Watson - Auth

Release 4.0.0

Aug 09, 2017

Contents

Authorization and authentication library for Watson.

CHAPTER 1

Build Status

CHAPTER 2

Requirements

- watson-db
- bcrypt
- pyjwt

CHAPTER 3

Installation

```
pip install watson-auth
```

CHAPTER 4

Testing

Watson can be tested with py.test. Simply activate your virtualenv and run `python setup.py test`.

CHAPTER 5

Contributing

If you would like to contribute to Watson, please feel free to issue a pull request via Github with the associated tests for your code. Your name will be added to the AUTHORS file under contributors.

CHAPTER 6

Table of Contents

Usage

A few things need to be configured from within the IocContainer of your application before beginning.

1. An INIT application event listener must be added to your applications events. This injects some default configuration into your application and creates a new dependency in the IocContainer.

```
'events': {
    events.INIT: [
        ('watson.auth.listeners.Init', 1, True)
    ],
}
```

2. Configure the user model that you're going to use. Make sure that it subclasses `watson.auth.models.UserMixin`.

```
'auth': {
    'common': {
        'model': {
            'class': 'app.models.User'
        },
    }
}
```

3. Configure the routes and email address you're going to use for forgotten/reset password.

```
'auth': {
    'common': {
        'system_email_from_address': 'you@site.com',
        'reset_password_route': 'auth/reset-password',
        'forgotten_password_route': 'auth/forgotten-password'
    }
}
```

The default configuration (below) can be overridden in your applications config if required.

```
'auth': {
    'common': {
        'model': {
            'identifier': 'username',
            'email_address': 'username'
        },
        'session': 'default',
        'key': 'watson.user',
        'encoding': 'utf-8',
        'password': {
            'max_length': 30
        },
    },
}
```

Note that any of the url's above can also be named routes.

If you'd like to include authentication information in the toolbar at the bottom of the page, add the `watson.auth.panels.User` panel to the debug configuration.

::

debug = { 'enabled': True, 'toolbar': True, 'panels': {

'watson.auth.panels.User': { 'enabled': True

}

}

}

Providers

As of watson.auth 5.0.0 there is now the concept of 'Auth Providers'. These allow you to authenticate users via number of means. Currently watson.auth provides session (`watson.auth.providers.Session`) and JWT (`watson.auth.providers.JWT`) authentication, with OAuth2 support coming shortly.

Depending on what you're application requirements are, you might want to use a different provider to the default provider that is used. In order to that, modify your auth configuration.

```
'auth': {
    'providers': {
        'watson.auth.providers.ProviderName': {
            'secret': 'APP_SECRET',
        },
    },
}
```

Each provider may require individual configuration settings, and you'll see a nice big error page if you try to access your site without configuring these first.

Authentication

Setting up authentication will differ slightly depending on the provider you've chosen, but only in the decorators that you are using. You still need to configure 2 things:

1. Routes
2. Controllers

We'll assume that for this example we're just going to use the Session provider.

Start by creating the routes that you're going to need:

```
'routes': {
    auth': {
        'children': {
            'login': {
                'path': '/login',
                'options': {'controller': 'app.auth.controllers.Auth'},
                'defaults': {'action': 'login'}
            },
            'logout': {
                'path': '/logout',
                'options': {'controller': 'app.auth.controllers.Auth'},
                'defaults': {'action': 'logout'}
            },
        }
    }
}
```

Now create the controllers that handle these requests:

```
from watson.auth.providers.session.decorators import login, logout
from watson.framework import controllers

class Auth(controllers.Action):
    @login(redirect='/')
    def login_action(self, form):
        return {'form': form}

    @logout(redirect='/')
    def logout_action(self):
        pass
```

You'll notice that there is a `form` argument which is not included in your route definition. This is because the decorators will automatically pass through the form that is being used to validate the user input.

If you'd like to override the views (which is highly suggested), you can put your own views in `views/auth/<action>.html`.

Anytime a user visits the **/auth/login**, if the request is a POST (this can be overridden if required) then the user with be authenticated. If they visit **/auth/logout** they they will be logged out and redirected to `redirect`. If `redirect` is omitted, then the logout view will be rendered.

Once the user has been autheticated, you can retrieve the user within the controller by using `self.request.user`.

Authorization

watson-auth provides a strongly customizable authorization system. It allows you to configure both roles, and permissions for users. The management of these however is not controlled by watson-auth, so it will be up to you to create the necessary UI to create/delete/update roles.

Please note that some of these actions can also be done via the command *./console.py auth*.

Defining the roles and permissions

First, define some roles for the system and add them to the session via the watson cli (from your application root).

```
./console.py auth add_role [key] [name]
./console.py auth add_permission [key] [name]
# where [key] is the identifier within the application and [name] is the human␣
↪readable name
```

Creating a new user

watson-auth provides a base user mixin that has some common fields, and should be subclassed. watson.auth.models.Model will be the declarative base of whatever session you have configured in config['auth']['model']['session'].

```
from watson.auth import models
from watson.form import fields

class User(models.UserMixin, models.Model):
    __tablename__ = 'users'
    username = Column(String(255), unique=True)
```

Next, create the user and give them some roles and permissions via the watson cli (from your application root).

:: ./console.py auth create_user [username] [password] ./console.py auth add_role_to_user [username] [key] ./console.py auth add_permission_to_user [username] [key] [value]

If no permissions are specified, then the user will receive inherited permissions from that role. Permissions can be given either allow (1) or deny (0).

Authorizing your controllers

Like authentication, authorizing your controllers is done via decorators.

```
from watson.auth.providers.session.decorators import auth
from watson.framework import controllers

class Public(controllers.Action):

    @auth
    def protected_action(self):
        # some sensitive page
```

`@auth` accepts different arguments, but the common ones are:

- roles: A string or tuple containing the roles the user must have
- permissions: A string or tuple containing the permissions the user must have
- requires: A list of `watson.validators.abc.Valiator` objects that are used to validate the user.

Check out the `watson.auth.providers.PROVIDER.decorators` module for more information.

Accessing the user

At any time within your controller you can access the user that's currently authenticated through the request.

```
class MyController(controllers.Action):
    def index_action(self):
        user = self.request.user
```

Resetting a password

As of v3.0.0, the user can now reset their password via the forgotten password functionality.

Several options are also configurable such as automatically logging the user in once they have successfully reset their password. See the configuration settings above for more information.

Create the routes you wish to use:

```
'routes': {
    auth': {
        'children': {
            'reset-password': {
                'path': '/reset-password',
                'options': {'controller': 'app.auth.controllers.Auth'},
                'defaults': {'action': 'reset_password'}
            },
            'forgotten-password': {
                'path': '/forgotten-password',
                'options': {'controller': 'app.auth.controllers.Auth'},
                'defaults': {'action': 'forgotten_password'}
            }
        }
    }
}
```

And then create the controllers that will handle these routes:

```
from watson.auth.providers.session.decorators import forgotten, reset
from watson.framework import controllers

class Auth(controllers.Action):
    @forgotten
    def forgotten_password_action(self, form):
        return {'form': form}

    @reset
    def reset_password_action(self, form):
        return {'form': form}
```

The user will be emailed a link to be able to reset their password. This template uses whatever renderer is the default set in your project configuration, and can therefore be overridden by creating a new template file in your views directory (*auth/emails/forgotten-password.html* and *auth/emails/reset-password.html*).

Reference Library

watson.auth.authorization

class watson.auth.authorization.**Acl**(*user*)

Access Control List functionality for managing users' roles and permissions.

By default, the user model contains an *acl* attribute, which allows access to the Acl object.

allow_default
boolean – Whether or not to allow/deny access if the permission has not been set on that role.

__init__(*user*)
Initializes the Acl.

Parameters **user** (*watson.auth.models.UserMixin*) – The user to validate against

_generate_user_permissions()
Internal method to generate the permissions for the user.

Retrieve all the permissions associated with the users roles, and then merge the users individual permissions to overwrite the inherited role permissions.

has_permission(*permission*)
Check to see if a user has a specific permission.

If the permission has not been set, then it access will be granted based on the allow_default attribute.

Parameters **permission** (*string*) – The permission to find.

has_role(*role_key*)
Validates a role against the associated roles on a user.

Parameters **role_key** (*string|tuple|list*) – The role(s) to validate against.

watson.auth.commands

watson.auth.config

watson.auth.crypto

watson.auth.forms

class watson.auth.forms.**ForgottenPassword**(*name=None*, *method='post'*, *action=None*, *detect_multipart=True*, *validators=None*, *values_provider=None*, ***kwargs*)
A standard forgotten password form.

class watson.auth.forms.**Login**(*name=None*, *method='post'*, *action=None*, *detect_multipart=True*, *validators=None*, *values_provider=None*, ***kwargs*)
A standard login form containing username and password fields.

class watson.auth.forms.**ResetPassword**(*name=None*, *method='post'*, *action=None*, *detect_multipart=True*, *validators=None*, *values_provider=None*, ***kwargs*)
A standard reset password form.

watson.auth.listeners

class watson.auth.listeners.**Init**
Bootstraps watson.auth into the event system of watson.

class watson.auth.listeners.**Route**
Listens for a route event and attempts to inject the user into the request if one has been authenticated.

watson.auth.models

Sphinx cannot automatically generate these docs. The source has been included instead:

```
# -*- coding: utf-8 -*-
from datetime import datetime
from sqlalchemy import (Column, Integer, String, DateTime, ForeignKey,
                        SmallInteger)
from sqlalchemy.ext.declarative import declared_attr
from sqlalchemy.orm import relationship
from watson.common import imports
from watson.auth import authorization, crypto
from watson.db.models import Model
from watson.db.utils import _table_attr


class Permission(Model):
    id = Column(Integer, primary_key=True)
    name = Column(String(255))
    key = Column(String(255))
    created_date = Column(DateTime, default=datetime.now)

    def __repr__(self):
        return '<{0} key:{1} name:{2}>'.format(
            imports.get_qualified_name(self), self.key, self.name)


class Role(Model):
    id = Column(Integer, primary_key=True)
    name = Column(String(255))
    key = Column(String(255))
    permissions = relationship('RolesHasPermission',
                               backref='roles')
    created_date = Column(DateTime, default=datetime.now)

    def add_permission(self, permission, value=1):
        """Adds a permission to the role.

        Args:
            Permission permission: The permission to attach
            int value: The value to give the permission, can be either:
                       0 - deny
                       1 - allow
        """
        role_permission = RolesHasPermission(value=value)
        role_permission.permission = permission
        self.permissions.append(role_permission)

    def __repr__(self):
        return '<{0} key:{1} name:{2}>'.format(
            imports.get_qualified_name(self), self.key, self.name)


class UserMixin(object):

    """Common user fields, custom user classes should extend this as well as
    Model.

```

```
    Attributes:
        string id_field: The name of the field to use as the id for the user

    Columns:
        string _password: The password of the user, aliased by self.password
        string salt: The salt used to generate the password
        list roles: The roles associated with the user
        list permissions: The permissions associated with the user, overrides
                          the permissions associated with the role.
        date created_date: The time the user was created.
        date updated_date: The time the user was updated.
    """
    __tablename__ = 'users'
    _acl_class = authorization.Acl
    _acl = None
    id = Column(Integer, primary_key=True)
    _password = Column(String(255), name='password')
    salt = Column(String(255), nullable=False)
    created_date = Column(DateTime, default=datetime.now)
    updated_date = Column(DateTime, default=datetime.now)

    @property
    def acl(self):
        """Convenience method to access the users ACL.

        See watson.auth.authorization.Acl for more information.
        """
        if not self._acl:
            self._acl = self._acl_class(self)
        return self._acl

    @declared_attr
    def permissions(cls):
        return relationship(UsersHasPermission, backref='user', cascade='all')

    @declared_attr
    def roles(cls):
        return relationship(Role,
                            secondary=UsersHasRole.__tablename__,
                            backref='roles', cascade=None)

    @declared_attr
    def forgotten_password_tokens(cls):
        return relationship(ForgottenPasswordToken, backref='user', cascade='all')

    @property
    def password(self):
        """Return the password.
        """
        return self._password

    @password.setter
    def password(self, password):
        """Automatically generates the hashed password and salt when set.

        Args:
            string password: The password to set.
        """
```

```
        _pass, salt = crypto.generate_password(password)
        self._password = _pass
        self.salt = salt

    def touch(self):
        """Updates the date the user was modified.
        """
        self.updated_date = datetime.now()

    def add_permission(self, permission, value=1):
        """Adds a permission to the user.

        This overrides any permission given by the associated roles.

        Args:
            Permission permission: The permission to attach
            int value: The value to give the permission, can be either:
                        0 - deny
                        1 - allow
        """
        user_permission = UsersHasPermission(value=value)
        user_permission.permission = permission
        self.permissions.append(user_permission)

    def __repr__(self):
        return '<{0} id:{1}>'.format(imports.get_qualified_name(self), self.id)


class RolesHasPermission(Model):
    role_id = Column(Integer,
                     ForeignKey(_table_attr(Role, 'id')),
                     primary_key=True)
    permission_id = Column(Integer,
                           ForeignKey(_table_attr(Permission, 'id')),
                           primary_key=True)
    permission = relationship(Permission)
    value = Column(SmallInteger, default=0)
    created_date = Column(DateTime, default=datetime.now)


class UsersHasPermission(Model):
    user_id = Column(Integer,
                     ForeignKey(_table_attr(UserMixin, 'id')),
                     primary_key=True)
    permission_id = Column(Integer,
                           ForeignKey(_table_attr(Permission, 'id')),
                           primary_key=True)
    permission = relationship(Permission)
    value = Column(SmallInteger, default=0)
    created_date = Column(DateTime, default=datetime.now)


class UsersHasRole(Model):
    user_id = Column(Integer,
                     ForeignKey(_table_attr(UserMixin, 'id')),
                     primary_key=True)
    role_id = Column(Integer,
                     ForeignKey(_table_attr(Role, 'id')),
```

```
                    primary_key=True)


class ForgottenPasswordToken(Model):
    id = Column(Integer, primary_key=True)
    token = Column(String(255))
    user_id = Column(Integer,
                     ForeignKey(_table_attr(UserMixin, 'id')))
    created_date = Column(DateTime, default=datetime.now)

    def __repr__(self):
        return '<{0} user id:{1}>'.format(
            imports.get_qualified_name(self), self.user.id)
```

watson.auth.panels

watson.auth.providers

watson.auth.providers.**JWT**
alias of Provider

watson.auth.providers.**Session**
alias of Provider

watson.auth.validators

watson.auth.**validators**
alias of *watson.auth.validators*

Python Module Index

W

Index

Symbols

A

F

H

I

J

L

R

S

V

W