str, is the alternative path to the permissions JSON to use. This is used to check if the user_id is allowed to actually request an API key.

- **`config`** (*`SimpleNamespace object or None`*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns

The dict returned is of the form:

```
{'success': True or False,
 'apikey': apikey dict,
 'expires': expiry datetime in ISO format,
 'refresh_token': refresh token str,
 'refresh_token_expires': expiry of refresh token in ISO format,
 'messages': list of str messages if any}
```

Return type dict

Notes

API keys are tied to an IP address, user ID, and role.

This function will return a dict with all the API key information. This entire dict should be serialized to JSON, encrypted and time-stamp signed by the frontend as the final "API key", and finally sent back to the client.

`authnzerver.actions.apikey_nosession.`**`refresh_apikey`**(*payload: dict, raiseonfail: bool = False, override_authdb_path: str = None, override_permissions_json: str = None, config: types.SimpleNamespace = None*)

Refreshes a no-session API key.

Requires a refresh token.

Parameters

- **`payload`** (*`dict`*) – This dict contains the following keys:
  - apikey_dict: the decrypted and verified API key info dict from the frontend.
  - user_id: the user ID of the person revoking this key. Only superusers or staff can revoke an API key that doesn't belong to them.
  - user_role: the user ID of the person revoking this key. Only superusers or staff can revoke an API key that doesn't belong to them.
  - refresh_token: the refresh token needed to refresh the API key
  - ip_address: the current IP address of the user
  - expires_seconds: int, the number of seconds after which the API key expires
  - not_valid_before: float or int, the amount of seconds after utcnow() when the API key becomes valid
  - refresh_expires: int, the number of seconds after which the API key's refresh token expires
  - refresh_nbf: float or int, the amount of seconds after utcnow() after which the refresh token becomes valid
- **`raiseonfail`** (*`bool`*) – If True, will raise an Exception if something goes wrong.

- **override_authdb_path** (*str or None*) – If given as a str, is the alternative path to the auth DB.
- **override_permissions_json** (*str or None*) – If given as a str, is the alternative path to the permissions JSON to use. This is used to check if the user_id is allowed to actually refresh ("delete" then "create") an API key.
- **config** (*SimpleNamespace object or None*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns

The dict returned is of the form:

```
{'success': True if API key was revoked and False otherwise,
 'messages': list of str messages if any}
```

Return type dict

`authnzerver.actions.apikey_nosession.`**`revoke_all_apikeys`**(*payload: dict, raiseonfail: bool = False, override_authdb_path: str = None, override_permissions_json: str = None, config: types.SimpleNamespace = None*) → dict

Revokes an API key.

This does not require a session, but does require a current valid and unexpired API key to revoke all API keys belonging to the specified user.

Parameters

- **payload** (*dict*) – This dict contains the following keys:
  - apikey_dict: the decrypted and verified API key info dict from the frontend.
  - user_id: the user ID of the person revoking this key. Only superusers or staff can revoke an API key that doesn't belong to them.
  - user_role: the user ID of the person revoking this key. Only superusers or staff can revoke an API key that doesn't belong to them.
- **raiseonfail** (*bool*) – If True, will raise an Exception if something goes wrong.
- **override_authdb_path** (*str or None*) – If given as a str, is the alternative path to the auth DB.
- **override_permissions_json** (*str or None*) – If given as a str, is the alternative path to the permissions JSON to use. This is used to check if the user_id is allowed to actually revoke ("delete") an API key.
- **config** (*SimpleNamespace object or None*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns

The dict returned is of the form:

```
{'success': True if API key was revoked and False otherwise,
 'messages': list of str messages if any}
```

Return type dict

`authnzerver.actions.apikey_nosession.`**`revoke_apikey`**(*payload: dict, raiseonfail: bool = False, override_authdb_path: str = None, override_permissions_json: str = None, config: types.SimpleNamespace = None*)

Revokes an API key.

This does not require a session.

Parameters

- **`payload`** (*`dict`*) – This dict contains the following keys:
  - apikey_dict: the decrypted and verified API key info dict from the frontend.
  - user_id: the user ID of the person revoking this key. Only superusers or staff can revoke an API key that doesn't belong to them.
  - user_role: the user ID of the person revoking this key. Only superusers or staff can revoke an API key that doesn't belong to them.
- **`raiseonfail`** (*`bool`*) – If True, will raise an Exception if something goes wrong.
- **`override_authdb_path`** (*`str or None`*) – If given as a str, is the alternative path to the auth DB.
- **`override_permissions_json`** (*`str or None`*) – If given as a str, is the alternative path to the permissions JSON to use. This is used to check if the user_id is allowed to actually revoke ("delete") an API key.
- **`config`** (*`SimpleNamespace object or None`*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns

The dict returned is of the form:

```
{'success': True if API key was revoked and False otherwise,
 'messages': list of str messages if any}
```

Return type dict

`authnzerver.actions.apikey_nosession.`**`verify_apikey`**(*payload: dict, raiseonfail: bool = False, override_authdb_path: str = None, override_permissions_json: str = None, config: types.SimpleNamespace = None*) → dict

Checks if an API key is valid.

This version does not require a session.

Parameters

- **`payload`** (*`dict`*) – This dict contains a single key:

  - apikey_dict: the decrypted and verified API key info dict from the frontend.
  - user_id: the user ID of the person wanting to verify this key.
  - user_role: the user role of the person wanting to verify this key.
- **raiseonfail** (*bool*) – If True, will raise an Exception if something goes wrong.
- **override_authdb_path** (*str or None*) – If given as a str, is the alternative path to the auth DB.
- **override_permissions_json** (*str or None*) – If given as a str, is the alternative path to the permissions JSON to use. This is used to check if the user_id is allowed to actually verify ("read") an API key.
- **config** (*SimpleNamespace object or None*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns

The dict returned is of the form:

```
{'success': True if API key is OK and False otherwise,
 'messages': list of str messages if any}
```

Return type dict

authnzerver.actions.email module

This contains functions to drive email-related auth actions.

`authnzerver.actions.email.`**`send_email`**(*sender: str, subject: str, text: str, recipients: Sequence[str], server: str, user: str, password: str, pii_salt: str, bcc: bool = False, port: int = 587*) → bool

This is a utility function to send email.

Parameters

- **sender** (*str*) – The name and email address of the entity sending the email in the following form:

```
"Sender Name <senderemail@example.com>"
```

- **subject** (*str*) – The subject of the email.
- **text** (*str*) – The text of the email.
- **recipients** (*list of str*) – A list of the email addresses to send the email to. Use either of the formats below for each email address:

```
"Recipient Name <recipient@example.com>"
"recipient@example.com"
```

- **server** (*str*) – The address of the email server to use.
- **user** (*str*) – The username to use when logging into the email server via SMTP.
- **password** (*str*) – The password to use when logging into the email server via SMTP.

- **pii_salt** (*str*) – The PII salt value passed in from a wrapping function. Used to censor personally identifying information in the logs emitted from this function.
- **bcc** (*bool or list of str*) – If True, will set the To: field in the email itself to "undisclosed-recipients" and send the email to all recipients such that none of them know who the message was sent to (effectively BCCs all the recipients). If this is set to a list of email addresses in RFC822 format as strings, will only BCC those email addresses. If this is set to False, the To: field in the email itself will contain the addresses of all recipients.
- **port** (*int*) – The SMTP port to use when logging into the email server via SMTP.

Returns **sent_ok** – Returns True if email sending succeeded. False otherwise.

Return type bool

authnzerver.actions.email.**send_forgotpass_verification_email**(*payload: dict*, *raiseonfail: bool = False*, *override_authdb_path: str = None*, *config: types.SimpleNamespace = None*) → dict

This actually sends the forgot password email.

Parameters

- **payload** (*dict*) – Keys expected in this dict from a client are:
  - email_address: str, the email address to send the email to
  - session_token: str, session token of the user being sent the email
  - server_name: str, the name of the frontend server
  - server_baseurl: str, the base URL of the frontend server
  - password_forgot_url: str, the URL fragment of the frontend forgot-password process initiation endpoint
  - verification_token: str, a verification token generated by frontend
  - verification_expiry: int, number of seconds after which the token expires

  In addition, the following items must be provided by a wrapper function to set up the email server.

  - emailuser
  - emailpass
  - emailserver
  - emailport
  - emailsender

  These can be provided as part of the payload as dict keys or as attributes in the SimpleNamespace object passed in the config kwarg. The config object will be checked first, and the payload items will override it.

  Finally, the payload must also include the following keys (usually added in by a wrapping function):

  - reqid: int or str
  - pii_salt: str

- **override_authdb_path** (*str or None*) – If given as a str, is the alternative path to the auth DB.
- **raiseonfail** (*bool*) – If True, will raise an Exception if something goes wrong.
- **config** (*SimpleNamespace object or None*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns Returns a dict containing the user_id, email_address, and the emailforgotpass_sent_datetime value if email was sent successfully.

Return type dict

`authnzerver.actions.email.`**`send_signup_verification_email`**(*payload: dict, raiseonfail: bool = False, override_authdb_path: str = None, config: types.SimpleNamespace = None*) → dict

Sends an account verification email.

Parameters

- **payload** (*dict*) – Keys expected in this dict from a client are:
  - email_address: str, the email address to send the email to
  - session_token: str, session token of the user being sent the email
  - created_info: dict, the dict returned by `users.auth_create_user()`
  - server_name: str, the name of the frontend server
  - server_baseurl: str, the base URL of the frontend server
  - account_verify_url: str, the URL fragment of the frontend verification endpoint
  - verification_token: str, a verification token generated by frontend
  - verification_expiry: int, number of seconds after which the token expires

  In addition, the following optional items must be provided by a wrapper function to set up the email server.

  - emailuser
  - emailpass
  - emailserver
  - emailport
  - emailsender

  These can be provided as part of the payload as dict keys or as attributes in the SimpleNamespace object passed in the config kwarg. The config object will be checked first, and the payload items will override it.

  Finally, the payload must also include the following keys (usually added in by a wrapping function):

  - reqid: int or str
  - pii_salt: str

- **override_authdb_path** (*str or None*) – If given as a str, is the alternative path to the auth DB.
- **raiseonfail** (*bool*) – If True, will raise an Exception if something goes wrong.
- **config** (*SimpleNamespace object or None*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns Returns a dict containing the user_id, email_address, and the emailverify_sent_datetime value if email was sent successfully.

Return type dict

authnzerver.actions.email.**set_user_email_sent**(*payload: dict*, *raiseonfail: bool = False*, *override_authdb_path: str = None*, *config: types.SimpleNamespace = None*) → dict

Sets the verify/forgot email sent flag & time for the newly created user.

This is useful when some other way of emailing the user to verify their sign up or their password forgot request is used, external to authnzerver. Use this function to let the authnzerver know that an email has been sent so it knows the correct move if someone tries to sign up for an account with the same email address later.

Parameters

- **payload** (*dict*) – This is a dict with the following key:
  - email, str
  - email_type, str: one of “signup”, “forgotpass”

  Finally, the payload must also include the following keys (usually added in by a wrapping function):
  - reqid: int or str
  - pii_salt: str
- **override_authdb_path** (*str or None*) – If given as a str, is the alternative path to the auth DB.
- **raiseonfail** (*bool*) – If True, will raise an Exception if something goes wrong.
- **config** (*SimpleNamespace object or None*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns Returns a dict containing the email address and email*_sent_datetime values if the sent-email notification was successfully set.

Return type dict

authnzerver.actions.email.**set_user_emailaddr_verified**(*payload: dict*, *raiseonfail: bool = False*, *override_authdb_path: str = None*, *config: types.SimpleNamespace = None*) → dict

Sets the verification status of the email address of the user.

This is called by the frontend after it verifies that the token challenge to verify the user’s email succeeded and has not yet expired. This will set the user_role to ‘authenticated’ (or the previous user role before locking) and the is_active column to True.

Parameters

- **`payload`** (*`dict`*) – This is a dict with the following key:
  - email

  Finally, the payload must also include the following keys (usually added in by a wrapping function):
  - reqid: int or str
  - pii_salt: str
- **`override_authdb_path`** (*`str or None`*) – If given as a str, is the alternative path to the auth DB.
- **`raiseonfail`** (*`bool`*) – If True, will raise an Exception if something goes wrong.
- **`config`** (*`SimpleNamespace object or None`*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns Returns a dict containing the user_id, is_active, and user_role values if verification status is successfully set.

Return type dict

authnzerver.actions.loginlogout module

This contains functions to log a user in and out.

`authnzerver.actions.loginlogout.`**`auth_user_login`**(*payload: dict*, *override_authdb_path: str = None*, *raiseonfail: bool = False*, *config: types.SimpleNamespace = None*) → dict

Logs a user in.

Login flow for frontend:

session cookie get -> check session exists -> check user login -> old session delete (no matter what) -> new session create (with actual user_id and other info now included if successful or same user_id = anon if not successful) -> done

The frontend MUST unset the cookie as well.

FIXME: update (and fake-update) the Users table with the last_login_try and last_login_success.

Parameters

- **`payload`** (*`dict`*) – The payload dict should contain the following keys:
  - session_token: str
  - email: str
  - password: str

  In addition to these items received from an authnzerver client, the payload must also include the following keys (usually added in by a wrapping function):
  - reqid: int or str
  - pii_salt: str

- **override_authdb_path** (*str or None*) – The SQLAlchemy database URL to use if not using the default auth DB.
- **raiseonfail** (*bool*) – If True, and something goes wrong, this will raise an Exception instead of returning normally with a failure condition.
- **config** (*SimpleNamespace object or None*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns Returns a dict containing the result of the password verification check.

Return type dict

authnzerver.actions.loginlogout.**auth_user_logout**(*payload: dict*, *override_authdb_path: str = None*, *raiseonfail: bool = False*, *config: types.SimpleNamespace = None*) → dict

Logs out a user.

Deletes the session token from the session store. On the next request (redirect from POST /auth/logout to GET /), the frontend will issue a new one.

The frontend MUST unset the cookie as well.

Parameters

- **payload** (*dict*) – The payload dict should contain the following keys:
  - session_token: str
  - user_id: int

  In addition to these items received from an authnzerver client, the payload must also include the following keys (usually added in by a wrapping function):
  - reqid: int or str
  - pii_salt: str
- **override_authdb_path** (*str or None*) – The SQLAlchemy database URL to use if not using the default auth DB.
- **raiseonfail** (*bool*) – If True, and something goes wrong, this will raise an Exception instead of returning normally with a failure condition.
- **config** (*SimpleNamespace object or None*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns Returns a dict containing the result of the password verification check.

Return type dict

authnzerver.actions.passcheck module

This contains functions to drive session-related auth actions.

authnzerver.actions.passcheck.**auth_password_check**(*payload: dict*, *override_authdb_path: str = None*, *raiseonfail: bool = False*, *config: types.SimpleNamespace = None*) → dict

This runs a password check given a session token and password.

Used to gate high-security areas or operations that require re-verification of the password for a user's existing session.

Parameters

- **`payload`** (*`dict`*) – This is a dict containing the following items:
  - session_token
  - password

  In addition to these items received from an authnzerver client, the payload must also include the following keys (usually added in by a wrapping function):
  - reqid: int or str
  - pii_salt: str
- **`override_authdb_path`** (*`str or None`*) – The SQLAlchemy database URL to use if not using the default auth DB.
- **`raiseonfail`** (*`bool`*) – If True, and something goes wrong, this will raise an Exception instead of returning normally with a failure condition.
- **`config`** (*`SimpleNamespace object or None`*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns Returns a dict containing the result of the password verification check.

Return type dict

`authnzerver.actions.passcheck.`**`auth_password_check_nosession`**(*payload: dict*, *override_authdb_path: str = None*, *raiseonfail: bool = False*, *config: types.SimpleNamespace = None*) → dict

This runs a password check given an email address and password.

Used to gate high-security areas or operations that require re-verification of the password for a user, without checking if they have a session.

Useful for APIs, where the 'password' is some API token.

Parameters

- **`payload`** (*`dict`*) – This is a dict containing the following items:
  - email
  - password

  In addition to these items received from an authnzerver client, the payload must also include the following keys (usually added in by a wrapping function):
  - reqid: int or str
  - pii_salt: str
- **`override_authdb_path`** (*`str or None`*) – The SQLAlchemy database URL to use if not using the default auth DB.
- **`raiseonfail`** (*`bool`*) – If True, and something goes wrong, this will raise an Exception instead of returning normally with a failure condition.

- **`config`** (*`SimpleNamespace object or None`*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns Returns a dict containing the result of the password verification check.

Return type dict

authnzerver.actions.passchange module

This contains functions to change passwords.

`authnzerver.actions.passchange.`**`change_user_password`**(*payload: dict, override_authdb_path: str = None, raiseonfail: bool = False, min_pass_length: int = 12, max_unsafe_similarity: int = 33, config: types.SimpleNamespace = None*) → dict

Changes the user's password.

Parameters

- **`payload`** (*`dict`*) – This is a dict with the following required keys:
  - user_id: int
  - session_token: str
  - full_name: str
  - email: str
  - current_password: str
  - new_password: str

  In addition to these items received from an authnzerver client, the payload must also include the following keys (usually added in by a wrapping function):

  - reqid: int or str
  - pii_salt: str
- **`override_authdb_path`** (*`str or None`*) – If given as a str, is the alternative path to the auth DB.
- **`raiseonfail`** (*`bool`*) – If True, will raise an Exception if something goes wrong.
- **`min_pass_length`** (*`int`*) – The minimum required character length of the password. The value provided in this kwarg will be overriden by the `passpolicy` attribute in the config object if that is passed in as well.
- **`max_unsafe_similarity`** (*`int`*) – The maximum ratio required to fuzzy-match the input password against the server's domain name, the user's email, or their name. The value provided in this kwarg will be overriden by the `passpolicy` attribute in the config object if that is passed in as well.
- **`config`** (*`SimpleNamespace object or None`*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns Returns a dict with the user's user_id and email as keys if successful.

Return type dict

Notes

This logs out the user from all of their other sessions.

`authnzerver.actions.passchange.`**`change_user_password_nosession`**(*payload: dict*, *override_authdb_path: str = None*, *raiseonfail: bool = False*, *min_pass_length: int = 12*, *max_unsafe_similarity: int = 33*, *config: types.SimpleNamespace = None*) → dict

Changes the user's password.

This version doesn't require an active session.

Parameters

- **`payload`** (*`dict`*) – This is a dict with the following required keys:
  - user_id: int
  - full_name: str
  - email: str
  - current_password: str
  - new_password: str

  In addition to these items received from an authnzerver client, the payload must also include the following keys (usually added in by a wrapping function):
  - reqid: int or str
  - pii_salt: str
- **`override_authdb_path`** (*`str or None`*) – If given as a str, is the alternative path to the auth DB.
- **`raiseonfail`** (*`bool`*) – If True, will raise an Exception if something goes wrong.
- **`min_pass_length`** (*`int`*) – The minimum required character length of the password. The value provided in this kwarg will be overriden by the `passpolicy` attribute in the config object if that is passed in as well.
- **`max_unsafe_similarity`** (*`int`*) – The maximum ratio required to fuzzy-match the input password against the server's domain name, the user's email, or their name. The value provided in this kwarg will be overriden by the `passpolicy` attribute in the config object if that is passed in as well.
- **`config`** (*`SimpleNamespace object or None`*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns Returns a dict with the user's user_id and email as keys if successful.

Return type dict

Notes

This logs out the user from all of their other sessions.

authnzerver.actions.passreset module

This contains functions to reset passwords.

`authnzerver.actions.passreset.`**`verify_password_reset`**(*payload: dict*, *raiseonfail: bool = False*, *override_authdb_path: str = None*, *min_pass_length: int = 12*, *max_unsafe_similarity: int = 33*, *config: types.SimpleNamespace = None*) → dict

Verifies a password reset request.

Parameters

- **`payload`** (*`dict`*) – This is a dict with the following required keys:
  - email_address: str
  - new_password: str
  - session_token: str

  In addition to these items received from an authnzerver client, the payload must also include the following keys (usually added in by a wrapping function):
  - reqid: int or str
  - pii_salt: str
- **`raiseonfail`** (*`bool`*) – If True, will raise an Exception if something goes wrong.
- **`override_authdb_path`** (*`str or None`*) – If given as a str, is the alternative path to the auth DB.
- **`min_pass_length`** (*`int`*) – The minimum required character length of the password.
- **`max_unsafe_similarity`** (*`int`*) – The maximum ratio required to fuzzy-match the input password against the server's domain name, the user's email, or their name.
- **`config`** (*`SimpleNamespace object or None`*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns Returns a dict containing a success key indicating if the user's password was reset.

Return type dict

`authnzerver.actions.passreset.`**`verify_password_reset_nosession`**(*payload: dict*, *raiseonfail: bool = False*, *override_authdb_path: str = None*, *min_pass_length: int = 12*, *max_unsafe_similarity: int = 33*, *config: types.SimpleNamespace = None*) → dict

Verifies a password reset request.

This version does not require an active session.

Parameters

- **`payload`** (*`dict`*) – This is a dict with the following required keys:
  - email_address: str
  - new_password: str
  - required_active: bool

  The *required_active* parameter can be used to check the required state of the *is_active* DB entry for the user before password reset is allowed to proceed. This is useful when user accounts are required to be locked when a successful password reset verification token is received by a frontend server.

  In addition to these items received from an authnzerver client, the payload must also include the following keys (usually added in by a wrapping function):
  - reqid: int or str
  - pii_salt: str
- **`raiseonfail`** (*`bool`*) – If True, will raise an Exception if something goes wrong.
- **`override_authdb_path`** (*`str or None`*) – If given as a str, is the alternative path to the auth DB.
- **`min_pass_length`** (*`int`*) – The minimum required character length of the password.
- **`max_unsafe_similarity`** (*`int`*) – The maximum ratio required to fuzzy-match the input password against the server's domain name, the user's email, or their name.
- **`config`** (*`SimpleNamespace object or None`*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns Returns a dict containing a success key indicating if the user's password was reset.

Return type dict

authnzerver.actions.passwords module

This contains functions for validating passwords.

`authnzerver.actions.passwords.`**`check_password_pwned`**(*password: str*, *email: str*, *reqid: str*, *pii_salt: str*, *min_matches: int = 25*) → tuple

Checks the password against the haveibeenpwned.com API.

https://haveibeenpwned.com/API/v3#PwnedPasswords

Parameters

- **`password`** (*`str`*) – The password to check against the haveibeenpwned.com API.
- **`email`** (*`str`*) – The email address of the user creating the account.
- **`reqid`** (*`int or str`*) – The request ID associated with this password validation request. Used to track and correlate these requests in logs.
- **`pii_salt`** (*`str`*) – The PII salt value passed in from a wrapping function. Used to censor personally identifying information in the logs emitted from this function.
- **`min_matches`** (*`int`*) – The minimum number of matches required in the matching set returned by the API to consider a password as compromised.

Returns (status, msg, sha1_suffix, all_matches) – If the password is considered to be compromised, returns "bad", msg for the first two elements in the tuple. Otherwise, returns "ok", "". If the API does not respond or there's an error, returns "unknown", "".

Return type tuple

`authnzerver.actions.passwords.`**`validate_input_password`**(*full_name: str*, *email: str*, *password: str*, *pii_salt: str*, *reqid: str*, *min_pass_length: int = 12*, *max_unsafe_similarity: int = 33*, *max_character_frequency: float = 0.3*, *min_pwned_matches: int = 25*, *config: types.SimpleNamespace = None*) → tuple

Validates user input passwords.

Password rules are:

1. must be at least min_pass_length characters (we'll truncate the password at 1024 characters since we don't want to store entire novels)
2. must not match within max_unsafe_similarity of their email or full_name
3. must not match within max_unsafe_similarity of the site's FQDN
4. must not have a single case-folded character take up more than 20% of the length of the password
5. must not be completely numeric
6. must not be in the top 10k passwords list

If all of the above pass, one last check is done:

7. must not be in the https://haveibeenpwned.com/Passwords database with at least *min_pwned_matches* matches

Parameters

- **full_name** (*str*) – The full name of the user creating the account.
- **email** (*str*) – The email address of the user creating the account.
- **password** (*str*) – The password of the user creating the account.
- **pii_salt** (*str*) – The PII salt value passed in from a wrapping function. Used to censor personally identifying information in the logs emitted from this function.
- **reqid** (*int or str*) – The request ID associated with this password validation request. Used to track and correlate these requests in logs.
- **min_pass_length** (*int*) – The minimum required character length of the password. The value provided in this kwarg will be overriden by the `passpolicy` attribute in the config object if that is passed in as well.
- **max_unsafe_similarity** (*int*) – The maximum ratio required to fuzzy-match the input password against the server's domain name, the user's email, or their name. The value provided in this kwarg will be overriden by the `passpolicy` attribute in the config object if that is passed in as well.
- **max_character_frequency** (*float*) – The maximum number of times a character can appear in the password as a fraction of the total number of characters in the password. Upper and lower case characters are counted separately.
- **min_pwned_matches** (*int*) – The minimum number of matches required in the matching set returned by the haveibeenpwned.com password compromise database API to consider a password as compromised.
- **config** (*SimpleNamespace object or None*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns **(password_ok, messages)** – *password_ok* is True if the password is OK to use and meets all specification, False otherwise. *messages* is a list of strings containing helpful messages on why the password was rejected (if it was) that can be passed to an end-user.

Return type tuple

`authnzerver.actions.passwords.`**`validate_password`**(*payload: dict*, *raiseonfail: bool = False*, *override_authdb_path: str = None*, *config: types.SimpleNamespace = None*) → dict

External interface to password validation.

Use this in a frontend server or client to validate any passwords sent by the end-user.

Parameters

- **payload** (*dict*) – This is a dict with the following required keys:
  - password: str
  - email: str
  - full_name: str

  The following keys are optional:
  - min_pass_length: int, default = 12
  - max_unsafe_similarity: int, default = 33
  - max_character_frequency: float, default = 0.3

– min_pwned_matches: int, default = 25

The *email* and *full_name* are required to check if the password is too similar to either of these items.

min_pass_length is the minimum number of characters required for the password. All passwords are capped at 256 characters. This value will be overriden by a value in the *config* object's *min_pass_length* attribute.

max_unsafe_similarity is the maximum ratio required to fuzzy-match the input password against the server's domain name, the user's email, or their name. This value will be overriden by a value in the *config* object's *max_unsafe_similarity* attribute.

max_character_frequency is the maximum ratio required to fuzzy-match the input password against the server's domain name, the user's email, or their name. The value provided in this kwarg will be overriden by the `passpolicy` attribute in the config object if that is passed in as well.

min_pwned_matches is the minimum number of matches required in the matching set returned by the haveibeenpwned.com password compromise database API to consider a password as compromised.

In addition to these items received from an authnzerver client, the payload must also include the following keys (usually added in by a wrapping function):

– reqid: int or str

– pii_salt: str

- **raiseonfail** (*bool*) – If True, will raise an Exception if something goes wrong.
- **override_authdb_path** (*str or None*) – If given as a str, is the alternative path to the auth DB.
- **config** (*SimpleNamespace object or None*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns Returns a dict containing a *success* key indicating if the user's password is valid and can be used. If the password is invalid, the *messages* key will contain messages that inform the user why their password was rejected.

Return type dict

authnzerver.actions.session module

This contains functions to drive session-related auth actions.

authnzerver.actions.session.**auth_delete_sessions_userid**(*payload: dict, override_authdb_path: str = None, raiseonfail: bool = False, config: types.SimpleNamespace = None*) → dict

Removes all session tokens corresponding to a user ID.

If keep_current_session is True, will not delete the session token passed in the payload. This allows for "delete all my other logins" functionality.

Parameters

- **payload** (*dict*) – This is a dict with the following required keys:

- session_token: str
- user_id: int
- keep_current_session: bool

In addition to these items received from an authnzerver client, the payload must also include the following keys (usually added in by a wrapping function):

- reqid: int or str
- pii_salt: str

- **override_authdb_path** (*str or None*) – If given as a str, is the alternative path to the auth DB.
- **raiseonfail** (*bool*) – If True, will raise an Exception if something goes wrong.
- **config** (*SimpleNamespace object or None*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns Returns a dict with a success key indicating if the sessions were deleted successfully.

Return type dict

authnzerver.actions.session.**auth_kill_old_sessions**(*session_expiry_days: int = 7*, *override_authdb_path: str = None*, *raiseonfail: bool = False*, *config: types.SimpleNamespace = None*) → dict

Kills all expired sessions.

Parameters

- **session_expiry_days** (*int*) – All sessions older than the current datetime + this value will be deleted.
- **override_authdb_path** (*str or None*) – If given as a str, is the alternative path to the auth DB.
- **raiseonfail** (*bool*) – If True, will raise an Exception if something goes wrong.
- **config** (*SimpleNamespace object or None*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns Returns a dict with a success key indicating if the sessions were deleted successfully.

Return type dict

authnzerver.actions.session.**auth_session_delete**(*payload: dict*, *override_authdb_path: str = None*, *raiseonfail: bool = False*, *config: types.SimpleNamespace = None*) → dict

Removes a session token, effectively ending a session.

Parameters

- **payload** (*dict*) – This is a dict with the following required keys:
  - session_token: str

  In addition to these items received from an authnzerver client, the payload must also include the following keys (usually added in by a wrapping function):

  - reqid: int or str
  - pii_salt: str

- **override_authdb_path** (*str or None*) – If given as a str, is the alternative path to the auth DB.
- **raiseonfail** (*bool*) – If True, will raise an Exception if something goes wrong.
- **config** (*SimpleNamespace object or None*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns Returns a dict with a success key indicating if the session was deleted successfully.

Return type dict

authnzerver.actions.session.**auth_session_exists**(*payload: dict*, *override_authdb_path: str = None*, *raiseonfail: bool = False*, *config: types.SimpleNamespace = None*) → dict

Checks if the provided session token exists.

Parameters

- **payload** (*dict*) – This is a dict, with the following keys required:
  - session_token: str

  In addition to these items received from an authnzerver client, the payload must also include the following keys (usually added in by a wrapping function):

  - reqid: int or str
  - pii_salt: str
- **override_authdb_path** (*str or None*) – If given as a str, is the alternative path to the auth DB.
- **raiseonfail** (*bool*) – If True, will raise an Exception if something goes wrong.
- **config** (*SimpleNamespace object or None*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns Returns a dict containing all of the session info if it exists and has not expired.

Return type dict

authnzerver.actions.session.**auth_session_new**(*payload: dict*, *override_authdb_path: str = None*, *raiseonfail: bool = False*, *config: types.SimpleNamespace = None*) → dict

Generates a new session token.

Parameters

- **payload** (*dict*) – This is the input payload dict. Required items:
  - ip_address: str
  - user_agent: str
  - user_id: int or None (None indicates an anonymous user)
  - expires: datetime object or date string in ISO format
  - extra_info_json: dict or None

In addition to these items received from an authnzerver client, the payload must also include the following keys (usually added in by a wrapping function):

- reqid: int or str
- pii_salt: str

- **override_authdb_path** (*str or None*) – If given as a str, is the alternative path to the auth DB.
- **raiseonfail** (*bool*) – If True, will raise an Exception if something goes wrong.
- **config** (*SimpleNamespace object or None*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns

The dict returned is of the form:

```
{'success: True or False,
 'session_token': str session token 32 bytes long in base64 format,
 'expires': str date in ISO format,
 'messages': list of str messages to pass on to the user if any}
```

Return type dict

authnzerver.actions.session.**internal_edit_session**(*payload: dict, raiseonfail: bool = False, override_authdb_path: str = None, config: types.SimpleNamespace = None*) → dict

Handles editing the *extra_info_json* field for an existing user session.

Meant for use internally in a frontend server.

Parameters

- **payload** (*dict*) – The input payload dict. Required items:
  - target_session_token: int, the session to edit
  - update_dict: dict, the changes to make to the *extra_info_json* column of the sessions table for the target session token.

  The *extra_info_json* field in the database will be updated with the info in *update_dict*. To delete an item from *extra_info_json*, pass in the special value of "__delete__" in *update_dict* for that item.

  In addition to these items received from an authnzerver client, the payload must also include the following keys (usually added in by a wrapping function):

  - reqid: int or str
  - pii_salt: str
- **raiseonfail** (*bool*) – If True, and something goes wrong, this will raise an Exception instead of returning normally with a failure condition.
- **override_authdb_path** (*str or None*) – The SQLAlchemy database URL to use if not using the default auth DB.
- **config** (*SimpleNamespace object or None*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns Returns a dict containing the new session information.

Return type dict

authnzerver.actions.user module

This contains functions to drive user account related auth actions.

`authnzerver.actions.user.`**`create_new_user`**(*payload: dict*, *min_pass_length: int = 12*, *max_unsafe_similarity: int = 33*, *override_authdb_path: str = None*, *raiseonfail: bool = False*, *config: types.SimpleNamespace = None*) → dict

Makes a new user.

Parameters

- **`payload`** (*`dict`*) – This is a dict with the following required keys:
  - full_name: str. Full name for the user
  - email: str. User's email address
  - password: str. User's password.

  Optional payload items include:
  - extra_info: dict. optional dict to add any extra info for this user, will be stored as JSON in the DB
  - verify_retry_wait: int, default: 6. This sets the amount of time in hours a user must wait before retrying a failed verification action, i.e., responding before expiry of and with the correct verification token.
  - system_id: str. If this is provided, must be a unique string that will serve as the system_id for the user. This ID is safe to share with client JS, etc., as opposed to the user_id primary key for the user. If not provided, a UUIDv4 will be generated and used for the system_id.
  - public_suffix_list: list of str. If this is provided as a payload item, it must be a list of domain name suffixes sources from the Mozilla Public Suffix list: https://publicsuffix.org/list/. This is used to check if the full name of the user may possibly be a spam link intended to be used when the authnzerver emails out verification tokens for new users. If the full name contains a suffix in this list, the user creation request will fail. If this item is not provided in the payload, this function will look up the current process's namespace to see if it was loaded there and use it from there if so. If the public suffix list can't be found in either item, new user creation will fail.

  In addition to these items received from an authnzerver client, the payload must also include the following keys (usually added in by a wrapping function):
  - reqid: int or str
  - pii_salt: str
- **`override_authdb_path`** (*`str or None`*) – If given as a str, is the alternative path to the auth DB.
- **`raiseonfail`** (*`bool`*) – If True, will raise an Exception if something goes wrong.
- **`min_pass_length`** (*`int`*) – The minimum required character length of the password.
- **`max_unsafe_similarity`** (*`int`*) – The maximum ratio required to fuzzy-match the input password against the server's domain name, the user's email, or their name.

- **config** (*SimpleNamespace object or None*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns Returns a dict with the user's user_id and user_email, and a boolean for send_verification.

Return type dict

Notes

If the email address already exists in the database, then either the user has forgotten that they have an account or someone else is being annoying. In this case, if is_active is True, we'll tell the user that we've sent an email but won't do anything. If is_active is False and emailverify_sent_datetime is at least *payload['verify_retry_wait']* hours in the past, we'll send a new email verification email and update the emailverify_sent_datetime. In this case, we'll just tell the user that we've sent the email but won't tell them if their account exists.

Only after the user verifies their email, is_active will be set to True and user_role will be set to 'authenticated'.

`authnzerver.actions.user.`**`delete_user`**(*payload: dict*, *raiseonfail: bool = False*, *override_authdb_path: str = None*, *config: types.SimpleNamespace = None*) → dict

Deletes a user.

This can only be called by the user themselves or the superuser.

FIXME: does this actually check if it's called by the right user?

FIXME: add check_permissions to this to make more robust

This will also immediately invalidate all sessions corresponding to the target user.

Superuser accounts cannot be deleted.

Parameters

- **payload** (*dict*) – This is a dict with the following required keys:
  - email: str
  - user_id: int
  - password: str

  In addition to these items received from an authnzerver client, the payload must also include the following keys (usually added in by a wrapping function):
  - reqid: int or str
  - pii_salt: str
- **override_authdb_path** (*str or None*) – If given as a str, is the alternative path to the auth DB.
- **raiseonfail** (*bool*) – If True, will raise an Exception if something goes wrong.
- **config** (*SimpleNamespace object or None*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns Returns a dict containing a success key indicating if the user was deleted.

Return type dict

authnzerver.actions.user.**internal_delete_user**(*payload: dict*, *raiseonfail: bool = False*, *override_authdb_path: str = None*, *config: types.SimpleNamespace = None*) → dict

Deletes a user and does not check for permissions.

Suitable ONLY for internal server use by a frontend. Do NOT expose this function to an end user.

Parameters

- **payload** (*dict*) – This is a dict with the following required keys:
  - target_userid: int

  In addition to these items received from an authnzerver client, the payload must also include the following keys (usually added in by a wrapping function):
  - reqid: int or str
  - pii_salt: str
- **override_authdb_path** (*str or None*) – If given as a str, is the alternative path to the auth DB.
- **raiseonfail** (*bool*) – If True, will raise an Exception if something goes wrong.
- **config** (*SimpleNamespace object or None*) – An object containing systemwide config variables as attributes. This is useful when the wrapping function needs to pass in some settings directly from environment variables.

Returns Returns a dict containing a success key indicating if the user was deleted.

Return type dict

Submodules

authnzerver.apiclient module

This contains an auto-generated API client for the authnzerver.

class authnzerver.apiclient.**APIClient**(*authnzerver_url: str = None*, *authnzerver_secret: bytes = None*, *asynchronous: bool = False*, *use_kwargs: bool = False*)

Bases: `object`

An API client for the authnzerver.

This auto-generates class methods to call for each API action available in the authnzerver API schema.

Parameters

- **authnzerver_url** (*str*) – The URL of the authnzerver to connect to.
- **authnzerver_secret** (*str*) – The shared secret key for the authnzerver.
- **asynchronous** (*bool, optional, default=False*) – If True, generates awaitable async methods for all API actions.
- **use_kwargs** (*bool, option, default=False*) – If this is True, all arguments for the auto-generated API methods will be keyword arguments instead of regular arguments for required parameters and keyword arguments for optional ones.

Notes

Since the class methods and their docstrings are dynamically generated, a simple `help()` call won't work to show docstrings.

If you're using IPython or the Jupyter notebook, using a `?` at the end of the method name works as expected:

```
# create a new client
srv = APIClient(authnzerver_url=..., authnzerver_secret=...)

# get help on the user_new() method
srv.user_new?
```

In a normal Python shell, however, you must use the following pattern to get help on an APIClient method:

```
# create a new client
srv = APIClient(authnzerver_url=..., authnzerver_secret=...)

# get help on the user_new() method
print(srv.user_new.__doc__)
```

`async_dynamic_api_function`(*api_action: str*, *use_kwargs: bool*, **args*, ***kwargs*)
Validates an API action, then fires the API call.

This version is async.

`dynamic_api_function`(*api_action: str*, *use_kwargs: bool*, **args*, ***kwargs*)
Validates an API action, then fires the API call.

`authnzerver.apiclient.`**`dynamic_docstring`**(*action: str*, *use_kwargs: bool = False*) → str
This adds a docstring to the dynamically generated function.

authnzerver.apischema module

This contains the API schema for all actions.

`authnzerver.apischema.`**`apply_typedef`**(*item*, *typedef*)
This applies isinstance() to item based on typedef.

`authnzerver.apischema.`**`validate_and_get_function`**(*request_type: str*, *request_payload: dict*)
Validates the request and returns the function needed to fulfill it.

Checks to see if the request_type can be found in the SCHEMA, then checks its request_payload dict to see if all items required are present and are the correct type.

Returns a 3-element tuple with the first element being the function name if successfully validates, None otherwise. The second element in the tuple is a list of missing or invalid request payload items for the request type. The third element in the tuple is a message.

`authnzerver.apischema.`**`validate_api_request`**(*request_type: str*, *request_payload: dict*)
Validates the incoming request.

Checks to see if the request_type can be found in the schema, then checks its request_payload dict to see if all items required are present and are the correct type.

Returns a 3-element tuple with the first element being True if the request successfully validates, False otherwise. The second element in the tuple is a list of missing or invalid request payload items for the request type. The third element in the tuple is a message.

authnzerver.authdb module

This contains SQLAlchemy models for the authnzerver.

`authnzerver.authdb.`**`create_authdb`**(*authdb_url: str*, *database_metadata: sqlalchemy.sql.schema.MetaData = MetaData()*, *echo: bool = False*, *returnconn: bool = False*) → Optional[Tuple[sqlalchemy.engine.base.Engine, sqlalchemy.sql.schema.MetaData]]

This creates an authentication database for an arbitrary SQLAlchemy DB URL.

`authnzerver.authdb.`**`create_sqlite_authdb`**(*auth_db_path: str*, *database_metadata: sqlalchemy.sql.schema.MetaData = MetaData()*, *echo: bool = False*, *returnconn: bool = False*) → Optional[Tuple[sqlalchemy.engine.base.Engine, sqlalchemy.sql.schema.MetaData]]

This creates the local SQLite auth DB.

`authnzerver.authdb.`**`get_auth_db`**(*authdb_path: str*, *database_metadata: sqlalchemy.sql.schema.MetaData = MetaData()*, *echo: bool = False*, *returnconn: bool = True*) → Union[Tuple[sqlalchemy.engine.base.Engine, sqlalchemy.engine.base.Connection, sqlalchemy.sql.schema.MetaData], Tuple[sqlalchemy.engine.base.Engine, sqlalchemy.sql.schema.MetaData]]

This just gets a connection to the auth DB.

`authnzerver.authdb.`**`initial_authdb_inserts`**(*auth_db_path: str*, *permissions_json: str = None*, *database_metadata: sqlalchemy.sql.schema.MetaData = MetaData()*, *superuser_email: str = None*, *superuser_pass: str = None*, *echo: bool = False*)

This does initial set up of the auth DB.

- adds an anonymous user
- adds a superuser with: - userid = UNIX userid - password = random 16 bytes)
- sets up the initial permissions table

Returns the superuser userid and password.

authnzerver.autosetup module

This contains functions to set up the authnzerver automatically on first-start.

`authnzerver.autosetup.`**`autogen_secrets_authdb`**(*basedir: str*, *database_url: str = None*, *interactive: bool = False*, *generate_envfile: bool = True*)

This automatically generates secrets files and an authentication DB.

Run this only once on the first start of an authnzerver.

Parameters

- **basedir** (*str*) – The base directory of the authnzerver.

  - The authentication database will be written to a file called `.authdb.sqlite` in this directory.
  - The secret token to authenticate HTTP communications between the authnzerver and a frontend server will be written to a file called `.authnzerver-secret-key` in this directory.
  - Credentials for a superuser that can be used to edit various authnzerver options, and users will be written to `.authnzerver-admin-credentials` in this directory.
  - A random salt value will be written to `.authnzerver-random-salt` in this directory. This is used to hash user IDs and other PII in logs.

- **database_url** (*str or None*) – If this is a str, must be a valid SQLAlchemy database URL to use to connect to a database and make the necessary tables for authentication info. If this is None, will create a new SQLite database in the `<basedir>/.authdb.sqlite` file.
- **interactive** (*bool*) – If True, will ask the user for an admin email address and password. Otherwise, will auto-generate both.
- **generate_envfile** (*bool*) – If True, generates an .env file in the basedir containing all the required information for the next start up of the server.

Returns **(authdb_path, creds, secret_file, salt_file, env_file)** – The names of the files written by this function will be returned as a tuple of strings.

Return type tuple of str

`authnzerver.autosetup.`**`generate_env`**(*database_path: str*, *fernet_secret_file: str*, *salt_file: str*, *basedir: str*) → Optional[str]

This generates environment variables containing the required items for authnzrv start up after autosetup is complete.

If `write_env_file` is True, will write these to an `.env` file in the `basedir`.

Parameters

- **database_path** (*str*) – The SQLAlchemy URL of the database to use, or the path on disk to an SQLite database. If `database_path` points to a file on disk, this function will assume it's an SQLite file and construct the appropriate SQLAlchemy database URL.
- **fernet_secret_file** (*str*) – The path to the shared secret key needed to secure authnzerver-frontend communications.
- **salt_file** (*str*) – The path to the file containing the PII salt to encrypt PII in authnzerver logs.
- **basedir** (*str*) – The path to the authnzerver's basedir.

Returns Returns the path to the `.env` file generated in the `basedir` as a string.

Return type environ_file

authnzerver.client module

This module contains an authnzerver client, useful for frontend servers.

class `authnzerver.client.`**`Authnzerver`**(*authnzerver_url: str = None*, *authnzerver_secret: bytes = None*, *tls_certfile: str = None*, *tls_keyfile: str = None*)

Bases: `object`

An authnzerver client class, capable of async and sync calls.

To do anything useful, an *authnzerver_url* and *authnzerver_token* are required. By default, this object will populate these from the environment using the following variables:

- AUTHNZERVER_URL -> authnzerver_url
- AUTHNZERVER_SECRET -> authnzerver_secret

These are overridden by whatever you provide in the *authnzerver_url* and *authnzerver_secret* kwargs.

If *tls_certfile* and *tls_keyfile* are both provided, they will be used to set up a TLS-enabled connection to the authnzerver.

async_request(*request_type: str*, *request_body: dict*, *request_id: Union[str, int] = None*)

This does an asynchronous request to the authnzerver.

Parameters

- **request_type** (*str*) – This should be one of the request types defined in the authnzerver HTTP API.
- **request_body** (*dict*) – A dict with the appropriate items needed for *request_type*. This should also contain a key: "client_ipaddr" with the IP address of the frontend server's client. This is used for rate-limiting authnzerver API actions per IP address per minute.
- **request_id** (*str or int, optional*) – If *request_id* is None, a random 8-byte request ID will be generated for you. Use *request_id* to track authnzerver requests throughout the response handling cycle of your frontend server.

Returns

Returns an *AuthnzerverResponse* named tuple with the following attributes:

```
(success, response, messages,
 headers, status_code, failure_reason)
```

where:

- *success* is a boolean indicating if the request was successful.
- *response* is a dict containing the full response from the authnzerver.
- *messages* is a list of strings containing any messages that are appropriate to pass on to the end-user.
- *headers* is a dict containing the response headers from the authnzerver.
- *status_code* is the HTTP status code of the authnzerver request. Use this to figure out if your request was being rate-limited (check for 429).
- *failure_reason* is None if the request was successful, but if it wasn't, contains the reason why the request might have failed; including details of any exceptions encountered. This MUST NOT be disclosed to an end-user of the frontend server.

Return type namedtuple

request(*request_type: str*, *request_body: dict*, *request_id: Union[str, int] = None*) → authnzerver.client.AuthnzerverResponse

This does a synchronous request to the authnzerver.

Parameters

- **request_type** (*str*) – This should be one of the request types defined in the authnzerver HTTP API.

- **request_body** (*dict*) – A dict with the appropriate items needed for *request_type*. This should also contain a key: "client_ipaddr" with the IP address of the frontend server's client. This is used for rate-limiting authnzerver API actions per IP address per minute.
- **request_id** (*str or int, optional*) – If *request_id* is None, a random 8-byte request ID will be generated for you. Use *request_id* to track authnzerver requests throughout the response handling cycle of your frontend server.

Returns

Returns an *AuthnzerverResponse* named-tuple with the following attributes:

```
(success, response, messages, headers,
 status_code, failure_reason)
```

where:

- *success* is a boolean indicating if the request was successful.
- *response* is a dict containing the full response from the authnzerver.
- *messages* is a list of strings containing any messages that are appropriate to pass on to the end-user.
- *headers* is a dict containing the response headers from the authnzerver.
- *status_code* is the HTTP status code of the authnzerver request. Use this to figure out if your request was being rate-limited (check for 429).
- *failure_reason* is None if the request was successful, but if it wasn't, contains the reason why the request might have failed; including details of any exceptions encountered. This MUST NOT be disclosed to an end-user of the frontend server.

Return type namedtuple

class authnzerver.client.**AuthnzerverResponse**(*success*, *response*, *messages*, *headers*, *status_code*, *failure_reason*)

Bases: `tuple`

failure_reason
Alias for field number 5

headers
Alias for field number 3

messages
Alias for field number 2

response
Alias for field number 1

status_code
Alias for field number 4

success
Alias for field number 0

authnzerver.confload module

This contains functions to load config from environ, command line params, or an envfile.

`authnzerver.confload.`**`get_conf_item`**(*env_key: Union[str, Sequence[T_co]], environment, options_object, options_key: str = None, vartype=<class 'str'>, default=None, readable_from_file: bool = False, postprocess_value: str = None, raiseonfail: bool = True, basedir: str = None*) → Any

This loads a config item from the environment or command-line options.

The order of precedence is:

1. environment or envfile if that is provided
2. command-line option

Parameters

- **`env_key`** (*`str or list/tuple of strings`*) – The environment variable that specifies the item to get. This is either a string or a list of strings. In the first instance, the specified environment variable key will be searched for and used if available. In the latter instance, each environment variable key specified as a string in the list will be searched for, left to right, and the first one found will be used as the source of the environment variable's value. This allows you to specify fallback environment variables, e.g., setting `'env': ['PORT', 'AUTHNZERVER_PORT']` in a *conf_dict* item will look for the environment variable key `PORT` first and fall back to `AUTHNZERVER_PORT`.
- **`environment`** (*`environment object or ConfigParser object`*) – This is an object similar to that obtained from `os.environ` or a similar ConfigParser object.
- **`options_object`** (*`Tornado options object`*) – If the environment variable isn't defined, the next place this function will try to get the item value from a passed-in Tornado options object, which parses command-line options.
- **`options_key`** (*`str`*) – This is the attribute to look up in the options object for the value of the conf item.
- **`vartype`** (*`Python type object:  float, str, int, etc.`*) – The type to use to coerce the input variable to a specific Python type.
- **`default`** (*`Any`*) – The default value of the conf item.
- **`readable_from_file`** (*`{'json','string', others, see below} or False`*) – If this is specified, and the conf item key (env_key or options_key above) is a valid filename or URL, will open it and read it in, cast to the specified variable type, and return the item. If this is set to False, will treat the config item pointed to by the key as a plaintext item and return it directly.

  There are several readable_from_file options. The first two below are strings, the rest are tuples.

  – `'string'`: read a file and use the resulting string as the value of the config item. The trailing `\n` character will be stripped. This is useful for simple text secret keys stored in a file on disk, etc.

  – `'json'`: read the entire file as JSON and return the loaded dict as the value of the config item.

  – `('json','path.to.item.or.listitem._arr_0')`: read the entire file as JSON, resolve the JSON object path pointed to by the second tuple element, get the value there and return it as the value of the config item.

  – `('http',{method dict},'string')`: HTTP GET/POST the URL pointed to by the config item key, assume the value returned is plain-text and return it as the value of the config item. This can be useful for things stored in AWS/GCP metadata servers.

- `('http',{method dict},'json')`: HTTP GET/POST the URL pointed to by the config item key, load it as JSON, and return the loaded dict as the value of the config item.
- `('http',{method dict},'json','path.to.item.or.listitem._arr_0')`: HTTP GET the URL pointed to by the config key, load it as JSON, resolve the JSON object path pointed to by the fourth element of the tuple, get the value there and return it as the value of the config item.

The `{method dict}` is a dict of the following form:

```
{'method':'post' or 'get',
 'headers':dict of header keys and values to send or None,
 'data':data dict to attach to the POST request or param dict to
        attach to the GET request or None,
 'timeout': time in seconds to wait for a response}
```

Using the method dict allows you to add in authentication headers and data needed to gain access to the URL indicated by the config item key.

If an item in the 'headers' or 'data' dicts requires something from an environment variable or .env file, indicate this by using `'[[NAME OF ENV VAR]]'` in the value of that key. For example, to get a bearer token to use in the 'Authorization' header:

```
method_dict['headers'] = {'Authorization': 'Bearer [[API_KEY]]'}
```

This will look up the environment variable 'API_KEY' and substitute that value in.

- **`postprocess_value`** (*`str`*) – This is a string pointing to a Python function to apply to the config item that was retrieved. The function must take one argument and return one item. The function is specified as either a fully qualified Python module name and function name, e.g.:

```
'base64.b64decode'
```

or a path to a Python module on disk and the function name separated by '::'

```
'~/some/directory/mymodule.py::custom_b64decode'
```

- **`raiseonfail`** (*`bool`*) – If this is set to True, the function will raise a ValueError for any missing config items that can't be set from the environment, the envfile or the command-line options. If this is set to False, the function won't immediately raise an exception, but will return None. This latter behavior is useful for indicating which configuration items are missing (e.g. when a server is being started for the first time.)
- **`basedir`** (*`str`*) – The directory where the server will do its work. This is used to fill in `'[[basedir]]'` template values in any conf item. By default, this is the current working directory.

Returns The value of the configuration item.

Return type Any

`authnzerver.confload.`**`item_from_file`**(*file_path: str, file_spec: Union[tuple, str], basedir: str = None*) → Any

Reads a conf item from a file.

Parameters

- **`file_path`** (*`str`*) – The file to open. Here you can use the following substitutions as necessary:
  - `[[homedir]]`: points to the home directory of the user running the server.
  - `[[basedir]]`: points to the base directory of the server.
- **`file_spec`** (*`str or tuple`*) – This specifies how to read the conf item from the file:
  - `'string'`: read a file and use the resulting string as the value of the config item. The trailing `\n` character will be stripped. This is useful for simple text secret keys stored in a file on disk, etc.
  - `'json'`: read the entire file as JSON and return the loaded dict as the value of the config item.
  - `('json','path.to.item.or.listitem._arr_0')`: read the entire file as JSON, resolve the JSON object path pointed to by the second tuple element, get the value there and return it as the value of the config item.
- **`basedir`** (*`str or None`*) – The base directory of the server. If None, the current working directory is used.

Returns **conf_value** – Returns the value of the conf item. The calling function is responsible for casting to the correct type.

Return type Any

`authnzerver.confload.`**`item_from_url`**(*url: str*, *url_spec: tuple*, *environment*, *timeout: Union[float, int] = 5.0*)

Reads a conf item from a URL.

Parameters

- **`url`** (*`str`*) – The URL to fetch.
- **`url_spec`** (*`tuple`*) – This specifies how to get the conf item from the URL:
  - `('http',{method dict},'string')`: HTTP GET/POST the URL pointed to by the config item key, assume the value returned is plain-text and return it as the value of the config item. This can be useful for things stored in AWS/GCP metadata servers.
  - `('http',{method dict},'json')`: HTTP GET/POST the URL pointed to by the config item key, load it as JSON, and return the loaded dict as the value of the config item.
  - `('http',{method dict},'json','path.to.item.or.listitem._arr_0')`: HTTP GET the URL pointed to by the config key, load it as JSON, resolve the JSON object path pointed to by the fourth element of the tuple, get the value there and return it as the value of the config item.

  The `{method dict}` is a dict of the following form:

```
{'method':'post' or 'get',
 'headers':dict of header keys and values to send or None,
 'data':data dict to attach to the POST request or param dict to
        attach to the GET request or None,
 'timeout': time in seconds to wait for a response}
```

  Using the method dict allows you to add in authentication headers and data needed to gain access to the URL indicated by the config item key.

If an item in the 'headers' or 'data' dicts requires something from an environment variable or .env file, indicate this by using `'[[NAME OF ENV VAR]]'` in the value of that key. For example, to get a bearer token to use in the 'Authorization' header:

```
method_dict['headers'] = {'Authorization': 'Bearer [[API_KEY]]'}
```

This will look up the environment variable 'API_KEY' and substitute that value in.

- **`environment`** (*`environment object or ConfigParser object`*) – This is an object similar to that obtained from `os.environ` or a similar ConfigParser object.
- **`timeout`** (*`int or float`*) – The default timeout in seconds to use for the HTTP request if one is not provided in the method dict in `url_spec`.

Returns **conf_value** – Returns the value of the conf item. The calling function is responsible for casting to the correct type.

Return type Any

`authnzerver.confload.`**`load_config`**(*conf_dict: dict*, *options_object*, *envfile: str = None*) → types.SimpleNamespace

Loads all the config items in config_dict.

Parameters

- **`conf_dict`** (*`dict`*) – This is a dict containing information on each config item to load and return. Each key in this dict serves as the name of the config item and the value for each key is a dict of the following form:

```
'conf_item_name':{
    'env':'The environmental variable to check',
    'cmdline':'The command-line option to check',
    'type':the Python type of the config item,
    'default':a default value for the config item or None,
    'help':'The help string to use for the command-line option',
    'readable_from_file':how to retrieve the item (see below),
    'postprocess_value': 'func to postprocess the item (see below)
↪',
},
```

The `env` key in each config item is either a string or a list of strings. In the first instance, the specified environment variable key will be searched for and used if available. In the latter instance, each environment variable key specified as a string in the list will be searched for, left to right, and the first one found will be used as the source of the environment variable's value. This allows you to specify fallback environment variables, e.g., setting `'env': ['PORT', 'AUTHNZERVER_PORT']` in a *conf_dict* item will look for the environment variable key `PORT` first and fall back to `AUTHNZERVER_PORT`.

The `'readable_from_file'` key in each config item's dict indicates how the value present in either the environment variable or the command-line option will be used to retrieve the config item. This is one of the following:

– `'string'`: read a file and use the resulting string as the value of the config item. The trailing `\n` character will be stripped. This is useful for simple text secret keys stored in a file on disk, etc.

– `'json'`: read the entire file as JSON and return the loaded dict as the value of the config item.

– `('json','path.to.item.or.listitem._arr_0')`: read the entire file as JSON, resolve the JSON object path pointed to by the second tuple element, get the value there and return it as the value of the config item.

- `('http',{method dict},'string')`: HTTP GET/POST the URL pointed to by the config item key, assume the value returned is plain-text and return it as the value of the config item. This can be useful for things stored in AWS/GCP metadata servers.
- `('http',{method dict},'json')`: HTTP GET/POST the URL pointed to by the config item key, load it as JSON, and return the loaded dict as the value of the config item.
- `('http',{method dict},'json','path.to.item.or.listitem._arr_0')`: HTTP GET the URL pointed to by the config key, load it as JSON, resolve the JSON object path pointed to by the fourth element of the tuple, get the value there and return it as the value of the config item.

The `{method dict}` is a dict of the following form:

```
{'method':'post' or 'get',
 'headers':dict of header keys and values to send or None,
 'data':data dict to attach to the POST request or param dict to
        attach to the GET request or None,
 'timeout': time in seconds to wait for a response}
```

Using the method dict allows you to add in authentication headers and data needed to gain access to the URL indicated by the config item key.

If an item in the 'headers' or 'data' dicts requires something from an environment variable or .env file, indicate this by using `'[[NAME OF ENV VAR]]'` in the value of that key. For example, to get a bearer token to use in the 'Authorization' header:

```
method_dict['headers'] = {'Authorization': 'Bearer [[API_KEY]]'}
```

This will look up the environment variable 'API_KEY' and substitute that value in.

The `'postprocess_value'` key in each config item's dict is used to point to a Python function to post-process the config item after it has been retrieved. The function must take one argument and return one item. The function is specified as either a fully qualified Python module name and function name, e.g.:

```
'base64.b64decode'
```

or a path to a Python module on disk and the function name separated by '::'

```
'~/some/directory/mymodule.py::custom_b64decode'
```

- **`options_object`** (*`Tornado options object`*) – If the environment variable isn't defined for a config item, the next place this function will try to get the item value from a passed-in Tornado options object, which parses command-line options.
- **`envfile`** (*`str or None`*) – The path to a file containing key=value pairs in the same manner as environment variables. This serves as an override to any environment variables that this function looks up to find config items.

Returns **loaded_config** – This returns an object with the parsed final values of each of the config items as object attributes.

Return type SimpleNamespace object

authnzerver.confvars module

Contains the configuration variables that define how the server operates.

The CONF dict in this file describes how to load these variables from the environment or command-line options.

You can change this file as needed. It will be copied over to the authnzerver's base directory when `authnzrv --autosetup` is run and you can tell authnzerver to use it like so: `authnzrv --confvars /path/to/basedir/confvars.py`.

You MUST NOT store any actual secrets in this file; just define how to get to them.

For example, look at the `secret` dict entry below in CONF:

```
'secret': {
    'env': '%s_SECRET' % ENVPREFIX,
    'cmdline': 'secret',
    'type': str,
    'default': None,
    'help': ('The shared secret key used to secure '
            'communications between authnzerver and any frontend servers.'),
    'readable_from_file': 'string',
    'postprocess_value': None,
}
```

This means the server will look at an environmental variable called `AUTHNZERVER_SECRET`, falling back to the value provided in the `--secret` command line option. The `readable_from_file` key tells the server how to handle the value it retrieved from either of these two sources.

To indicate that the retrieved value is to be used directly, set `"readable_from_file" = False`.

To indicate that the retrieved value can either be: (i) used directly or, (ii) may be a path to a file and the actual value of the `secret` item is a string to be read from that file, set `"readable_from_file" = "string"`.

To indicate that the retrieved value is a URL and the authnzerver must fetch the actual secret from this URL, set:

```
"readable_from_file" = ("http",
                        {'method': 'get',
                         'headers': {header dict},
                         'data': {param dict},
                         'timeout': 5.0},
                         'string')
```

Finally, you can also tell the server to fetch a JSON and pick out a key in the JSON. See the docstring for : py: func: *authnzerver.confload.get_conf_item* for more details on the various ways to retrieve the actual item pointed to by the config variable key.

To make this example more concrete, if the authnzerver `secret` was stored as a GCP Secrets Manager item, you'd set some environmental variables like so:

```
GCP_SECMAN_URL=https://secretmanager.googleapis.com/v1/projects/abcproj/secrets/abc/
↪versions/z:access
GCP_AUTH_TOKEN=some-secret-token
```

Then change the `secret` dict item in CONF dict below to:

```
'secret': {
    'env': 'GCP_SECMAN_URL',
    'cmdline': 'secret',
```

(continues on next page)

(continued from previous page)

```
    'type': str,
    'default': None,
    'help': ('The shared secret key used to secure '
            'communications between authnzerver and any frontend servers.'),
    'readable_from_file': see below,
    'postprocess_value': 'custom_decode.py:: custom_b64decode',
}
```

The `readable_from_file` key would be set to something like:

```
"readable_from_file" = ("http",
                        {"method": "get",
                         "headers": {
                             "Authorization": "Bearer [[GCP_AUTH_TOKEN]]",
                             "Content-Type": "application/json",
                             "x-goog-user-project": "abcproj"
                         },
                         "data": None,
                         "timeout": 5.0},
                        'json',
                        "payload.data")
```

This would then load the authnzerver `secret` directly from the Secrets Manager.

Notice that we used a path to a Python module and function for the `postprocess_value` key. This is because GCP's Secrets Manager base-64 encodes the data you put into it and we need to post-process the value we get back from the stored item's URL. This module looks like:

```
import base64

def custom_b64decode(input):
    return base64.b64decode(input.encode('utf-8')).decode('utf-8')
```

The function above will base-64 decode the value returned from the Secrets Manager and finally give us the `secret` value we need.

authnzerver.dictcache module

This contains a simple dict-based in-memory key-value store.

Suitable for light-duty caching purposes.

class `authnzerver.dictcache.`**`DictCache`**(*capacity: int = 20000*)

Bases: `object`

`add`(*key*, *value*, *ttl=None*, *extras=None*)

Adds a key and sets it to the value.

If the key already exists, does nothing.

If value is None, does not add the key to the cache because this would be pointless.

if extras is provided, it must be a dict with key:val pairs. These will be added to the stored item in the container in a dict key called 'extras'.

`counter_add`(*key*, *initial_value*, *ttl=None*)

Adds a new counter key to the cache with the specified initial value.

`counter_decrement`(*key*, *ttl=None*)
Decrements a counter key by 1 every time it's called.

This will pop the key when its count reaches zero either after the current decrement or if the count has already reached zero before the decrement operation will be performed.

Returns the new count after decrement or None if the key doesn't exist in the cache.

`counter_get`(*key*)
This gets the current count for a counter key.

`counter_increment`(*key*, *ttl=None*)
This increments a counter key by 1 every time it's called and returns the new count.

If the key doesn't exist, adds it to the cache with an initial count of 1.

`counter_rate`(*key*, *period_seconds*, *return_allinfo=False*, *absolute_rate=True*)
This gets the rate of increment/decrement over period (in seconds) for a counter key that was incremented in the past.

If the counter key does not exist, returns None.

If return_allinfo = True, returns a tuple with the current rate, the current value, the initial value, the current time, and the insertion time. Otherwise, returns only the rate as a float.

If absolute_rate is True, returns the absolute value of the rate.

`counter_set`(*key*, *value*, *ttl=None*)
Sets the counter key to the specified value.

`delete`(*key*)
Deletes the key from the cache.

`flush`()
This removes all items in the cache.

`get`(*key*, *time_and_ttl=False*, *extras=False*)
Gets the value of key from the cache.

`info`()
Returns the capacity of the cache, and number of normal and TTL items.

`load`(*infile*, *hmac_key=None*)
This loads contents of the cache from a pickle file on disk.

If hmac_key is not None, this function will assume it has to load a signed pickle. If hmac_key is None but the saved pickle was signed, loading will throw an exception.

`pop`(*key*)
Pops the key from the cache.

`save`(*outfile*, *protocol=4*, *hmac_key=None*)
This saves the current contents of the cache to disk.

The items stored must be pickleable.

If hmac_key is not None, the pickle will be signed before saving it to disk.

`set`(*key*, *value*, *ttl=None*, *extras=None*, *add_ifnotexists=True*)
This sets the value of key to value and returns the new value.

If the key doesn't exist and add_ifnotexists is False, returns None. If add_ifnotexists is True, adds the key to the cache and returns the value.

ttl = None implies that the TTL no longer applies, in which it will be removed from the key, meaning the key becomes persistent.

extras is a dict with key:val pairs that will update the existing extras dict for item in the container using the dict.update() method.

size()
Returns the number of items in the cache.

time()
Returns the cache's current time time counter.

class authnzerver.dictcache.**KeyWithTime**(*keytime*, *key*)
Bases: `tuple`

key
Alias for field number 1

keytime
Alias for field number 0

authnzerver.handlers module

These are handlers for the authnzerver.

class authnzerver.handlers.**AuthHandler**(*application: tornado.web.Application*, *request: tornado.httputil.HTTPServerRequest*, ***kwargs*)
Bases: `tornado.web.RequestHandler`, *authnzerver.ratelimit.RateLimitMixin*, *authnzerver.ratelimit.UserLockMixin*

This handles the actual auth requests.

initialize(*config*, *executor*, *cacheobj*, *failed_passchecks*)
This sets up stuff.

post()
Handles the incoming POST request.

send_response(*response*, *reqid*)
This handles the response generation.

write_error(*status_code*, ***kwargs*)
This writes the error as a response.

authnzerver.healthcheck module

These are handlers to respond to health-check requests.

class authnzerver.healthcheck.**HealthCheckHandler**(*application: tornado.web.Application*, *request: tornado.httputil.HTTPServerRequest*, ***kwargs*)
Bases: `tornado.web.RequestHandler`, *authnzerver.ratelimit.RateLimitMixin*

This handles health check endpoints.

get()
This responds to a health-check request.

Returns 200 if the server is up and all the background workers report their DB connection is good.

initialize(*config*, *executor*, *cacheobj*)
This sets up some config.

write_error(*status_code*, ***kwargs*)
This writes the error as a response.

authnzerver.jsonencoder module

The JSON encoder class for all handlers.

class `authnzerver.jsonencoder.`**FrontendEncoder**(***, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

Bases: `json.encoder.JSONEncoder`

default(*obj*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

authnzerver.main module

This is the main file for the authnzerver, a simple authorization and authentication server backed by SQLite, SQLAlchemy, and Tornado.

`authnzerver.main.`**main**()
This is the main function.

authnzerver.messaging module

This module handles the serialization-deserialization of messages between the authnzerver and any frontends.

`authnzerver.messaging.`**chacha_decrypt_message**(*message: bytes*, *key: bytes*, *reqid: Union[str, int] = None*, *ttl: int = None*) → Optional[dict]

Decrypts a ChaCha20-Poly1305-encrypted message back to a message dict.

This depends on OpenSSL containing the cipher, so OpenSSL > 1.1.0 probably. The version of the cipher used is the IETF-approved one (https://tools.ietf.org/html/rfc7539; with the 96-bit nonce).

Parameters

- **message** (*bytes*) – The encrypted message to decrypt.

- **key** (*bytes*) – This is the 32-byte encryption key. Must be the same one as used for encrypting the message (i.e. this is a pre-shared secret key)
- **reqid** (*str or int or None*) – A request ID used to track a decryption request. This will appear in any logging messages emitted by this function to allow tracking of requests and correlation.
- **ttl** (*int or None*) – The age in seconds that the encrypted message must not exceed in order for it to be considered valid. This is useful for time-stamped verification tokens. If None, the message will not be checked for expiry.

Returns **message_dict** – Returns the decrypted message dict. If the message expired or if the message failed to decrypt because of an invalid key or if it was tampered with, returns None instead.

Return type dict or None

authnzerver.messaging.**chacha_encrypt_message**(*message_dict: dict*, *key: bytes*, *nonce: bytes = None*) → bytes

Encrypts a dict using the ChaCha20-Poly1305 symmetric cipher.

This depends on OpenSSL containing the cipher, so OpenSSL > 1.1.0 probably. The version of the cipher used is the IETF-approved one (https://tools.ietf.org/html/rfc7539; with the 96-bit nonce).

Parameters

- **message_dict** (*dict*) – A dict containing items that will be encrypted.
- **key** (*bytes*) – This is a 32-byte encryption key. Generate one using:

```
import secrets
key = secrets.token_bytes(32)
```

- **nonce** (*bytes or None*) – This is a 12-byte nonce used for encryption. This MUST NOT be re-used with the same key if you want the message to remain secret. If None, a random 12-byte value will be used.

Returns **encrypted_message** – Returns the encrypted message as base64 encoded bytes.

Return type bytes

Notes

The encrypted message is generated in the following format:

```
base64(encrypt(<nonce><message dict + iat + ver>))
```

authnzerver.messaging.**decrypt_message**(*message: bytes*, *key: bytes*, *reqid: Union[int, str] = None*, *ttl: int = None*) → Optional[dict]

Decrypts a Fernet-encrypted message back to a message dict.

Parameters

- **message** (*bytes*) – The encrypted message to decrypt.
- **key** (*bytes*) – This is the 32-byte encryption key in URL-safe base64 format. Must be the same one as used for encrypting the message (i.e. this is a pre-shared secret key)
- **reqid** (*str or int or None*) – A request ID used to track a decryption request. This will appear in any logging messages emitted by this function to allow tracking of requests and correlation.

- **ttl** (*int or None*) – The age in seconds that the encrypted message must not exceed in order for it to be considered valid. This is useful for time-stamped verification tokens. If None, the message will not be checked for expiry.

Returns **message_dict** – Returns the decrypted message dict. If the message expired or if the message failed to decrypt because of an invalid key or if it was tampered with, returns None instead.

Return type dict or None

`authnzerver.messaging.`**`encrypt_message`**(*message_dict: dict*, *key: bytes*) → bytes

Encrypts a message dict using Fernet from the PyCA cryptography package.

Parameters

- **message_dict** (*dict*) – A dict containing items that will be encrypted.
- **key** (*bytes*) – This is a 32-byte encryption key in URL-safe base64 format. Generate one using:

```
import os, base64
fernet_key = base64.urlsafe_b64encode(os.urandom(32))
```

Returns **encrypted_message** – Returns the encrypted message as base64 encoded bytes.

Return type bytes

`authnzerver.messaging.`**`xsalsa_decrypt_message`**(*message: bytes*, *key: bytes*, *reqid: Union[int, str] = None*, *ttl: int = None*) → Optional[dict]

Decrypts a XSalsa20-Poly1305-encrypted message back to a message dict.

This function requires PyNACL.

Parameters

- **message** (*bytes*) – The encrypted message to decrypt.
- **key** (*bytes*) – This is the 32-byte encryption key. Must be the same one as used for encrypting the message (i.e. this is a pre-shared secret key)
- **reqid** (*str or int or None*) – A request ID used to track a decryption request. This will appear in any logging messages emitted by this function to allow tracking of requests and correlation.
- **ttl** (*int or None*) – The age in seconds that the encrypted message must not exceed in order for it to be considered valid. This is useful for time-stamped verification tokens. If None, the message will not be checked for expiry.

Returns **message_dict** – Returns the decrypted message dict. If the message expired or if the message failed to decrypt because of an invalid key or if it was tampered with, returns None instead.

Return type dict or None

`authnzerver.messaging.`**`xsalsa_encrypt_message`**(*message_dict: dict*, *key: bytes*) → bytes

Encrypts a dict using the XSalsa20-Poly1305 symmetric cipher.

This function requires PyNACL.

Parameters

- **message_dict** (*dict*) – A dict containing items that will be encrypted.
- **key** (*bytes*) – This is a 32-byte encryption key. Generate one using:

```
import secrets
key = secrets.token_bytes(32)
```

Returns **encrypted_message** – Returns the encrypted message as base64 encoded bytes.

Return type bytes

authnzerver.modtools module

This contains functions to dynamically import modules and get Python objects.

`authnzerver.modtools.`**`module_from_string`**`(`*module*, *force_reload=False*`)`

This imports the module specified.

Used to dynamically import Python modules.

Parameters

- **`module`** (*`str`*) – This is either:
  - a Python module import path, e.g. 'concurrent.futures' or
  - a path to a Python file, e.g. '~/authnzerver/authnzerver/main.py'
- **`force_reload`** (*`bool`*) – If True, will reload a previous imported module even if it's been previously imported. This is useful to pick up changes in Python module files used as program config files.

Returns This returns a Python module if it's able to successfully import it.

Return type Python module

Notes

Hypens are not allowed in module filenames.

`authnzerver.modtools.`**`object_from_string`**`(`*objectpath*, *force_reload=False*`)`

This returns a Python object pointed to by the given string.

An object can be any valid Python object. One of the main uses for this function is to dynamically load Python functions from a module given its file path on disk or a fully qualified module string.

The string should be in one of the forms below:

- fully qualified module name and object name, e.g.:

```
'authnzerver.confvars.CONF' -> gets the CONF dict
'sqlalchemy.dialects.postgresql.JSONB' -> gets the JSONB class
'scipy.ndimage.convolve' -> gets the convolve() function
```

- path to a module on disk and the object name separated by '::', e.g.:

```
'~/authnzerver/authnzerver/actions/user.py::create_new_user'
'~/authzerver/authnzerver/authdb.py::Users'
```

(This is similar to the format used by pytest.)

authnzerver.permissions module

This contains the permissions and user-role models for authnzerver.

`authnzerver.permissions.`**`check_item_access`**(*permissions_model: dict*, *userid: int = 2*, *role: str = 'anonymous'*, *action: str = 'view'*, *target_name: str = 'collection'*, *target_owner: int = 1*, *target_visibility: str = 'private'*, *target_sharedwith: str = None*, *debug: bool = False*) → bool

This does a check for user access to a target item.

Parameters

- **`permissions_model`** (*`dict`*) – A permissions model returned by *`load_permissions_json()`*.
- **`userid`** (*`int`*) – The userid of the user requesting access.
- **`role`** (*`str`*) – The role of the user requesting access.
- **`action`** (*`str`*) – The action requested to be applied to the item.
- **`target_name`** (*`str`*) – The name of the item for which the policy will be checked.
- **`target_owner`** (*`int`*) – The userid of the user that owns the item for which the policy will be checked.
- **`target_visibility`** (*`str`*) – The visibility of the item for which the policy will be checked.
- **`target_sharedwith`** (*`str`*) – A CSV string of the userids that the target item is shared with.
- **`debug`** (*`bool`*) – If True, will report the various policy decisions applied.

Returns True if access was granted. False otherwise.

Return type bool

`authnzerver.permissions.`**`check_role_limits`**(*permissions_model: dict, role: str, limit_name: str, value_to_check: Union[float, int]*) → bool

This applies the role limits to a value to check.

Parameters

- **`permissions_model`** (*`dict`*) – A permissions model returned by *`load_permissions_json()`*.
- **`role`** (*`str`*) – The name of the role to check the limits for.
- **`limit_name`** (*`str`*) – The name of limit to check.
- **`value_to_check`** (*`float or int`*) – The value to check against the limit.

Returns Returns True if the limit hasn't been exceeded. Returns False otherwise.

Return type bool

`authnzerver.permissions.`**`get_item_actions`**(*permissions_model: dict*, *role_name: str*, *target_name: str*, *target_visibility: str*, *target_ownership: str*, *debug: bool = False*) → Set[T]

Returns the possible actions for a target given a role and target status.

Parameters

- **permissions_model** (*dict*) – A permissions model returned by *load_permissions_json()*.
- **role_name** (*str*) – The name of the role to find the valid actions for.
- **target_name** (*str*) – The name of the item to check the valid actions for.
- **target_visibility** (*str*) – The visibility of the tiem to check the valid actions for.
- **target_ownership** (*{'for_owned', 'for_other'}*) – If 'for_owned', only the valid actions for the target item available if the item is owned by the user will be returned. If 'for_other', only the valid actions subject to the visibility of the item owned by other users will be returned.
- **debug** (*bool*) – If True, will print the policy decisions being taken.

Returns Returns a set of valid actions for the target item based on the applied policy. If the actions don't make sense, returns an empty set, in which case access MUST be denied.

Return type set

authnzerver.permissions.**load_permissions_json**(*model_json: str*) → dict

Loads a permissions JSON and returns the model.

authnzerver.permissions.**load_policy_and_check_access**(*permissions_json: str, userid: int = 2, role: str = 'anonymous', action: str = 'view', target_name: str = 'collection', target_owner: int = 1, target_visibility: str = 'private', target_sharedwith: str = None, debug: bool = False*) → bool

Does a check for user access to a target item.

This version loads a permissions JSON from disk every time it is called.

Parameters

- **permissions_json** (*str*) – A JSON file containing a permissions model.
- **userid** (*int*) – The userid of the user requesting access.
- **role** (*str*) – The role of the user requesting access.
- **action** (*str*) – The action requested to be applied to the item.
- **target_name** (*str*) – The name of the item for which the policy will be checked.
- **target_owner** (*int*) – The userid of the user that owns the item for which the policy will be checked.
- **target_visibility** (*str*) – The visibility of the item for which the policy will be checked.
- **target_sharedwith** (*str*) – A CSV string of the userids that the target item is shared with.
- **debug** (*bool*) – If True, will report the various policy decisions applied.

Returns True if access was granted. False otherwise.

Return type bool

authnzerver.permissions.**load_policy_and_check_limits**(*permissions_json: str, role: str, limit_name: str, value_to_check: Union[float, int]*)

Applies the role limits to a value to check.

This version loads a policy JSON every time it is called.

Parameters

- **permissions_json** (*dict*) – A JSON file containing a permissions model.
- **role** (*str*) – The name of the role to check the limits for.
- **limit_name** (*str*) – The name of limit to check.
- **value_to_check** (*float or int*) – The value to check against the limit.

Returns Returns True if the limit hasn't been exceeded. Returns False otherwise.

Return type bool

authnzerver.permissions.**pii_blake2b_hash**(*item*, *key*)

This generates a 12-byte digest for an item using the Blake-2b hash function. This should be more secure than the pii_hash() function above.

authnzerver.permissions.**pii_hash**(*item*, *key*)

This wraps one of the functions above.

authnzerver.permissions.**pii_sha256_hash**(*item*, *salt*)

This generates a SHA256 hash for a PII-item that is concatenated with a random 'salt' to add some security against attacks, but still allow correlation of the PII item in logs.

authnzerver.ratelimit module

This module contains RequestHandler mixins that do rate-limiting for the authnzerver's own API, handle throttling of incorrect password attempts, and do user locking/unlocking for repeated password check failures.

None of these will work without bits already defined in handlers.AuthHandler or close derivatives.

class authnzerver.ratelimit.**RateLimitMixin**

Bases: `object`

This class contains a method that rate-limits the authnzerver's own API.

Requires:

- self.cacheobj (from AuthHandler)
- self.ratelimits (from AuthHandler)
- self.pii_salt (from AuthHandler)
- self.request.remote_ip (from tornado.web.RequestHandler)

ratelimit_request(*reqid: Union[int, str], request_type: str, frontend_client_ipaddr: str, request_body: dict = None*) → None

This rate-limits the request based on the request type and the set ratelimits passed in the config object.

class authnzerver.ratelimit.**UserLockMixin**

Bases: `object`

This class handles user locking/unlocking and slowing down repeated password failures.

handle_failed_logins(*payload: dict*) → tuple

This handles failed logins.

- Adds increasing wait times to successive logins if they keep failing.
- If the number of failed logins exceeds 10, the account is locked for one hour, and an unlock action is scheduled on the ioloop.

Requires:

- self.failed_passchecks (from AuthHandler)
- self.config (from AuthHandler)

lockuser_repeated_login_failures(*payload: dict*, *unlock_after_seconds: int = 3600*) → dict

This locks the user account. Also schedules an unlock action for later.

Requires:

- self.config (from AuthHandler)
- self.executor (from AuthHandler)
- self.scheduled_user_unlock() (from UserLockMixin)

scheduled_user_unlock(*user_id: int, reqid: Union[int, str], pii_salt: str*)

This function is scheduled on the ioloop to unlock the specified user.

authnzerver.tokens module

This module handles generation of various tokens.

authnzerver.tokens.**generate_email_token**(*ip_address: str*, *user_agent: str*, *email_address: str*, *session_token: str*, *session_cookie_key: bytes*) → bytes

This generates a token useful for verifying email addresses.

Also used for forgot-password emails.

This encodes the user's IP address, user agent, email address, and session token into the token generated. The token is encrypted using the Fernet scheme and the session cookie (the key used to sign the frontend's cookies) to keep things simple.

authnzerver.tokens.**verify_email_token**(*token: bytes*, *ip_address: str*, *user_agent: str*, *session_token: str*, *email_address: str*, *session_cookie_key: bytes*, *match_returned_items: Sequence[T_co] = ('ipa', 'ema')*, *ttl_seconds: int = 900*, *reqid: Union[int, str] = None*) → bool

This verifies the token returned by the user.

By default, it requires that the token be returned no more than 15 minutes after it's been issued. It also tries to match the specified items in `match_returned_items` to the current values provided as args:

```
'ipa' -> ip_address
'usa' -> user_agent
'stk' -> session_token
'ema' -> email_address
```

authnzerver.validators module

This module contains validation functions taken from the James Bennett's excellent django-registration package. I've modified it a bit so the validators don't need Django to work. The original docstring and the BSD License for that package are reproduced immediately below.

Copyright (c) 2007-2018, James Bennett All rights reserved.

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 author nor the names of other contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Error messages, data and custom validation code used in django-registration's various user-registration form classes.

`authnzerver.validators.`**`get_public_suffix_list`**(*return_set: bool = False*, *save_to_currproc: bool = False*) → Union[list, set]

Gets the public suffix list and caches it if necessary.

`authnzerver.validators.`**`normalize_value`**(*value: str*, *casefold: bool = True*) → str

This normalizes a given value and casefolds it.

Assumes that the value has already passed validation.

`authnzerver.validators.`**`public_suffix_list`**(*return_set: bool = False*) → Union[list, set]

Retrieves the Internet names public suffix list and loads it into a set.

`authnzerver.validators.`**`validate_confusables`**(*value: str*)

This validates if the value is not a confusable homoglyph.

`authnzerver.validators.`**`validate_confusables_email`**(*value: str*) → bool

Validator which disallows 'dangerous' email addresses likely to represent homograph attacks.

An email address is 'dangerous' if either the local-part or the domain, considered on their own, are mixed-script and contain one or more characters appearing in the Unicode Visually Confusable Characters file.

`authnzerver.validators.`**`validate_email_address`**(*emailaddr: str*) → bool

This validates an email address using the HTML5 specification, which is good enough for most purposes.

The regex is taken from here:

https://blog.gerv.net/2011/05/html5_email_address_regexp/

And was transformed to Python using the excellent https://regex101.com.

`authnzerver.validators.`**`validate_reserved_name`**(*value: str*) → bool

This validates if the value is not one of the reserved names.

`authnzerver.validators.`**`validate_unique_value`**(*value: str, check_list: Sequence[T_co]*) → bool

This checks if the input value does not already exist in the check_list.

The check_list comes from the DB and should contain user names, etc. that have been already normalized and casefolded.

CHAPTER 3

Changelog and TODO

Please see CHANGELOG.md in the Github repository for the latest changelog for tagged versions.

Similarly, please see TODO.md in the Github repository for items being worked on or in the pipeline for future versions.

CHAPTER 4

License

Authnzerver is provided under the MIT License. See the LICENSE file for the full text.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index