authnzerver

Release 0.1.3

Jun 24, 2020

Full API documentation

Authnzerver is a tiny authentication-authorization server, meant to add these capabilities to other HTTP servers.

This documentation is a work in progress. For now, see the auto-generated API docs.

CHAPTER 1

authnzerver

1.1 authnzerver package

1.1.1 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*, *raiseonfail=False*, *override_permissions_json=None*, *override_authdb_path=None*)

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.

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*, *raiseonfail=False*, *override_permissions_json=None*, *override_authdb_path=None*)

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.

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*, *raiseonfail=False*, *override_permissions_json=None*, *override_authdb_path=None*)

This edits users.

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.

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.**internal_toggle_user_lock**(*payload*, *raiseonfail=False*, *override_authdb_path=None*)

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.

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*, *raiseonfail=False*, *override_authdb_path=None*)

This lists users.

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.

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.**toggle_user_lock**(*payload*, *raiseonfail=False*, *override_authdb_path=None*)

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.

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.apikey module

This contains functions to drive API key related auth actions.

authnzerver.actions.apikey.**issue_new_apikey**(*payload*, *raiseonfail=False*, *override_authdb_path=None*)

Issues a new API key.

Parameters

- **payload** (*dict*) – The payload dict must have the following keys:
  - 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.

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.**verify_apikey**(*payload*, *raiseonfail=False*, *override_authdb_path=None*)

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.
- **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.

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.**authnzerver_send_email**(*sender*, *subject*, *text*, *recipients*, *server*, *user*, *password*, *pii_salt*, *port=587*)

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.
- **port** (*int*) – The SMTP port to use when logging into the email server via SMTP.

Returns Returns True if email sending succeeded. False otherwise.

Return type bool

authnzerver.actions.email.**send_forgotpass_verification_email**(*payload*, *raiseonfail=False*, *override_authdb_path=None*)

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 keys must be provided by a wrapper function to set up the email server.
  - smtp_user
  - smtp_pass
  - smtp_server
  - smtp_port
  - smtp_sender

  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.

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

Return type dict

authnzerver.actions.email.**send_signup_verification_email**(*payload*, *raiseonfail=False*, *override_authdb_path=None*)

This actually sends the 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: str, 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 keys must be provided by a wrapper function to set up the email server.

- smtp_user
- smtp_pass
- smtp_server
- smtp_port
- smtp_sender

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.

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

Return type dict

authnzerver.actions.email.**verify_user_email_address**(*payload*, *raiseonfail=False*, *override_authdb_path=None*)

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' 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.

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.session module

This contains functions to drive session-related auth actions.

authnzerver.actions.session.**auth_delete_sessions_userid**(*payload*, *override_authdb_path=None*, *raiseonfail=False*)

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.

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=7*, *override_authdb_path=None*, *raiseonfail=False*)

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.

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

Return type dict

authnzerver.actions.session.**auth_password_check**(*payload*, *override_authdb_path=None*, *raiseonfail=False*)

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.

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

Return type dict

authnzerver.actions.session.**auth_session_delete**(*payload*, *override_authdb_path=None*, *raiseonfail=False*)

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.

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*, *override_authdb_path=None*, *raiseonfail=False*)

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.

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*, *override_authdb_path=None*, *raiseonfail=False*)

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.

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.**auth_session_set_extrainfo**(*payload*, *raiseonfail=False*, *override_authdb_path=None*)

Adds info to the extra_info_json key of a session column.

Parameters

- **payload** (*dict*) – This should contain the following items:
  - session_token : str, the session token to update
  - extra_info : dict, the update dict to put into the extra_info_json

  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.

Returns Returns a dict containing the new session info dict.

Return type dict

authnzerver.actions.session.**auth_user_login**(*payload*, *override_authdb_path=None*, *raiseonfail=False*)

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.

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

Return type dict

authnzerver.actions.session.**auth_user_logout**(*payload*, *override_authdb_path=None*, *raiseonfail=False*)

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.

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

Return type dict

authnzerver.actions.user module

This contains functions to drive user account related auth actions.

authnzerver.actions.user.**change_user_password**(*payload*, *override_authdb_path=None*, *raiseonfail=False*, *min_pass_length=12*, *max_similarity=30*)

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.
- **max_similarity** (*int*) – The maximum UQRatio required to fuzzy-match the input password against the server's domain name, the user's email, or their name.

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.user.**create_new_user**(*payload*, *min_pass_length=12*, *max_similarity=30*, *override_authdb_path=None*, *raiseonfail=False*)

Makes a new user.

Parameters

- **payload** (*dict*) – This is a dict with the following required keys:
  - full_name: 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*) – 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_similarity** (*int*) – The maximum UQRatio required to fuzzy-match the input password against the server's domain name, the user's email, or their name.

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

Return type dict

Notes

The emailverify_sent_datetime is set to the current time. The initial account's is_active is set to False and user_role is set to 'locked'.

The email verification token sent by the frontend expires in 2 hours. If the user doesn't get to it by then, they'll have to wait at least 24 hours until another one can be sent.

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 24 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*, *raiseonfail=False*, *override_authdb_path=None*)

Deletes a user.

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

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.

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

Return type dict

authnzerver.actions.user.**validate_input_password**(*full_name*, *email*, *password*, *pii_salt*, *min_length=12*, *max_match_threshold=20*)

Validates user input passwords.

1. must be at least min_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_match_threshold of their email or full_name
3. must not match within max_match_threshold 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

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.
- **min_length** (*int*) – The minimum required character length of the password.
- **max_match_threshold** (*int*) – The maximum UQRatio required to fuzzy-match the input password against the server's domain name, the user's email, or their name.

Returns Returns True if the password is OK to use and meets all specification. False otherwise.

Return type bool

authnzerver.actions.user.**verify_password_reset**(*payload*, *raiseonfail=False*, *override_authdb_path=None*, *min_pass_length=12*, *max_similarity=30*)

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_similarity** (*int*) – The maximum UQRatio required to fuzzy-match the input password against the server's domain name, the user's email, or their name.

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

Return type dict

1.1.2 Submodules

authnzerver.authdb module

This contains SQLAlchemy models for the authnzerver.

authnzerver.authdb.**create_authdb**(*authdb_url*, *database_metadata=MetaData(bind=None)*, *echo=False*, *returnconn=False*)

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

authnzerver.authdb.**create_sqlite_authdb**(*auth_db_path*, *database_metadata=MetaData(bind=None)*, *echo=False*, *returnconn=False*)

This creates the local SQLite auth DB.

authnzerver.authdb.**get_auth_db**(*authdb_path*, *database_metadata=MetaData(bind=None)*, *echo=False*)

This just gets a connection to the auth DB.

authnzerver.authdb.**initial_authdb_inserts**(*auth_db_path*, *permissions_json=None*, *database_metadata=MetaData(bind=None)*, *superuser_email=None*, *superuser_pass=None*, *echo=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*, *database_url=None*, *interactive=False*)

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.

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

Return type tuple of str

authnzerver.cache module

This contains functions to drive the cache.

`authnzerver.cache.`**`cache_add`**(*key*, *value*, *timeout_seconds=0.3*, *expires_seconds=None*, *cache_dirname='/tmp/authnzerver-cache'*)

This sets a key to the value specified in the cache.

`authnzerver.cache.`**`cache_decrement`**(*key*, *timeout_seconds=0.3*, *cache_dirname='/tmp/authnzerver-cache'*)

This decrements the counter for key.

`authnzerver.cache.`**`cache_delete`**(*key*, *timeout_seconds=0.3*, *cache_dirname='/tmp/authnzerver-cache'*)

This sets a key to the value specified in the cache.

authnzerver.cache.**cache_flush**(*timeout_seconds=0.3*, *cache_dirname='/tmp/authnzerver-cache'*)

This removes all keys from the cache.

authnzerver.cache.**cache_get**(*key*, *timeout_seconds=0.3*, *cache_dirname='/tmp/authnzerver-cache'*)

This sets a key to the value specified in the cache.

authnzerver.cache.**cache_getrate**(*key*, *timeout_seconds=0.3*, *cache_dirname='/tmp/authnzerver-cache'*)

This gets the rate of increment for the key by looking at the time of insertion inserted at key and the number of times it was incremented in key-counter. The rate is then:

key-counter_val/((time_now - time_insertion)/60.0)

authnzerver.cache.**cache_increment**(*key*, *timeout_seconds=0.3*, *cache_dirname='authnzerver-cache'*)

This sets up a counter for the key in the cache.

Sets the key -> time of initial insertion Then increments 'key-counter'.

authnzerver.cache.**cache_pop**(*key*, *timeout_seconds=0.3*, *cache_dirname='/tmp/authnzerver-cache'*)

This sets a key to the value specified in the cache.

authnzerver.confload module

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

authnzerver.confload.**get_conf_item**(*env_key*, *environment*, *options_object*, *options_key=None*, *vartype=<class 'str'>*, *default=None*, *readable_from_file=False*, *postprocess_value=None*, *raiseonfail=True*, *basedir=None*)

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*) – The environment variable that specifies the item to get.
- **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.
- **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.
- **options_key** (*str*) – This is the attribute to look up in the options object for the 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*, *file_spec*, *basedir=None*)

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*, *url_spec*, *environment*, *timeout=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*, *options_object*, *envfile=None*)

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 `'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 *`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.

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`

This handles the actual auth requests.

initialize(*config*, *executor*, *reqid_cache*, *failed_passchecks*)

This sets up stuff.

post()

Handles the incoming POST request.

class authnzerver.handlers.**EchoHandler**(*application: tornado.web.Application*, *request: tornado.httputil.HTTPServerRequest*, ***kwargs*)

Bases: tornado.web.RequestHandler

This just echoes back whatever we send.

Useful to see if the encryption is working as intended.

initialize(*authdb*, *fernet_secret*, *executor*)
This sets up stuff.

post()
Handles the incoming POST request.

class authnzerver.handlers.**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.handlers.**auth_echo**(*payload*)
This just echoes back the payload.

authnzerver.handlers.**check_host**(*remote_ip*)
This just returns False if the remote_ip != 127.0.0.1

authnzerver.handlers.**decrypt_request**(*requestbody_base64*, *fernet_key*)
This decrypts the incoming request.

authnzerver.handlers.**encrypt_response**(*response_dict*, *fernet_key*)
This encrypts the outgoing response.

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.permissions module

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

authnzerver.permissions.**check_item_access**(*permissions_model*, *userid=2*, *role='anonymous'*, *action='view'*, *target_name='collection'*, *target_owner=1*, *target_visibility='private'*, *target_sharedwith=None*, *debug=False*)

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

Parameters

- **permissions_policy** (*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*, *role*, *limit_name*, *value_to_check*)

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*, *role_name*, *target_name*, *target_visibility*, *target_ownership*, *debug=False*)

Returns the possible actions for a target given a role and target status.

Parameters

- **permissions_policy** (*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*)

Loads a permissions JSON and returns the model.

authnzerver.permissions.**load_policy_and_check_access**(*permissions_json*, *userid=2*, *role='anonymous'*, *action='view'*, *target_name='collection'*, *target_owner=1*, *target_visibility='private'*, *target_sharedwith=None*, *debug=False*)

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_policy** (*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.**load_policy_and_check_limits**(*permissions_json*, *role*, *limit_name*, *value_to_check*)

Applies the role limits to a value to check.

This version loads a policy JSON every time it is called.

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.**pii_hash**(*item*, *salt*)

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.

Error messages, data and custom validation code used in django-registration's various user-registration form classes.

authnzerver.validators.**normalize_value**(*value*)
This normalizes a given value and casefolds it.

Assumes that the value has already passed validation.

authnzerver.validators.**validate_confusables**(*value*)
This validates if the value is not a confusable homoglyph.

authnzerver.validators.**validate_confusables_email**(*value*)
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*)

This validates an email address using the HTML5 specification, which is good enough for most purposes.

The regex is taken from here:

http://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*)

This validates if the value is not one of the reserved names.

authnzerver.validators.**validate_unique_value**(*value*, *check_list*)

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 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

Index

E

F

G

I

L

M

N

P

S

T

V