Keg-Auth

Release 0.7.2

May 22, 2023

Documentation:

CHAPTER 1

Getting Started

1.1 Installation

- Bare functionality: *pip install keg-auth*
- With mail (i.e. with a mail manager configured, see below): *pip install keg-auth[mail]*
- JWT (for using JWT tokens as authenticators): *pip install keg-auth[jwt]*
- LDAP (for using LDAP target for authentication): *pip install keg-auth[ldap]*
- OAuth (e.g. Google Auth): *pip install keg-auth[oauth]*
- Internationalization extensions: *pip install keg-auth[i18n]*

1.2 Configuration

- `SERVER_NAME = 'somehost'`: Required for Keg Auth when generating URL in create-user CLI command
  - include a port number if needed (e.g. *localhost:5000*)
- **`PREFERRED_URL_SCHEME = 'https'`: This is important so that generated auth related URLS are** secure. You could have an SSL redirect but by the time that would fire, the key would have already been sent in the URL.
- `KEGAUTH_TOKEN_EXPIRE_MINS`: Integer, defaults to 240 minutes (4 hours)
  - If mail functions are enabled and tokens in the model, affects the time a verification token remains valid
- `KEGAUTH_CLI_USER_ARGS`: List of strings, defaults to *['email']*
  - Names arguments to be accepted by CLI user commands and passed to the model
- `KEGAUTH_HTTP_METHODS_EXCLUDED`: List of HTTP methods to exclude from auth checks
  - Useful for CORS-applicable situations, where it may be advantageous to respond normally to an OPTIONS request. Then, auth will apply as expected on the ensuing GET/POST/PUT/etc.
- `KEGAUTH_LOGOUT_CLEAR_SESSION`: Flag to clear flask session on logout. Default True
- `KEGAUTH_CRUD_INCLUDE_TITLE`: Control whether form/grid CRUD templates render an h1 tag
- `KEGAUTH_TEMPLATE_TITLE_VAR`: Template var to set for use in a base template's head -> title tag
- `KEGAUTH_REDIRECT_LOGIN_TARGET`: If using the redirect authenticator (like for OAuth), set this to the target
- `KEGAUTH_OAUTH_PROFILES`: Set of OAuth config, see section below
- Email settings
  - `KEGAUTH_EMAIL_OPS_ENABLED`: Defaults to True if mail manager is given, controls all email ops
  - `KEGAUTH_EMAIL_SITE_NAME = 'Keg Application'`: Used in email body if mail is enabled
  - `KEGAUTH_EMAIL_SITE_ABBR = 'Keg App'`: Used in email subject if mail is enabled
  - Example message:
    - Subject: [Keg App] Password Reset Link
    - Body: Somebody asked to reset your password on Keg Application. If this was not you...

1.3 Extension Setup

- Set up an auth manager (in app setup or extensions)
- The entity registry hooks up user, group, bundle, and permission entities. You will need to create a registry to associate with the auth manager, and register your entities from the model (see model notes)
- Note that the mail_manager is optional. If a mail_manager is not given, no mail will be sent
- Permissions may be passed as simple string tokens, or as tuples of *(token, description)*
- Note, the `auth_manage` permission is not assumed to be present, and must be specified to be preserved during sync.

```
from flask_mail import Mail
from keg_auth import AuthManager, AuthMailManager, AuthEntityRegistry

mail_ext = Mail()
auth_mail_manager = AuthMailManager(mail_ext)
auth_entity_registry = AuthEntityRegistry()

_endpoints = {'after-login': 'public.home'}
permissions = (
    ('auth-manage', 'manage users, groups, bundles, and view permissions'),
    ('app-permission1', 'access view Foo'),
    ('app-permission2', 'access the Bar area'),
)

auth_manager = AuthManager(mail_manager=auth_mail_manager, endpoints=_endpoints,
                           entity_registry=auth_entity_registry,␣
↪permissions=permissions)
auth_manager.init_app(app)
```

1.4 Login Authenticators

Login Authenticators control validation of users.

- Includes logic for verifying a user from a login route, and other view-layer operations needed for user workflow (e.g. verifying email, password resets, etc.)
- Authenticator may be specified on the auth_manager:
  - 'KegAuthenticator' is the default primary authenticator, and uses username/password
  - `AuthManager(mail_ext, login_authenticator=LdapAuthenticator)`
- LDAP authentication
  - `from keg_auth import LdapAuthenticator`
  - Uses python-ldap, which needs to be installed: `pip install keg-auth[ldap]`
  - Additional config:
    - `KEGAUTH_LDAP_TEST_MODE`: When True, bypasses LDAP calls. Defaults to False
    - `KEGAUTH_LDAP_SERVER_URL`: Target LDAP server or list of servers to use for queries. If a list is given, authentication is attempted on each server in the given order until a successful query is made.
    - `KEGAUTH_LDAP_DN_FORMAT`: Format-able string to set up for the query

            - ex. `uid={},dc=example,dc=org`
- OAuth authentication
  - `from keg_auth import OAuthAuthenticator`
  - Uses additional dependencies: `pip install keg-auth[oauth]`
  - Leans on `authlib` for the OAuth client
    - A number of client configurations may be found at https://github.com/authlib/loginpass
  - Additional config:
    - `KEGAUTH_OAUTH_PROFILES`: list of OAuth provider profile dicts
    - Each profile should have the following keys:
      - `domain_filter`: string or list of strings
      - `id_field`: field in the resulting user info to use as the user identity
      - `oauth_client_kwargs`: `authlib` client configuration. All of these args will be passed.
    - Multiple providers are supported. Login will be served at `/login/<profile-name>`
    - If using a single provider and OAuth will be the only authenticator, consider mapping `/login` via the `RedirectAuthenticator` and setting `KEGAUTH_REDIRECT_LOGIN_TARGET`.
  - Domain exclusions
    - If an OAuth profile is given a domain filter, only user identities within that domain will be allowed to login via that provider.
    - Filtered domains will be disallowed from password login, if `KegAuthenticator` is the primary.
    - Filtered domains will also prevent a user's domain from being changed in user admin.

1.5 Request Loaders

Request Loaders run when a user is not in session. Each loader will look for identifying data in the request, such as an authentication header.

- `AuthManager(mail_ext, request_loaders=JwtRequestLoader)`
- Token authenticators, like JwtRequestLoader, have a *create_access_token* method
  - `token = auth_manager.get_request_loader('jwt').create_access_token(user)`
- JWT:
  - `from keg_auth import JwtRequestLoader`
  - uses flask-jwt-extended, which needs to be installed: `pip install keg-auth[jwt]`

1.6 Blueprints

Include an auth blueprint along with your app's blueprints, which includes the login views and user/group/bundle management. Requires AuthManager instance:

```
from keg_auth import make_blueprint
from my_app.extensions import auth_manager
auth_bp = make_blueprint(__name__, auth_manager)
```

1.7 CLI

An auth group is provided and set up on the app during extension init. You can extend the group by using the cli_group attribute on the app's auth_manager, but you need access to the app during startup to do that. You can use an event signal to handle this - just be sure your app's *visit_modules* has the location of the event.

```
# in app definition
visit_modules = ['.events']


# in events module
from keg.signals import init_complete

from my_app.cli import auth_cli_extensions


@init_complete.connect
def init_app_cli(app):
    auth_cli_extensions(app)


# in cli
def auth_cli_extensions(app):
    @app.auth_manager.cli_group.command('command-extension')
    def command_extension():
        pass
```

Built-in commands:

- `create-user`: Create a user record and (depending on config) send a verify email.
- Mail can be turned off with the *–no-mail* option
- Create a superuser with the *–as-superuser* option
- By default, has one required argument (email). If you wish to have additional arguments, put the list of arg names in *KEGAUTH_CLI_USER_ARGS* config
- `set-password`: Allows you to set/reset the password for a given username.
- `purge-attempts`: Reset login attempts on a user to clear blocking.

1.8 Model

Create entities using the existing mixins, and register them with keg_auth. - Note: the User model assumes that the entity mixed with UserMixin will have a PK id - Email address and token verification by email are in *UserEmailMixin*

- i.e. if your app will not use email token verification for passwords, leave that mixin out

```
from keg.db import db
from keg_elements.db.mixins import DefaultColsMixin, MethodsMixin
from keg_auth import UserMixin, UserEmailMixin, PermissionMixin, BundleMixin,␣
↪GroupMixin

from my_app.extensions import auth_entity_registry


class EntityMixin(DefaultColsMixin, MethodsMixin):
    pass


@auth_entity_registry.register_user
class User(db.Model, UserEmailMixin, UserMixin, EntityMixin):
    __tablename__ = 'users'


@auth_entity_registry.register_permission
class Permission(db.Model, PermissionMixin, EntityMixin):
    __tablename__ = 'permissions'

    def __repr__(self):
        return '<Permission id={} token={}>'.format(self.id, self.token)


@auth_entity_registry.register_bundle
class Bundle(db.Model, BundleMixin, EntityMixin):
    __tablename__ = 'bundles'


@auth_entity_registry.register_group
class Group(db.Model, GroupMixin, EntityMixin):
    __tablename__ = 'groups'
```

1.8.1 Migrations

Keg-Auth does not provide any model migrations out of the box. We want to be very flexible with regard to the type of auth model in the app, so migrations become the app developer's responsibility.

If you are using a migration library like `alembic`, you can autogenerate a migration after upgrading Keg-Auth to ensure any model updates from mixins are included.

__Note__: autogenerated migrations solve most of the problems, but if you are starting with an existing database that already has user records, you may have some data issues to resolve as well. The following are known issues:

- Email field is expected to have all lowercase data. The model type assumes that because email

addresses are not case-sensitive, it can coerce input to lowercase for comparison, and expects that persisted data matches that assumption.

1.9 Navigation Helpers

Keg-Auth provides navigation helpers to set up a menu tree, for which nodes on the tree are restricted according to the authentication/authorization requirements of the target endpoint.

Note: requirements are any class-level permission requirements. If authorization is defined by an instance-level `check_auth` method, that will not be used by the navigation helpers.

- Usage involves setting up a menu structure with NavItem/NavURL objects. Note that permissions on a route may be overridden for navigation purposes
- Menus may be tracked on the auth manager, which will reset their cached access on login/logout
- `keg_auth/navigation.html` template has a helper `render_menu` to render a given menu as a ul
  - `{% import "keg-auth/navigation.html" as navigation %}`
  - `render_menu(auth_manager.menus['main'])`
  - `render_menu(auth_manager.menus['main'], expand_to_current=True)`
  - Automatically expand/collapse menu groups for the currently-viewed item. Useful for vertical menus.
- Collapsible groups can be added to navigation menus by nesting NavItems in the menu. The group item will get a `nav_group` attribute, which can be referred to in CSS.
  - `NavItem('Auth Menu', NavItem(...))` will have a `nav_group` of `#navgroup-auth-menu`
  - `NavItem('Auth Menu', NavItem(...), nav_group='foo')` will have a `nav_group` of `#navgroup-foo`
- NavItems can specify an icon to display in the menu item by passing an `icon_class` string to the NavItem constructor. e.g., `NavItem('Title', NavURL(...), icon_class='fas fa-shopping-cart')`.
- NavItems can be given a `class_` kwarg that will be applied to the whole `li` tag in the default render. This applies to both group items and the menu links themselves.
- NavItems can also be provided a `code` kwarg, which is useful when doing custom templating to render the menu. The code is a code-only tag for the menu that can remain the same even if the menu wording changes. For example, the code could be used in a conditional template block to render certain menu items differently from the rest.

Example:

```
from keg.signals import init_complete

from keg_auth import NavItem, NavURL

@init_complete.connect
def init_navigation(app):
    app.auth_manager.add_navigation_menu(
        'main',
        NavItem(
            NavItem('Home', NavURL('public.home')),
            NavItem(
                'Nesting',
                NavItem('Secret1', NavURL('private.secret1')),
                NavItem('Secret1 Class', NavURL('private.secret1-class')),
                class_='my-nest-class',
            ),
            NavItem('Permissions On Stock Methods', NavURL('private.secret2')),
            NavItem('Permissions On Methods', NavURL('private.someroute')),
            NavItem('Permissions On Class And Method', NavURL('private.secret4')),
            NavItem('Permissions On NavURL',
                NavURL(
```

(continues on next page)

(continued from previous page)

```
                    'private.secret3', requires_permissions='permission3'
                )),
            NavItem('User Manage', NavURL('auth.user:add')),
            NavItem('Logout', NavURL('auth.logout'), code='i-am-different'),
            NavItem('Login', NavURL('auth.login', requires_anonymous=True)),
        )
    )
```

1.10 Templates

Templates are provided for the auth views, as well as base crud templates.

Base templates use keg-elements' form-view and grid-view parent templates. The app template to extend is referenced from settings. The first of these defined is used:

- *BASE_TEMPLATE*
- *KEG_BASE_TEMPLATE*

Keg-Auth will assume that a variable is used in the master template to determine the contents of a title block. That variable name defaults to `page_title`, but may be customized via `KEGAUTH_TEMPLATE_TITLE_VAR`.

1.11 Views

- Views may be restricted for access using the requires* decorators
- Each decorator can be used as a class decorator or on individual view methods
- Additionally, the decorator may be used on a Blueprint to apply the requirement to all routes on the blueprint
- `requires_user`
  - Require a user to be authenticated before proceeding (authentication only)
  - Usage: `@requires_user` or `@requires_user()` (both usage patterns are identical if no secondary authenticators are needed)
  - Note: this is similar to `flask_login.login_required`, but can be used as a class/blueprint decorator
  - You may pass a custom *on_authentication_failure* callable to the decorator, else it will redirect to the login page
  - A decorated class/blueprint may have a custom *on_authentication_failure* instance method instead of passing one to the decorator
  - `KEGAUTH_HTTP_METHODS_EXCLUDED` can be overridden at the individual decorator level by passing `http_methods_excluded` to the decorator's constructor
- `requires_permissions`
  - Require a user to be conditionally authorized before proceeding (authentication + authorization)
  - `has_any` and `has_all` helpers can be used to construct complex conditions, using string permission tokens, nested helpers, and callable methods
  - You may pass a custom *on_authorization_failure* callable to the decorator, else it will respond 403 Unauthorized

  - A decorated class/blueprint may have a custom *on_authorization_failure* instance method instead of passing one to the decorator
  - Usage:
    * `@requires_permissions(('token1', 'token2'))`
    * `@requires_permissions(has_any('token1', 'token2'))`
    * `@requires_permissions(has_all('token1', 'token2'))`
    * `@requires_permissions(has_all(has_any('token1', 'token2'), 'token3'))`
    * `@requires_permissions(custom_authorization_callable that takes user arg)`

- A standard CRUD view is provided which has add, edit, delete, and list "actions"
  - `from keg_auth import CrudView`
  - Because the standard action routes are predefined, you can assign specific permission(s) to them in the view's *permissions* dictionary, keyed by action (e.g. *permissions['add'] = 'foo'*)

1.12 Global Request Hooks

The authorization decorators will likely normally be used against view methods/classes and blueprints. However, another scenario for usage would be request hooks. For example, if authorization needs to be run across the board for any request, we can register a callback on that hook, and apply the decorator accordingly.

```
from keg.signals import app_ready

@app_ready.connect
def register_request_started_handler(app):
    from keg_auth.libs.decorators import requires_permissions

    @app.before_request
    @requires_permissions(lambda user: user.is_qualified)
    def request_started_handler(*args, **kwargs):
        # Nothing special needs to happen here - the decorator does it all
        pass
```

1.13 Attempt Limiting

Login, forgot password, and reset attempts are limited by registering an Attempt entity. The Attempt entity must be a subclass of *AttemptMixin*.

Attempt limiting is enabled by default, which requires the entity. But, it may be disabled in configuration.

Login attempts are limited by counting failed attempts. A successful login attempt will reset the limit counter. Reset attempts are limited by counting all password reset attempts.

Attempt limiting can be configured with the following options:

- `KEGAUTH_ATTEMPT_LIMIT_ENABLED`: primary config switch, default True.
- `KEGAUTH_ATTEMPT_LIMIT`: maximum number of attempts within the timespan, default 15.
- `KEGAUTH_ATTEMPT_TIMESPAN`: timespan in seconds in which the limit can be reached, default 10 minutes.

- KEGAUTH_ATTEMPT_LOCKOUT: timespan in seconds until a successful attempt can be made after the limit is reached, default 1 hour.
- KEGAUTH_ATTEMPT_IP_LIMIT: base locking on IP address as well as input, default True.
- KEGAUTH_LOGIN_ATTEMPT_LIMIT: overrides KEGAUTH_ATTEMPT_LIMIT for the login view.
- KEGAUTH_LOGIN_ATTEMPT_TIMESPAN: overrides KEGAUTH_ATTEMPT_TIMESPAN for the login view.
- KEGAUTH_LOGIN_ATTEMPT_LOCKOUT: overrides KEGAUTH_ATTEMPT_LOCKOUT for the login view.
- KEGAUTH_FORGOT_ATTEMPT_LIMIT: overrides KEGAUTH_ATTEMPT_LIMIT for the forgot password view.
- KEGAUTH_FORGOT_ATTEMPT_TIMESPAN: overrides KEGAUTH_ATTEMPT_TIMESPAN for the forgot password view.
- KEGAUTH_FORGOT_ATTEMPT_LOCKOUT: overrides KEGAUTH_ATTEMPT_LOCKOUT for the forgot password view.
- KEGAUTH_RESET_ATTEMPT_LIMIT: overrides KEGAUTH_ATTEMPT_LIMIT for the reset password view.
- KEGAUTH_RESET_ATTEMPT_TIMESPAN: overrides KEGAUTH_ATTEMPT_TIMESPAN for the reset password view.
- KEGAUTH_RESET_ATTEMPT_LOCKOUT: overrides KEGAUTH_ATTEMPT_LOCKOUT for the reset password view.

CLI *purge-attempts* will delete attempts for a given username. Optionally accepts *–attempt-type* argument to only delete attempts of a certain type.

1.14 Testing and User Login

This library provides `keg_auth.testing.AuthTestApp` which is a sub-class of `flask_webtest.TestApp` to make it easy to set the logged-in user during testing:

```
from keg_auth.testing import AuthTestApp

class TestViews(object):

    def setup_method(self):
        ents.User.delete_cascaded()

    def test_authenticated_client(self):
        """
            Demonstrate logging in at the client level.  The login will apply to all␣
↪requests made
            by this client.
        """
        user = ents.User.fake()
        client = AuthTestApp(flask.current_app, user=user)
        resp = client.get('/secret2', status=200)
        assert resp.text == 'secret2'

    def test_authenticated_request(self):
        """
            Demonstrate logging in at the request level.  The login will only apply␣
↪to one request.
```

(continues on next page)

(continued from previous page)

```
        """
        user = ents.User.fake(permissions=('permission1', 'permission2'))
        client = AuthTestApp(flask.current_app)

        resp = client.get('/secret-page', status=200, user=user)
        assert resp.text == 'secret-page'

        # User should only stick around for a single request (and will get a 302␣
↪redirect to the)
        # login view.
        client.get('/secret-page', status=302)
```

A helper class is also provided to set up a client and user, given the permissions specified on the class definition:

```
from keg_auth.testing import ViewTestBase

class TestMyView(ViewTestBase):
    permissions = 'permission1', 'permission2', ...

    def test_get(self):
        self.client.get('/foo')
```

1.15 Using Without Email Functions

Keg Auth is designed out of the box to use emailed tokens to:

- verify the email addresses on user records
- provide a method of initially setting passwords without the admin setting a known password

While this provides good security in many scenarios, there may be times when the email methods are not desired (for example, if an app will run in an environment where the internet is not accessible). Only a few changes are necessary from the examples above to achieve this:

- leave *UserEmailMixin* out of the *User* model
- do not specify a mail_manager when setting up *AuthManager*

1.16 Email/Reset Password Functionality

- **The JWT tokens in the email / reset password emails are salted with**
  - username/email (depends on which is enabled)
  - password hash
  - last login utc
  - is_active (verified/enabled combination)

  This allows for tokens to become invalidate anytime of the following happens:
  - username/email changes
  - password hash changes
  - a user logs in (last login utc will be updated and invalidate the token)

  - is active (depending on the model this is calculated from is_enabled/is_verified fields)

1.17 Internationalization

Keg-Auth supports *Babel*-style internationalization of text strings through the *morphi* library. To use this feature, specify the extra requirements on install:

```
pip install keg-auth[i18n]
```

Currently, English (default) and Spanish are the supported languages in the UI.

1.17.1 Helpful links

- https://www.gnu.org/software/gettext/manual/html_node/Mark-Keywords.html
- https://www.gnu.org/software/gettext/manual/html_node/Preparing-Strings.html

1.17.2 Message management

The `setup.cfg` file is configured to handle the standard message extraction commands. For ease of development and ensuring that all marked strings have translations, a tox environment is defined for testing i18n. This will run commands to update and compile the catalogs, and specify any strings which need to be added.

The desired workflow here is to run tox, update strings in the PO files as necessary, run tox again (until it passes), and then commit the changes to the catalog files.

```
tox -e i18n
```

CHAPTER 2

Upgrading Keg-Auth

While we attempt to preserve backward compatibility, some KegAuth versions do introduce breaking changes. This list should provide information on needed app changes.

- 0.6.0 - OIDC authenticator (deprecated) has been removed. Use OAuth with configured profiles instead. - Model `testing_create` renamed to `fake` (follows KegElements 0.8.0)
  - Template files now follow keg's more recent naming scheme to use dashes instead of underscores. E.g. `keg_auth/crud-list.html` became `keg-auth/crud-list.html`
  - `keg-elements/form-view.html` and `keg-elements/grid-view.html` are now available, so are used as bases for CRUD templates. `form-base.html` has been removed.
  - KegElements now handles Select2 css/js inclusion in its `form-view.html` template. That has been removed from KegAuth.

CHAPTER 3

Core

class keg_auth.core.**AuthManager**(*mail_manager=None*, *blueprint='auth'*, *endpoints=None*, *cli_group_name=None*, *grid_cls=None*, *login_authenticator=<class 'keg_auth.libs.authenticators.KegAuthenticator'>*, *request_loaders=None*, *permissions=None*, *entity_registry=None*, *oauth_authenticator=<class 'keg_auth.libs.authenticators.OAuthAuthenticator'>*, *password_policy_cls=<class 'keg_auth.libs.authenticators.DefaultPasswordPolicy'>*)

Set up an auth management extension

Main manager for keg-auth authentication/authorization functions, and provides a central location and handle on the flask app to access CLI setup, navigation, authenticators, etc.

Parameters

- **mail_manager** – AuthMailManager instance used for mail functions. Can be None.
- **blueprint** – name to use for the blueprint containing auth views
- **endpoints** – dict of overrides to auth view endpoints
- **cli_group_name** – name of the CLI group under which auth commands appear
- **grid_cls** – webgrid class to serve as a base class to auth CRUD grids
- **login_authenticator** – login authenticator class used by login view default: KegAuthenticator
- **request_loaders** – registered loaders used for loading a user at request time from information not contained in the session (e.g. with an authorization header token). Can be scalar or an iterable
- **permissions** – permission strings defined for the app, which will be synced to the database on app init. Can be a single string or an iterable
- **entity_registry** – EntityRegistry instance on which User, Group, etc. are registered

- **password_policy_cls** – A PasswordPolicy class to check password requirements in forms and CLI

add_navigation_menu(*name*, *menu*)
Create a navigation menu that may be referenced with the given name.

create_user(*user_kwargs*, *_commit=True*)
Create a new user record and optionally persist to the database.

Parameters

- **user_kwargs** – dict of values to construct the User record. Special arg is *mail_enabled*, which will be popped out.
- **_commit** – option for persisting record to database. Default True.

create_user_cli(*extra_args=None*, ***kwargs*)
A thin layer between the cli and *create_user()* to transform the cli args into what the User entity expects for fields.

For example, if you had a required *name* field on your User entity, then you could do something like:

```
$ yourkegapp auth create-user john.smith@example.com "John Smith"
```

Then this method would get overriden in a sub-class like:

def create_user_cli(self, email, extra_args): user_kwargs = dict(email=email, name=extra_args[0]) return self.create_user(user_kwargs)

endpoint(*ident*)
Return an auth endpoint on the configured blueprint.

get_request_loader(*identifier*)
Returns a registered request loader, keyed by its identifier.

init_app(*app*)
Inits KegAuth as a flask extension on the given app.

init_cli(*app*)
Add a CLI group for auth.

init_config(*app*)
Provide app config defaults for crypto, mail, logins, etc.

init_jinja(*app*)
Set up app jinja loader to use keg-auth templates, select2, etc.

init_loaders(*app*)
Initialize user session loaders.

init_managers(*app*)
Place this extension on the app for reference, and onfigure flask-login.

init_model(*app*)
Set up the entity registry for all auth objects.

init_permissions(*app*)
Configure database with the defined set of permissions.

Synchronizes permission records in the database with those defined in the app. Ensures the sync method is called in the proper place during test runs, when the database is not fully available and set up at extension-loading time.

resend_verification_email(*user_id*)
Generate a fresh token and send the account verification email.

test_request_loader(*request*)
Load a user from a request when testing. This gives a nice API for test clients to be logged in, rather than expecting all tests to set up an actual session.

See *keg_auth.testing.AuthTestApp* for a webtest wrapper using this loader.

url_for(*ident*, ***kwargs*)
Generate the URL for the endpoint identified by *ident*.

user_by_id(*user_id*)
Fetch a user record via ID.

user_loader(*session_key*)
Fetch a user record via session key.

CHAPTER 4

Model

4.1 Entity Registry

class keg_auth.model.entity_registry.**EntityRegistry**(*user=None*, *permission=None*, *bundle=None*, *group=None*, *attempt=None*)

Registry cache that identifies entities for particular usage/function in KegAuth.

KegAuth does not provide its own entities for the database model. Instead, mixins are given so that an application can customize as needed. To support this model and still know what entity to use, we register it in an EntityRegistry.

Entities may be registered in one of two ways: - Mark an entity with a registration decorator:

```
@registry.register_user
class User(UserMixin, db.EntityBase):
    pass
```

- Pass the entity into the EntityRegistry constructor:

```
registry = EntityRegistry(user=User, permission=Permission, ...)
```

Registered entities may be subsequently referenced via the properties, e.g.

registry.user_cls

attempt_cls
Return the entity registered for Attempt.

bundle_cls
Return the entity registered for Bundle.

group_cls
Return the entity registered for Group.

is_registered(*type*)
Helper for determining if functionality is unlocked via a registered entity.

permission_cls
Return the entity registered for Permission.

register_attempt(*cls*)
Mark given class as the entity for Attempt.

register_bundle(*cls*)
Mark given class as the entity for Bundle.

register_group(*cls*)
Mark given class as the entity for Group.

register_permission(*cls*)
Mark given class as the entity for Permission.

register_user(*cls*)
Mark given class as the entity for User.

user_cls
Return the entity registered for User.

exception keg_auth.model.entity_registry.**RegistryError**

4.2 Utils

class keg_auth.model.utils.**AllCondition**(**conditions*)
Condition requiring all contained permissions/conditions to be satisfied.

Rules governing the contained permission/conditions: - Not all conditions are guaranteed to be checked. Checking will exit on the first failure. - Callable conditions are expected to take a *user* argument. - Permission token conditions are run if the user has a *has_all_permissions* method (default case).

class keg_auth.model.utils.**AnyCondition**(**conditions*)
Condition requiring only one of the contained permissions/conditions to be satisfied.

Rules governing the contained permission/conditions: - Not all conditions are guaranteed to be checked. Checking will exit on the first success. - Callable conditions are expected to take a *user* argument. - Permission token conditions are run if the user has a *has_all_permissions* method (default case).

class keg_auth.model.utils.**PermissionCondition**(**conditions*)
Basic permission condition that will return True/False upon user access check.

One or more permissions or conditions must be provided to the constructor.

keg_auth.model.utils.**has_all**
alias of *keg_auth.model.utils.AllCondition*

keg_auth.model.utils.**has_any**
alias of *keg_auth.model.utils.AnyCondition*

keg_auth.model.utils.**has_permissions**(*condition*, *user*)
Check a user against a single condition/permission.

class keg_auth.model.__init__.**AttemptMixin**
Generic mixin for logging user login attempts.

classmethod purge_attempts(*username=None*, *older_than=None*, *attempt_type=None*)
Delete attempt records optionally filtered by username, age, or type.

class keg_auth.model.__init__.**BundleMixin**
Generic mixin for permission bundles.

class keg_auth.model.__init__.**GroupMixin**
Generic mixin for user groups.

exception keg_auth.model.__init__.**InvalidToken**

class keg_auth.model.__init__.**KAPasswordType**(*max_length=None*, ***kwargs*)

load_dialect_impl(*dialect*)
Return a TypeEngine object corresponding to a dialect.

This is an end-user override hook that can be used to provide differing types depending on the given dialect. It is used by the TypeDecorator implementation of type_engine() to help determine what type should ultimately be returned for a given TypeDecorator.

By default returns self.impl.

class keg_auth.model.__init__.**PermissionMixin**
Generic mixin for permissions.

class keg_auth.model.__init__.**UserEmailMixin**
Mixin for users who will be authenticated by email/password.

change_password(*token*, *new_password*)
Change a password based on token authorization.

class keg_auth.model.__init__.**UserMixin**
Generic mixin for user entities.

get_token_salt()
Create salt data for password reset token signing. The return value will be hashed together with the signing key. This ensures that changes to any of the fields included in the salt invalidates any tokens produced with the old values Values included:

- **user login identifier -> if username/email change it will invalidate** the user token
- is_active -> Anytime a user verifies will invalidate a token
- current password hash or empty string if no password has been set -> If the password is updated we want to invalidate the token
- **last login time -> Any time a user logs in it will invalidate any** verification and reset password emails

Returns JSON string of list containing the values listed above

token_generate()
Create a new token for this user. The returned value is an expiring JWT signed with the application's crypto key. Externally this token should be treated as opaque. The value returned by this function must not be persisted. :return: a string representation of the generated token

token_verify(*token*, *_use_legacy=False*, *_block_legacy=False*)

Verify a password reset token. The token is validated for:

- user identity
- tampering
- expiration

- password was not already reset since token was generated
- user has not signed in since token was generated

Parameters **token** – string representation of token to verify

Returns bool indicating token validity

class keg_auth.model.__init__.**UserTokenMixin**

Mixin for users who will be authenticated by tokens.

classmethod generate_raw_auth_token(*length=32*, *entropy=None*, *charset='ascii_50'*)

Return a raw authentication token

NOTE(nZac): You should not store this directly in the database. When using this mixin, simply setting this value to `self.token = generate_raw_auth_token` is enough (though, there is a helper method for that `reset_auth_token`).

reset_auth_token(***kwargs*)

Reset the authentication token for this user

Takes the same parameter as *:cls:generate_auth_token*

keg_auth.model.__init__.**get_username**(*user*)

Based on the registered user entity, find the column representing the login ID.

CHAPTER 5

Testing

class keg_auth.testing.**AuthAttemptTests**

Tests to verify that automated attempt logging/blocking works as intended. These tests are included in the AuthTests class and are intended to be used in target applications to verify customization hasn't broken basic KegAuth functionality.

test_forgot_attempts_blocked(*limit*, *timespan*, *lockout*)

Test that forgot attempts get blocked after reaching the failed forgot attempt limit. forgot attempts after the lockout period has passed (since the failed attempt that caused the lockout) should not be blocked.

test_forgot_attempts_not_blocked()

Test that we do not block any attempts with missing attempt entity.

test_login_attempts_blocked(*limit*, *timespan*, *lockout*, *create_user*, *view_config*)

Test that login attempts get blocked after reaching the failed login attempt limit. Login attempts after the lockout period has passed (since the failed attempt that caused the lockout) should not be blocked.

test_login_attempts_blocked_but_not_configured(*_*)

Test that we check for the registered attempt entity when limiting is enabled.

test_login_attempts_blocked_by_ip(*limit*, *timespan*, *lockout*)

Test that login attempts get blocked for an IP address

test_login_attempts_not_blocked()

Test that we do not block any attempts with missing attempt entity.

test_reset_pw_attempts_blocked(*limit*, *timespan*, *lockout*)

Test that login attempts get blocked after reaching the failed login attempt limit. Login attempts after the lockout period has passed (since the failed attempt that caused the lockout) should not be blocked.

test_successful_forgot_resets_attempt_counter(*limit*, *timespan*, *lockout*)

Test that several failed forgots before a successful forgot do not count towards the attempt lockout counter.

test_successful_login_resets_attempt_counter(*limit*, *timespan*, *lockout*)

Test that several failed logins before a successful login do not count towards the attempt lockout counter.

class keg_auth.testing.**AuthTestApp**(*app*, ***kwargs*)

Wrapper of *flask_webtest.TestApp* that will inject a user into the session.

Pass in a user instance to "log in" the session:

user = User.fake(permissions=['auth-manage', 'do-something']) test_app = AuthTestApp(flask.current_app, user=user)

When running integration tests, following the view sequence to log a user in can be quite time-consuming and unnecessary. Login tests can be elsewhere. Once a user is logged in, they are identified by their *session_key*. So, we simply inject that key in the environment, and then follow the request out to webtest per normal.

class keg_auth.testing.**AuthTests**

These tests are designed so they can can be imported into an application's tests and ran to ensure customization of KegAuth hasn't broken basic functionality.

test_next_parameter_not_open_redirect()

ensure following the "next" parameter doesn't allow for an open redirect

class keg_auth.testing.**ViewTestBase**

Simple helper class that will set up Permission tokens as specified, log in a user, and provide the test app client on the class for use in tests.

Usage: `permissions` class attribute can be scalar or list.

For tests:

- `self.current_user`: User instance that is logged in
- `self.client`: AuthTestApp instance

classmethod create_user()

Creates a User record for tests. By default, simply calls `fake` with permissions.

classmethod setup_user()

Hook to do further setup on `cls.current_user`.

keg_auth.testing.**with_crypto_context**(*field*, *context=None*)

Wrap a test to use a real cryptographic context for a `KAPasswordType`

Temporarily assign a `passlib.context.CryptoContext` to a particular entity column.

Parameters (optional) (*context*) – `passlib.context.CryptoContext` to use for this test. The default value is *keg_auth.core.DEFAULT_CRYPTO_SCHEMES*.

In most situations we don't want a real crypto scheme to run in the tests, it is slow on entities like Users which have a password. `User.fake` will generate a value for that instance and then hash which takes a bunch of time. However, when testing certain schemes, it is useful to execute the real behavior instead of the `plaintext` behaviour.

```
import bcrypt

bcrypt_context = passlib.context.CryptContext(scheme=['bcrypt'])

@with_crypto_context(ents.User.password, context=bcrypt_context)
def test_with_real_context():
    user = ents.User.fake(password='abc')
    assert bcrypt.checkpw('abc', user.password.hash)
```

CHAPTER 6

Views

class keg_auth.views.**AuthRespondedView**
Base for views which will refer out to the login authenticator for responders

URL gets calculated from the responder class and must be a class attribute there.

Note: if the login authenticator doesn't have the referenced key, the view will 404.

classmethod calc_url(***kwargs*)
Leans on login authenticator's responders to provide a URL.

on_missing_responder()
Handler for requests that do not match a responder in authenticator.

By default, aborts with 404 response.

responder(**args*, ***kwargs*)
Refer all requests to the responder and return the response.

If no responder, call *on_missing_responder*.

class keg_auth.views.**Bundle**(**args*, ***kwargs*)
Default Bundle CRUD view. Uses auth-manage permission for all targets.

create_form(*obj*)
Create an instance of *form_cls*. Must return a form if overloaded.

obj is an instance of *orm_cls* (edit) or None (add).

static form_cls(*endpoint*)
Returns a form for Bundle CRUD.

update_obj(*obj*, *form*)
Update an existing object instance from form data. Does not explicitly flush or commit.

class keg_auth.views.**CrudView**(**args*, ***kwargs*)
Base CRUD view class providing add/edit/delete/list functionality.

Basic subclass setup involves: - set the *grid_cls*, *form_cls*, and *orm_cls* attributes - set *object_name* to be the human readable label. - assign *object_name_plural* only if necessary - assign base permissions for reach of the four endpoints

Grid is assumed to be WebGrid. Form is assumed to be WTForms. ORM is assumed to be SQLAlchemy. Default templates are provided with keg-auth.

Permissions are set for each endpoint under the *permissions* dict attribute. Note that it is usually helpful to put a general @requires_permissions on the class itself, as that will aid in conditionally displaying navigation links based on a user's access level.

add()
View method for add. Enforce permissions and call *add_edit*.

add_edit(*meth*, *obj=None*)
Handle form-related requests for add/edit.

Form instance comes from *create_form*. Valid form updates the object via *update_obj*. If post successful, returns result of *on_add_edit_success*. If post failure, runs *on_add_edit_failure* and renders the form via *render_form*. If get, renders the form via *render_form*.

add_orm_obj()
Generate a blank object instance and add it to the session.

add_url_with_session(*session_key*)
Return add url with the session_key from the request, to support webgrid sessions.

cancel_url()
Return list url with the session_key from the request, to support webgrid sessions.

create_form(*obj*)
Create an instance of *form_cls*. Must return a form if overloaded.

obj is an instance of *orm_cls* (edit) or None (add).

delete(*objid*)
View method for delete. Enforce permissions, load the record, run ORM delete.

If delete succeeds, return result of *on_delete_success*. If delete fails, return result of *on_delete_failure*.

edit(*objid*)
View method for edit. Enforce permissions, load the record, and call *add_edit*.

classmethod endpoint_for_action(*action*)
Compute the flask endpoint for the given CRUD action.

flash_success(*verb*)
Add a flask flash message for success with the given *verb*.

form_page_heading(*action*)
Allows customization of add/edit heading. Defaults to *page_title*.

form_template_args(*arg_dict*)
Allows customization of jinja template args for add/edit views.

arg_dict contains the default arguments, including anything set with *self.assign*.

Must return a dict of template args.

grid_page_heading
Allows customization of grid heading. Defaults to *page_title*.

grid_template_args(*arg_dict*)
Allows customization of jinja template args for list view.

arg_dict contains the default arguments, including anything set with *self.assign*.

Must return a dict of template args.

init_object(*obj_id*, *action=None*)
Load record from ORM for edit/delete cases.

Forces 404 response if the record does not exist.

Additional object-loading customization can be provided on action-specific hooks *init_object_edit* and *init_object_delete*. These methods will take no parameters, but they may assume *self.objinst* refers to the requested entity.

classmethod init_routes()
Creates the standard set of routes from methods (add, edit, delete, list).

To extend to further action routes: *cls.map_method_route(method_name, url, HTTP methods)* ex. *cls.map_method_route('read', '/foo', ('GET',))*

list()
View method for list. Enforce permissions, then render grid via *render_grid*.

list_url_with_session
Return list url with the session_key from the request, to support webgrid sessions.

make_grid()
Create an instance of *grid_cls* and initialize from request.

Returns a grid instance.

object_name_plural
Plural version of *object_name*. Uses the inflect library for a default value.

on_add_edit_failure(*entity*, *is_edit*)
Flash an add/edit message. No redirect in this case.

on_add_edit_success(*entity*, *is_edit*)
Flash an add/edit success message, and redirect to list view.

on_delete_failure()
Flash a delete failure message, and redirect to list view.

on_delete_success()
Flash a delete success message, and redirect to list view.

on_render_limit_exceeded(*grid*)
Flash a message for webgrid limit exceeded case.

This gets run in export cases where more records are in the set than the file format can support.

page_title(*action*)
Generates a heading title based on the page action.

action should be a string. Values "Create" and "Edit" are handled, with a fall-through to return *object_name_plural* (for the list case).

post_args_grid_setup(*grid*)
Apply changes to grid instance after QS args/session are loaded.

render_form(*obj*, *action*, *form*, *action_button_text='Save Changes'*)
Renders the form template.

Template arguments may be customized with the *form_template_args* method.

render_grid()
Renders the grid template.

Grid instance comes from *make_grid*. Grid instance may be customized via *post_args_grid_setup*. If grid is set to export, give that response or handle the limit exceeded error. Otherwise, render *grid_template* with *grid_template_args*.

update_obj(*obj*, *form*)
Update an existing object instance from form data. Does not explicitly flush or commit.

class keg_auth.views.**ForgotPassword**
Forgot Password view that uses the login authenticator's responders.

class keg_auth.views.**Group**(**args*, ***kwargs*)
Default Group CRUD view. Uses auth-manage permission for all targets.

create_form(*obj*)
Create an instance of *form_cls*. Must return a form if overloaded.

obj is an instance of *orm_cls* (edit) or None (add).

static form_cls(*endpoint*)
Returns a form for Group CRUD.

update_obj(*obj*, *form*)
Update an existing object instance from form data. Does not explicitly flush or commit.

class keg_auth.views.**Login**
Login view that uses the login authenticator's responders.

class keg_auth.views.**Logout**
Logout view that uses the login authenticator's responders.

class keg_auth.views.**OAuthAuthorize**
Authorization view that uses the OAuth authenticator's responders.

Completes the OAuth login flow.

class keg_auth.views.**OAuthLogin**
Login view that uses the OAuth authenticator's responders.

class keg_auth.views.**Permission**(*responding_method=None*)
Default Permission view. Uses auth-manage permission.

class keg_auth.views.**ResetPassword**
Reset Password view that uses the login authenticator's responders.

class keg_auth.views.**User**(**args*, ***kwargs*)
Default User CRUD view. Uses auth-manage permission for all targets.

create_form(*obj*)
Create an instance of *form_cls*. Must return a form if overloaded.

obj is an instance of *orm_cls* (edit) or None (add).

delete(*objid*)
View method for delete. Enforce permissions, load the record, run ORM delete.

If delete succeeds, return result of *on_delete_success*. If delete fails, return result of *on_delete_failure*.

static form_cls(*config=None, allow_superuser=False, endpoint='', fields=['is_enabled', 'disabled_utc']*)
Returns a form for User CRUD.

Form is customized depending on the fields and superuser setting passed in.

update_obj(*obj*, *form*)
Update an existing object instance from form data. Does not explicitly flush or commit.

class keg_auth.views.**VerifyAccount**
Verification view that uses the login authenticator's responders.

keg_auth.views.**make_blueprint**(*import_name*, *_auth_manager*, *bp_name='auth'*, *login_cls=<class 'keg_auth.views.Login'>*, *forgot_cls=<class 'keg_auth.views.ForgotPassword'>*, *reset_cls=<class 'keg_auth.views.ResetPassword'>*, *logout_cls=<class 'keg_auth.views.Logout'>*, *verify_cls=<class 'keg_auth.views.VerifyAccount'>*, *user_crud_cls=<class 'keg_auth.views.User'>*, *group_crud_cls=<class 'keg_auth.views.Group'>*, *bundle_crud_cls=<class 'keg_auth.views.Bundle'>*, *permission_cls=<class 'keg_auth.views.Permission'>*, *oauth_login_cls=<class 'keg_auth.views.OAuthLogin'>*, *oauth_auth_cls=<class 'keg_auth.views.OAuthAuthorize'>*, *blueprint_class=<class 'flask.blueprints.Blueprint'>*, ***kwargs*)
Blueprint factory for keg-auth views

Most params are assumed to be view classes. *_auth_manager* is the extension instance meant for the app on which this blueprint will be used: it is necessary in order to apply url routes for user functions.

blueprint_class is the class to be instantiated as the Flask blueprint for auth views. The default is flask.blueprint, but a custom blueprint may be provided.

CHAPTER 7

Libs

7.1 Authenticators

exception keg_auth.libs.authenticators.**AttemptBlocked**

class keg_auth.libs.authenticators.**DefaultPasswordPolicy**
A bare-bones, very permissive policy to use as a default if none is set on initialization.

class keg_auth.libs.authenticators.**ForgotPasswordViewResponder**(*parent*)
Master responder for keg-integrated logins, using an email form

form_cls
alias of *keg_auth.forms.ForgotPassword*

get_last_limiting_attempt(*username*)
Get the last attempt that counts toward the limit count. Attempts that count toward the limit before this attempt will be counted to determine if this attempt caused a lockout.

For login, this will be the last failed attempt. For password reset, this will be the last attempt.

get_limiting_attempt_count(*before_time*, *username*)
Return the number of attempts that count toward the limit up to before_time.

class keg_auth.libs.authenticators.**FormResponderMixin**
Wrap form usage for auth responders, contains GET and POST handlers

class keg_auth.libs.authenticators.**JwtRequestLoader**(*app*)
Loader for JWT tokens contained in the Authorization header.

Requires flask-jwt-extended (*pip install keg-auth[jwt]*)

class keg_auth.libs.authenticators.**KegAuthenticator**(*app*)
Uses username/password authentication with a login form, validates against keg-auth db

is_domain_excluded(*login_id*)
Domains configured for OAuth access are excluded from the password authenticator.

Any operations not using `verify_user` should check for exclusion.

class keg_auth.libs.authenticators.**LdapAuthenticator**(*app*)
Uses username/password authentication with a login form, validates against LDAP host

Most responder types won't be relevant here.

verify_password(*user*, *password*)
Check the given username/password combination at the application's configured LDAP server. Returns *True* if the user authentication is successful, *False* otherwise. NOTE: By request, authentication can be bypassed by setting the KEGAUTH_LDAP_TEST_MODE configuration setting to *True*. When set, all authentication attempts will succeed!

class keg_auth.libs.authenticators.**LoginAuthenticator**(*app*)
Manages verification of users as well as relevant view-layer logic

Relevant auth views (login, verification, resets, etc.) get passed through to responders on this layer, to process and render for the specific type of authentication happening.

For example, a password authenticator will want a user/password login form, but other types like oauth may get a different form entirely (and handle resets differently, etc.).

responder_cls is a key/value store for associating view keys with responder classes. If a view key is not present, we assume that view is not relevant to the authenticator, and the view itself will return 404.

class keg_auth.libs.authenticators.**LoginResponderMixin**
Wrap user authentication view-layer logic

Flash messages, what to do when a user has been authenticated (by whatever method the parent authenticator uses), redirects to a safe URL after login, etc.

static is_safe_url(*target*)
Returns *True* if the target is a valid URL for redirect

class keg_auth.libs.authenticators.**LogoutViewResponder**(*parent*)

class keg_auth.libs.authenticators.**OAuthAuthenticator**(*app*)
Uses OAuth authentication via authlib, validates user info against keg-auth db

class keg_auth.libs.authenticators.**OAuthAuthorizeViewResponder**(*parent*)
OAuth logins, using a provider via authlib

class keg_auth.libs.authenticators.**OAuthLoginViewResponder**(*parent*)
OAuth logins, using a provider via authlib

class keg_auth.libs.authenticators.**PasswordAuthenticatorMixin**
Username/password authenticators will need a way to verify a user is valid prior to making it the current user in flask login

class keg_auth.libs.authenticators.**PasswordCharset**(*name*, *alphabet*)

alphabet
Alias for field number 1

name
Alias for field number 0

class keg_auth.libs.authenticators.**PasswordFormViewResponder**(*parent*)
Master responder for username/password-style logins, using a login form

get_last_limiting_attempt(*username*)
Get the last attempt that counts toward the limit count. Attempts that count toward the limit before this attempt will be counted to determine if this attempt caused a lockout.

For login, this will be the last failed attempt. For password reset, this will be the last attempt.

get_limiting_attempt_count(*before_time*, *username*)
Return the number of attempts that count toward the limit up to before_time.

class keg_auth.libs.authenticators.**PasswordPolicy**
A base class that defines password requirements for the application. This class defines some basic, common validations and can be extended or limited by subclassing.

To define additional password checks, create a method on your subclass that accepts a password string and a user entity object and raises PasswordPolicyError if the password does not meet the requirement you intend to check. Then override password_checks to add your method to the returned list of methods.

To remove a password check that is enabled by default, override password_checks and return only the methods you wish to use.

Default settings are based on NIST guidelines and some common restrictions.

check_character_set(*pw: str*, *user*)
Raises PasswordPolicyError if a password does not contain at least one character from at least *required_at_least_char_types* of the alphabets in *required_char_sets*. :param pw: password to check :param user: user entity

check_does_not_contain_username(*pw: str*, *user*)
Raises PasswordPolicyError if the password contains the username. This is case insensitive. :param pw: password to check :param user: user entity

check_length(*pw: str*, *user*)
Raises PasswordPolicyError if a password is not at least min_length characters long. :param pw: password to check :param user: user entity

min_length = 8
Character sets used for checking minimum "complexity" in check_character_set validation

required_char_types = [PasswordCharset(name='lowercase letter', alphabet='abcdefghijkl
Minimum character number of different character types required in check_character_set validation

exception keg_auth.libs.authenticators.**PasswordPolicyError**

class keg_auth.libs.authenticators.**PasswordSetterResponderBase**(*parent*)
Base logic for resetting passwords and verifying accounts via token

form_cls
alias of *keg_auth.forms.SetPassword*

class keg_auth.libs.authenticators.**RedirectAuthenticator**(*app*)
Redirects to another source for authentication. Useful for when we have an OAuth source in mind for primary auth. We will want to redirect /login there, keep /logout, and direct other responder keys to return 404.

Use KEGAUTH_REDIRECT_LOGIN_TARGET to set the login target.

class keg_auth.libs.authenticators.**RedirectLoginViewResponder**(*parent*)

class keg_auth.libs.authenticators.**RequestLoader**(*app*)
Generic loader interface for determining if a user should be logged in

class keg_auth.libs.authenticators.**ResetPasswordViewResponder**(*parent*)
Responder for resetting passwords via token on keg-auth logins

get_last_limiting_attempt(*username*)
Get the last attempt that counts toward the limit count. Attempts that count toward the limit before this attempt will be counted to determine if this attempt caused a lockout.

For login, this will be the last failed attempt. For password reset, this will be the last attempt.

get_limiting_attempt_count(*before_time*, *username*)
Return the number of attempts that count toward the limit up to before_time.

class keg_auth.libs.authenticators.**TokenLoaderMixin**
Token authenticators will need a way to generate an access token, which will then be loaded in the request to log a user into flask-login

class keg_auth.libs.authenticators.**TokenRequestLoader**(*app*)

exception keg_auth.libs.authenticators.**UserInactive**(*user*)

exception keg_auth.libs.authenticators.**UserInvalidAuth**(*user*)

exception keg_auth.libs.authenticators.**UserNotFound**

class keg_auth.libs.authenticators.**VerifyAccountViewResponder**(*parent*)
Responder for verifying users via email token for keg-auth logins

class keg_auth.libs.authenticators.**ViewResponder**(*parent*)
View-layer logic wrapper for use in the Authenticator

Responder should be combined with needed mixins for various functionality (forms, logins, etc.).

Expected to have methods named for the request method (get, post, etc.)

template_name is passed to *flask.render_template* by default

handle_csrf()
For some views that are rate-limited, we want to log all attempts, including those that would fail CSRF validation. Because of this, we need to circumvent flask-csrf's default before-request hook. The auth manager will work to exempt any of our auth endpoints whose class is marked with _csrf_custom_handling.

If CSRF fails, ensure the attempt is logged, and then raise the error.

If CSRF succeeds, the ensuing view responder methods should do any appropriate logging.

7.2 Decorators

class keg_auth.libs.decorators.**RequiresPermissions**(*condition*, *on_authentication_failure=None*, *on_authorization_failure=None*, *http_methods_excluded=None*, *request_loaders=None*)
Require a user to be conditionally authorized before proceeding to decorated target. May be used as a class decorator or method decorator.

Usage: @requires_permissions(condition)

Note: if using along with a route decorator (e.g. Blueprint.route), requires_permissions should be the closest decorator to the method

Examples: - @requires_permissions((‘token1’, ‘token2’)) - @requires_permissions(has_any(‘token1’, ‘token2’)) - @requires_permissions(has_all(‘token1’, ‘token2’)) - @requires_permissions(has_all(has_any(‘token1’, ‘token2’), ‘token3’)) - @requires_permissions(custom_authorization_callable that takes user arg) - @requires_permissions(‘token1’, on_authorization_failure=lambda: flask.abort(404))

class keg_auth.libs.decorators.**RequiresUser**(*on_authentication_failure=None*, *http_methods_excluded=None*, *request_loaders=None*)
Require a user to be authenticated before proceeding to decorated target. May be used as a class decorator or

method decorator.

Usage: @requires_user

Note: if using along with a route decorator (e.g. Blueprint.route), requires_user should be the closest decorator to the method

Examples: - @requires_user - @requires_user() - @requires_user(on_authentication_failure=lambda: flask.abort(400)) - @requires_user(http_methods_excluded=['OPTIONS']) - @requires_user(request_loaders=[JwtRequestLoader])

keg_auth.libs.decorators.**requires_permissions**
alias of *keg_auth.libs.decorators.RequiresPermissions*

keg_auth.libs.decorators.**requires_user**(*arg=None*, **args*, ***kwargs*)
Require a user to be authenticated before proceeding to decorated target. May be used as a class decorator or method decorator.

Usage: @requires_user OR @requires_user() (both usage forms are identical)

Parameters:

on_authentication_failure: method called on authentication failures. If one is not specified, behavior is derived from login manager (redirect or 401)

on_authorization_failure: method called on authorization failures. If one is not specified, response will be 403

7.3 Navigation

class keg_auth.libs.navigation.**NavItem**(**args*, *nav_group=None*, *icon_class=None*, *class_=None*, *code=None*)
Defines a menu item or structure of a menu.

Example:

```
my_menu = NavItem(
    NavItem(
        'Admin',
        NavItem('Users', NavURL('auth.user:list')),
        NavItem('Groups', NavURL('auth.group:list')),
        nav_group='admin',
        icon_class='fas fa-briefcase',
        class_='my-menu-group'
    ),
    NavItem(
        'Reports',
        NavItem('Frequency', NavURL('frequency-report'), code='frequency'),
        NavItem('Financial', NavURL('money-report', requires_permissions='secret-
↪perm'))
    )
)
```

clear_authorization(*session_key*)
Reset cached authorization in this and all subnodes for the given session key.

has_current_route
Returns true if current request matches this nav node.

is_permitted
Compute/cache authorization from permission conditions, and return bool.

node_type
Return type NavItemType indicating whether this node is at the end of the structure.

permitted_sub_nodes
Return list of subnodes accessible to current user.

class keg_auth.libs.navigation.**NavURL**(*route_string*, **args*, ***kwargs*)
Wraps url_for with permission-checking to determine if user should see a route.

Endpoint is checked for user/permission requirements. - method/class/blueprint permissions from decorators (preferred in most cases) - *requires_permissions* kwarg specifies conditions and disregards the decorators - *requires_anonymous* kwarg forces nav for only unauthenticated users

Note that permission requirements are checked at all levels of the view hierarchy as needed: method, class, and blueprint.

is_permitted
Check permitted status of this route for the current user

keg_auth.libs.__init__.**get_current_user**()
Helper to grab the authenticated user from the session.

Trivial case is when the user is loaded in flask-login. If not, run the registered request loaders until we find a user.

keg_auth.libs.__init__.**get_domain_from_email**(*email*)
Extract domain portion of email address.

CHAPTER 8

Forms

class keg_auth.forms.**ForgotPassword**(**args*, ***kwargs*)
Returns a form to capture email for password reset.

class keg_auth.forms.**SetPassword**(**args*, ***kwargs*)
Returns a form to capture password/confirmation and apply password policy.

keg_auth.forms.**bundle_form**(*endpoint*)
Returns a form for Bundle CRUD.

keg_auth.forms.**group_form**(*endpoint*)
Returns a form for Group CRUD.

keg_auth.forms.**login_form**()
Returns a Login form class that handles username options.

keg_auth.forms.**user_form**(*config=None, allow_superuser=False, endpoint='', fields=['is_enabled', 'disabled_utc']*)
Returns a form for User CRUD.

Form is customized depending on the fields and superuser setting passed in.

CHAPTER 9

Grids

class keg_auth.grids.**ActionColumn**(*label*, *key=None*, *filter=None*, *can_sort=False*, *render_in=('html',)*, *has_subtotal=False*, *edit_endpoint=None*, *delete_endpoint=None*, *view_endpoint=None*, *edit_permission_for=<function ActionColumn.<lambda>>*, *delete_permission_for=<function ActionColumn.<lambda>>*, *view_permission_for=<function ActionColumn.<lambda>>*, *view_link_class_for=None*, *edit_link_class_for=None*, *delete_link_class_for=None*, ***kwargs*)

Places various action buttons in a Column.

Parameters

- **edit_permission_for** – is a function that takes a row and returns the permission required to open the edit endpoint for that row.
- **delete_permission_for** – is like *edit_permission_for*, but for the delete endpoint.
- **view_permission_for** – is like *edit_permission_for*, but for the view endpoint.
- **view_link_class_for** – is a function that takes a row and returns the HTML class to place on the view link.
- **edit_link_class_for** – is a function that takes a row and returns the HTML class to place on the edit link.
- **delete_link_class_for** – is a function that takes a row and returns the HTML class to place on the delete link.

extract_and_format_data(*record*)

Extract a value from the record for this column and run it through the data formaters.

format_data(*value*, *show_edit*, *show_delete*, *show_view*, *view_link_class*, *edit_link_class*, *delete_link_class*)

Use to adjust the value extracted from the record for this column. By default, no change is made. Useful in sub-classes.

keg_auth.grids.**make_bundle_grid**(*edit_endpoint*, *edit_permission*, *delete_endpoint*, *delete_permission*, *grid_cls=None*)
Factory method to create a Bundle grid class for CRUD.

keg_auth.grids.**make_group_grid**(*edit_endpoint*, *edit_permission*, *delete_endpoint*, *delete_permission*, *grid_cls=None*)
Factory method to create a Group grid class for CRUD.

keg_auth.grids.**make_permission_grid**(*grid_cls=None*)
Factory method to create a Permission grid class.

keg_auth.grids.**make_user_grid**(*edit_endpoint*, *edit_permission*, *delete_endpoint*, *delete_permission*, *grid_cls=None*, *resend_verification_endpoint=None*)
Factory method to create a User grid class for CRUD.

CHAPTER 10

Mail

class keg_auth.mail.**AuthMailManager**(*mail_ext*)

Manager to handle sending auth-related mail via a flask-mail extension.

Pass the mail extension in the constructor.

new_user_message(*user*)

Creates message from *new_user_templates* with the given user.

reset_password_message(*user*)

Creates message from *reset_password_template* with the given user.

reset_password_url(*user*)

Returns URL to use in mail for password reset for the given user.

send_new_user(*user*)

Send account creation email from *new_user_message* with the given user.

send_reset_password(*user*)

Send password reset email from *reset_password_message* with the given user.

verify_account_url(*user*)

Returns URL to use in mail for account verification for the given user.

class keg_auth.mail.**MailParts**(*subject*, *text*, *html*)

html

Alias for field number 2

subject

Alias for field number 0

text

Alias for field number 1

Python Module Index

k

Index

F

G

H

I

R

S

T

U

V

W