## Contents

1 User’s Guide .......................... 3
   1.1 Overview ........................................ 3
   1.2 Installation ..................................... 4
   1.3 Configuration ................................. 5
   1.4 Usage ........................................... 5

2 API Reference .......................... 13
   2.1 API Docs ....................................... 13

3 Additional Notes ...................... 19
   3.1 Contributing ................................. 19
   3.2 Changes ....................................... 21
   3.3 License ....................................... 21
   3.4 Contributors ................................. 22

Python Module Index .................... 23

Index ......................................... 25
Role-based access control (RBAC) for Invenio.

Invenio-Access works together with Invenio-Accounts to provide a full-fledge authentication and authorization system for Flask and Invenio based on a suite of existing Flask extensions such as:

- Flask-Security
- Flask-Login
- Flask-Principal
- passlib

Features:

- Role-based access control with object level permissions.
- CLI and administration interface for allowing/denying actions to users, roles or system roles.
- Support for superuser privileges.

Further documentation is available on https://invenio-access.readthedocs.io/
This part of the documentation will show you how to get started in using Invenio-Access.

### 1.1 Overview

The following is a walk-through of the important concepts in the access control system.

#### 1.1.1 Users & roles

First we have **subjects** which can be granted access to a protected resource.

- **User**: Represents an authenticated account in the system.
- **Role**: Represents a job function. Roles are created by e.g. system administrators and defined by a name. Users can be assigned zero or more roles.
- **System role**: Represents special roles that are created and defined by the system and automatically assigned to users (i.e. system roles cannot be created and defined by system administrators).

#### 1.1.2 Permissions and needs

Second, we have two entities to describe access control:

- **Need**: A need represents the smallest level of access control. It is very generic and can express statements such as **“has admin role”** and **“has read access to record 42”**.
- **Permission**: Represents a set of required needs, any of which should be fulfilled to access a resouce. E.g. a permission can combine the two statements above into **“has admin role or has read access to record 42”**.

The concept of a need can be somewhat hard to grasp at first, so let’s dive in to describe how a need is used. Essentially needs are used to express a) what a permission require and b) what a user provides, i.e.:

- A permission requires a set of needs.
• A user provides a set of needs.

Thus, checking if a user can access a resource protected by a permission amounts to checking for a non-empty intersection between the above sets.

1.1.3 Types of needs

Needs can have different types. For instance the statement “has admin role” can be expressed as a role need type with the argument admin. This means that a permission can require the admin role need and that a user can provide the admin role need. Some basic need types include:

1.1.4 Actions

Action need are a special type of need that represents actions (surprise!). Action needs can have zero or more parameters. For instance the statement “has read access to record 42” can be decomposed into read record action need with the parameter 42.

Action needs has the advantage that they do not tie a permission to a specific role/user name and are much easier to compose and re-use.

1.1.5 Protecting resources

In order to protect a resource, you will usually create a new permission which will require one or more action needs. This new permission and the action needs are usually expressed explicitly in the source code. In particular note that Invenio usually always protects resources via action needs instead of user and role needs.

1.1.6 Granting access

Subjects (users, roles and system roles) are assigned actions. E.g. a user or a role can be assigned the action “read record”. If the action has parameters, then it can be assigned to the subject for any parameters or for a specific parameter (e.g. read any record vs read record 42).

1.1.7 Identity

The last entity to cover is an identity. During request handling any user (authenticated or unauthenticated) is represented as an identity. The identity is used to express the set of needs that the current user provides. It is solely an abstraction layer on top of users and roles such that we do not have to care if actions are assigned directly to a user or indirectly to a user via a role.

1.2 Installation

Invenio-Access is on PyPI so all you need is:

$ pip install invenio-access

Invenio-Access depends on Invenio-DB and Invenio-Accounts.
1.3 Configuration

Default values for access configuration.

Note: By default no caching is enabled. For production instances it is highly advisable to enable caching as the permission checking is very query intensive on the database.

```
invenio_access.config.ACCESS_ACTION_CACHE_PREFIX = 'Permission::action::'
    Prefix for actions cached when used in dynamic permissions.

invenio_access.config.ACCESS_CACHE = None
    A cache instance or an importable string pointing to the cache instance.

invenio_access.config.ACCESS_LOAD_SYSTEM_ROLE_NEEDS = True
    Enables the loading of system role needs when users’ identity change.
```

1.4 Usage

Role-based access control for Invenio.

Invenio-Access works together with Invenio-Accounts to provide a full-fledged authentication and authorization system for Flask and Invenio based on a suite of existing Flask extensions such as:

- Flask-Security
- Flask-Login
- Flask-Principal
- passlib

Make sure you check out Overview to have a basic understanding of the entities in the access control system. This part of the usage documentation is focused on the programmatic APIs and are intended for developers.

1.4.1 Initialization

Create a Flask application:

```
>>> import os
>>> db_url = os.environ.get('SQLALCHEMY_DATABASE_URI', 'sqlite://')
>>> from flask import Flask
>>> app = Flask('myapp')
>>> app.config.update({
...     'SQLALCHEMY_DATABASE_URI': db_url,
...     'SQLALCHEMY_TRACK_MODIFICATIONS': False,
... })
```

Initialize Invenio-Access dependencies, which are Invenio-DB and Invenio-Accounts, and then Invenio-Access itself:

```
>>> from invenio_db import InvenioDB
>>> from invenio_accounts import InvenioAccounts
>>> from invenio_access import InvenioAccess
>>> ext_db = InvenioDB(app)
>>> ext_accounts = InvenioAccounts(app)
>>> ext_access = InvenioAccess(app)
```
The following examples needs to run in a Flask application context, so let’s push one:

```python
code
>>> app.app_context().push()
```

Also, for the examples to work we need to create the database and tables (note, in this example we use an in-memory SQLite database by default):

```python
code
>>> from invenio_db import db
>>> db.create_all()
```

## Demo data

Let’s also create two initial users and a role:

```python
code
>>> from invenio_accounts.models import User, Role
>>> alice = User(email='alice@inveniosoftware.org')
>>> bob = User(email='bob@inveniosoftware.org')
>>> admin = Role(name='admin')
```

Now, assign Alice to the admin role:

```python
code
>>> admin.users.append(alice)
```

Last, persist the changes to the database:

```python
code
>>> db.session.add_all([alice, bob, admin])
>>> db.session.commit()
```

### 1.4.2 Protecting resources

The basics of protecting a resource involves:

1. Define an action.
2. Create a permission that requires one or more actions.
3. Check if a permission allows a given identity (i.e. the identity provides one or more of the required actions).

#### 1. Define an action

First, let’s start with defining an action (e.g. view an index page in our module) using the action creation factory:

```python
code
>>> from invenio_access import action_factory
>>> view_index_action = action_factory('mymodule-index-view')
```

#### 2. Create a permission

Next, we create a permission that requires the just created action:

```python
code
>>> from invenio_access import Permission
>>> permission = Permission(view_index_action)
```

#### 3. Check permission

In order to check the permission we first need an identity, so let’s start out with an anonymous identity (this happens transparently in the background when a user login):
>>> from flask_principal import AnonymousIdentity
>>> anonymous = AnonymousIdentity()

Next, we can check if the permission allows the given identity (we will see in detail below how to use permissions in a view):

>>> permission.allows(anonymous)
False

1.4.3 Granting access

Checking if the anonymous identity is granted access by a permission is often not too interesting, so let’s grant our admin role access to our action:

>>> from invenio_access.models import ActionRoles, ActionUsers

>>> db.session.add(ActionRoles.allow(view_index_action, role=admin))
>>> db.session.commit()

Next, we need identity instances for our two users (normally you will not have to worry about this when checking permissions in a view as it is handled transparently by Flask-Security):

>>> from invenio_access.utils import get_identity

>>> alice_identity = get_identity(alice)
>>> bob_identity = get_identity(bob)

Now that we have the identities, we can check if the permission grants access to the identities:

>>> permission.allows(alice_identity)
True
>>> permission.allows(bob_identity)
False

Notice, that we granted access to Alice by assigning her the role admin and granting the role permission on the action. The Flask-Principal API is pretty rich, and there are multiple other ways that you can check if a permission grants access to an identity. For instance below is another example, but please explore the Flask-Principal API documentation for a full reference:

>>> bob_identity.can(permission)
False

1.4.4 Action parameters

Above we created an action that did not take any parameters. These actions are useful to grant/restrict access to e.g. an entire administration interface. However, in many cases you need object level permissions, in which case you need to use actions with parameters.

Action with parameters are also created with the action_factory(), but works a bit different as they take a parameter.

First you create the new action with parameter:

>>> ObjectReadAction = action_factory(
... 'mymodule-object-read', parameter=True)
Everytime you create the action, you also need to create an instance of the action representing any parameter as done like below:

```python
>>> object_read_action_all = ObjectReadAction(None)
```

**Granting access to actions with parameters**

You grant access to actions with parameters in a similar way as for normal actions, but you can now grant access to either all objects:

```python
>>> db.session.add(ActionRoles.allow(object_read_action_all, role=admin))
```

Or you can grant access to a specific object like this:

```python
>>> db.session.add(ActionUsers.allow(ObjectReadAction(42), user=bob))
>>> db.session.commit()
```

**Checking permissions for a specific object**

Similar you also create a permission that checks access to a specific object:

```python
>>> permission = Permission(ObjectReadAction(42))
>>> permission.allows(bob_identity)
True
>>> permission.allows(alice_identity)
True
```

### 1.4.5 Denying access

Besides granting access, you can also deny access to specific users or roles. Below for instance we deny access to Alice on the view_index_action.

```python
>>> from invenio_access.models import ActionUsers
>>> db.session.add(ActionUsers.deny(view_index_action, user=alice))
>>> db.session.commit()
```

When we now check the permission, Alice no longer has access:

```python
>>> permission = Permission(view_index_action)
>>> permission.allows(alice_identity)
False
```

**Deny takes precedence over allow**

Note, that the deny grant takes precedence over allow grant. Alice was earlier granted access to the action via her role assignment, however since the deny grant takes precedence Alice is ultimately denied access.

This is useful if you for instance want to grant access to all objects except one.

### 1.4.6 Protecting views

The most common use for permissions is to protect a view. For actions without parameters you can simply use a decorator for the view:
Permission factories

In most situations, you however have to deal with object level permissions, and thus you will have to create the
permission on-the-fly via a factory method. A simple permission factory can look like the one below:

```python
>>> def permission_factory(obj):
...     return Permission(ObjectReadAction(obj['id']))
```

The factory function simply takes your object and returns a permission for the specific action. This unfortunately also
means that you usually cannot use the decorator option shown above. Instead you usually have to first fetch your
object from e.g. the database, and then run the permission check:

```python
>>> @app.route('/objects/<int:object_id>')
... def object_view(object_id):
...     with permission_factory({'id': object_id}).require(http_exception=404):
...         return 'Protected index page'
```

Note: Invenio source code almost exclusively use the permission factory approach for protecting views. In addition
usually the permission factory is configurable so that Invenio instances can fully override the internal permission
handling.

Security considerations

We can now test the two views via the built-in Flask test client, and see that anonymous requests are denied in both
cases:

```python
>>> with app.test_client() as c:
...     c.get('/')
<Response streamed [403 FORBIDDEN]>
>>> with app.test_client() as c:
...     c.get('/objects/42')
<Response streamed [404 NOT FOUND]>
```

In the two above examples for protecting views, you will notice that in one we return an HTTP 403 Forbidden error,
and in the other we return a HTTP 404 Not Found error.

In views, you should always make a conscious decision if you should return 401/403 or 404 errors as it has important
security considerations.

- **403/401** errors exposes existence of an object under a given URL. Hence, by using 401/403 errors, the system
  is “leaking” knowledge that certain objects exists in the system. Hence, only use 401/403 errors when this
  behavior is desired. In all other cases use 404 errors.
- **404** errors does not leak any additional information and should be the default error used when a permission
  check fails.
1.4.7 Superuser

Invenio-Access provides a way to grant superuser privileges to users or roles via a superuser action. Granting superuser access to a user implicitly gives that user access to any action in the system without explicitly having to grant the action.

For instance, currently Bob does not have permissions on our view_index_action:

```python
>>> permission = Permission(view_index_action)
>>> permission.allows(bob_identity)
False
```

We can however grant Bob superuser access like this:

```python
>>> from invenio_access.permissions import superuser_access
>>> db.session.add(ActionUsers.allow(superuser_access, user=bob))
>>> db.session.commit()
```

Now, Bob will have access to the view_index_action even though we did not explicitly grant Bob access:

```python
>>> permission.allows(bob_identity)
True
```

1.4.8 System roles

Invenio-Access, in addition to roles defined by the administrator, provides also system roles. System roles are defined by the system and automatically assigned to users.

By default the following system roles exists:

- Any user (guests and authenticated users)
- Authenticated user

System roles works very much like normal roles, so you can e.g. assign actions to them:

```python
>>> from invenio_access import ActionSystemRoles, any_user
>>> db.session.add(ActionSystemRoles.allow(
...     view_index_action, role=any_user))
```

In order to test system roles from the shell, we have to manually add the need into the identity.

```python
>>> anonymous.provides.add(any_user)
```

Now we can check the permission:

```python
>>> permission = Permission(view_index_action)
>>> permission.allows(anonymous)
True
```

Creating system roles

Invenio modules may provide additional system roles. You could, for instance create a system role that could be used to grant permissions based on IP address.

First the module should define the system role:
>>> from invenio_access import SystemRoleNeed
>>> campus_user = SystemRoleNeed('campus_user')

Next, connect a receiver to the `identity_loaded` signal and add the system role need to the identity:

```python
>>> from flask import request
>>> from flask_principal import identity_loaded
>>> @identity_loaded.connect_via(app)
... def on_identity_loaded(sender, identity):
...     if request.remote_addr.startswith('192.168. '):
...         identity.provides.add(campus_user)
```

Last, you need to register the system role in the Invenio module’s entry points in `setup.py`:

```python
entry_points={
    'invenio_access.system_roles': [
        'campus_user' = mymodule.permissions:campus_user',
    ],
}
```

### 1.4.9 Registering actions

All actions that a package provides should be registered in the `invenio_access.actions` entry points. This ensures the actions are e.g. available in the administration interface and via the CLI.

Below is an example for of the entry point part of the `setup.py`:

```python
entry_points={
    'invenio_access.actions': [
        # Action with parameter
        'object_read_action_all' = mymodule.permissions:object_read_action_all',
        # Action without parameter
        'view_index_action' = mymodule.permissions:view_index_action',
    ],
}
```

Note that for action with parameters you need point to the import path of the action representing *any* parameters.

### 1.4.10 Listing actions

In order to discover which actions are available in a given installation, one can retrieve them via:

```python
>>> sorted(app.extensions['invenio-access'].actions.keys())
['admin-access', 'superuser-access']
```
If you are looking for information on a specific function, class or method, this part of the documentation is for you.

## 2.1 API Docs

Invenio module for common role based access control.

```python
class invenio_access.ext.InvenioAccess(app=None, **kwargs)
```

Invenio Access extension.

Extension initialization.

**Parameters**

- `app` – The Flask application. (Default: None)

`init_app(app,
entry_point_actions='invenio_access.actions',
entry_point_system_roles='invenio_access.system_roles', **kwargs)`

Flask application initialization.

**Parameters**

- `app` – The Flask application.

- `entry_point_actions` – The entriypoint for actions extensions. (Default: 'invenio_access.actions')

- `entry_point_system_roles` – The entriypoint for system roles extensions. (Default: 'invenio_access.system_roles')

- `cache` – The cache system. (Default: None)

`init_config(app)`

Initialize configuration.

**Parameters**

- `app` – The Flask application.
2.1.1 Action factory

Factory method for creating new action needs.

```python
invenio_access.factory.action_factory(name, parameter=False)
```

Factory method for creating new actions (w/wo parameters).

**Parameters**

- `name` – Name of the action (prefix with your module name).
- `parameter` – Determines if action should take parameters or not. Default is `False`.

2.1.2 Permissions

```python
class invenio_access.permissions.Permission(*needs)
```

Represents a set of required needs.

Extends Flask-Principal's `flask_principal.Permission` with support for loading action grants from the database including caching support.

Essentially the class works as a translation layer that expands action needs into a list of user/roles needs. For instance, take the following permission:

```python
Permission(ActionNeed('my-action'))
```

Once the permission is checked with an identity, the class will fetch a list of all users and roles that have been granted/denied access to the action, and expand the permission into something similar to (depending on the state of the database):

```python
Permission(UserNeed('1'), RoleNeed('admin'))
```

The expansion is cached until the action is modified (e.g. a user is granted access to the action). The alternative approach to expanding the action need like this class is doing, would be to load the list of allowed actions for a user on login and cache the result. However retrieving all allowed actions for a user could result in very large lists, where as caching allowed users/roles for an action would usually yield smaller lists (especially if roles are used).

Initialize permission.

**Parameters**

- `*needs` – The needs for this permission.

- `allow_by_default = False`
  
  If enabled, all permissions are granted when they are not assigned to anybody. Disabled by default.

- `excludes`
  
  Return denied permissions from database.

- `needs`
  
  Return allowed permissions from database.

2.1.3 Needs

```python
invenio_access.permissions.ParameterizedActionNeed = <functools.partial object>
```

A need having the method preset to “`action`” and a parameter.
If it is called with `argument=None` then this need is equivalent to `ActionNeed`.

```python
invenio_access.permissions.SystemRoleNeed = <functools.partial object>
```
A need with the method preset to "system_role".

### 2.1.4 System roles

```python
invenio_access.permissions.any_user = Need(method='system_role', value='any_user')
```
Any user system role.

This role is used to assign all possible users (authenticated and guests) to an action.

```python
invenio_access.permissions.authenticated_user = Need(method='system_role', value='authenticated_user')
```
Authenticated user system role.

This role is used to assign all authenticated users to an action.

```python
invenio_access.loaders.load_permissions_on_identity_loaded = <function load_permissions_on_identity_loaded>
```
Add system roles “Needs” to users’ identities.

Every user gets the `any_user` Need. Authenticated users get in addition the `authenticated_user` Need.

### 2.1.5 Actions

```python
invenio_access.permissions.superuser_access = Need(method='action', value='superuser-access')
```
Superuser access action which allow access to everything.

### 2.1.6 Models

Database models for access module.

```python
class invenio_access.models.ActionNeedMixin
```
Define common attributes for Action needs.

```python
action = Column(None, String(length=80), table=None)
```
Name of the action.

```python
classmethod allow(action, **kwargs)
```
Allow the given action need.

**Parameters**

- `action` – The action to allow.

**Returns**

A `invenio_access.models.ActionNeedMixin` instance.

```python
argument = Column(None, String(length=255), table=None)
```
Action argument.

```python
classmethod create(action, **kwargs)
```
Create new database row using the provided action need.

**Parameters**

- `action` – An object containing a method equal to 'action' and a value.
- `argument` – The action argument. If this parameter is not passed, then the `action`. `argument` will be used instead. If the `action`. `argument` does not exist, `None` will be set as argument for the new action need.

**Returns**

An `invenio_access.models.ActionNeedMixin` instance.
classmethod deny (action, **kwargs)
Deny the given action need.

Parameters
action – The action to deny.

Returns
A invenio_access.models.ActionNeedMixin instance.

exclude = Column (None, Boolean (name='exclude'), table=None, nullable=False, default=ColumnDefault (False), server_default=DefaultClause ('0', for_update=False))
If set to True, deny the action, otherwise allow it.

id = Column (None, Integer (), table=None, primary_key=True, nullable=False)
Primary key. It allows the other fields to be nullable.

need
Return the need corresponding to this model instance.

This is an abstract method and will raise NotImplementedError.

classmethod query_by_action (action, argument=None)
Prepare query object with filtered action.

Parameters
• action – The action to deny.

• argument – The action argument. If it’s None then, if exists, the action.argument will be taken. In the worst case will be set as None. (Default: None)

Returns
A query object.
class invenio_access.models.ActionRoles (**kwargs)
ActionRoles data model.

It relates an allowed action with a role.

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance’s class are allowed. These could be, for example, any mapped columns or relationships.

need
Return RoleNeed instance.
class invenio_access.models.ActionSystemRoles (**kwargs)
ActionSystemRoles data model.

It relates an allowed action with a predefined role. Example: “any user”

A simple constructor that allows initialization from kwargs.

Sets attributes on the constructed instance using the names and values in kwargs.

Only keys that are present as attributes of the instance’s class are allowed. These could be, for example, any mapped columns or relationships.

classmethod create (action, **kwargs)
Create new database row using the provided action need.

need
Return the corresponding Need instance.

validate_role_name (key, role_name)
Checks that the role name has been registered.
class invenio_access.models.ActionUsers(**kwargs)

    ActionUsers data model.
    It relates an allowed action with a user.
    A simple constructor that allows initialization from kwargs.
    Sets attributes on the constructed instance using the names and values in kwargs.
    Only keys that are present as attributes of the instance’s class are allowed. These could be, for example, any mapped columns or relationships.

    need
    Return UserNeed instance.

invenio_access.models.changed_action(mapper, connection, target)

    Remove the action from cache when an item is updated.

invenio_access.models.get_action_cache_key(name, argument)

    Get an action cache key string.

invenio_access.models.removed_or_inserted_action(mapper, connection, target)

    Remove the action from cache when an item is inserted or deleted.

2.1.7 Utils

Utility functions for Invenio-Access.

invenio_access.utils.get_identity(user)

    Create an identity for a given user instance.
    Primarily useful for testing.

2.1.8 Proxies

Helper proxy to the state object.

invenio_access.proxies.current_access = <LocalProxy unbound>

    Helper proxy to access state object.

2.1.9 CLI

Command line interface for Invenio-Access.

invenio_access.cli.lazy_result(f)

    Decorate function to return LazyProxy.

invenio_access.cli.process_action(ctx, param, value)

    Return an action if exists.

invenio_access.cli.process_allow_action(*args, **kwargs)

    Process allow action.

invenio_access.cli.process_deny_action(*args, **kwargs)

    Process deny action.

invenio_access.cli.process_email(ctx, param, value)

    Return an user if it exists.
invenio_access.cli.process_remove_action(*args, **kwargs)
    Process action removals.

invenio_access.cli.process_role(ctx, param, value)
    Return a role if it exists.

invenio_access.cli.allow_action = <Group allow>
    Allow action.

invenio_access.cli.deny_action = <Group deny>
    Deny actions.

invenio_access.cli.list_actions = <Command list>
    List all registered actions.
Notes on how to contribute, legal information and changes are here for the interested.

3.1 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

3.1.1 Types of Contributions

Report Bugs

If you are reporting a bug, please include:

• Your operating system name and version.
• Any details about your local setup that might be helpful in troubleshooting.
• Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with “feature” is open to whoever wants to implement it.
Write Documentation

Invenio Access could always use more documentation, whether as part of the official Invenio Access docs, in doc-
strings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/inveniosoftware/invenio-access/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

3.1.2 Get Started!

Ready to contribute? Here’s how to set up invenio-access for local development.

1. Fork the inveniosoftware/invenio-access repo on GitHub.
2. Clone your fork locally:

   ```
   $ git clone git@github.com:your_name_here/invenio-access.git
   ```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up
your fork for local development:

   ```
   $ mkvirtualenv invenio-access
   $ cd invenio-access/
   $ pip install -e .[all]
   ```

4. Create a branch for local development:

   ```
   $ git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

5. When you’re done making changes, check that your changes pass tests:

   ```
   $ ./run-tests.sh
   ```

   The tests will provide you with test coverage and also check PEP8 (code style), PEP257 (documentation), flake8
   as well as build the Sphinx documentation and run doctests.

6. Commit your changes and push your branch to GitHub:

   ```
   $ git add .
   $ git commit -s
   -m "component: title without verbs"
   -m "* NEW Adds your new feature."
   -m "* FIX Fixes an existing issue."
   -m "* BETTER Improves and existing feature."
   -m "* Changes something that should not be visible in release notes."
   $ git push origin name-of-your-bugfix-or-feature
   ```

7. Submit a pull request through the GitHub website.
3.1.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests and must not decrease test coverage.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring.
3. The pull request should work for Python 2.7, 3.3, 3.4 and 3.5. Check https://travis-ci.org/inveniosoftware/invenio-access/pull_requests and make sure that the tests pass for all supported Python versions.

3.2 Changes

Version 1.4.1 (released 2020-05-07)
• set Sphinx <3 because of errors related to application context
• stop using example app

Version 1.4.0 (released 2020-03-12)
• drop Python 2.7 support
• change Flask dependency management to centralised by invenio-base

Version 1.3.2 (released TBD)
• set Sphinx <3 because of errors related to application context
• stop using example app

Version 1.3.1 (released 2020-01-22)
• increase minimal six version

Version 1.3.0 (released 2019-11-15)
• Adds explicit excludes of needs feature to load permission

Version 1.2.0 (released 2019-08-02)
• Removes DynamicPermission

Version 1.1.0 (released 2018-12-14)

Version 1.0.2 (released 2018-10-31)
• Additional test for AnyonymousIdentity loaded on request

Version 1.0.1 (released 2018-05-18)
• Removal of Click warning messages.

Version 1.0.0 (released 2018-03-23)
• Initial public release.

3.3 License

MIT License
Copyright (C) 2015-2018 CERN.
Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

**Note:** In applying this license, CERN does not waive the privileges and immunities granted to it by virtue of its status as an Intergovernmental Organization or submit itself to any jurisdiction.

### 3.4 Contributors

- Alizee Pace
- Bruno Cuc
- Diego Rodriguez
- Dimitrios Semitsoglou-Tsiapos
- Dinos Kousidis
- Javier Delgado
- Javier Martin Montull
- Jiri Kuncar
- Krzysztof Nowak
- Lars Holm Nielsen
- Leonardo Rossi
- Nicola Tarocco
- Nicolas Harraudeau
- Nikos Filippakis
- Paulina Lach
- Rémi Ducceschi
- Sami Hiltunen
- Samuele Kaplun
- Sebastian Witowski
- Tibor Simko
invenio_access, 5
invenio_access.cli, 17
invenio_access.config, 5
invenio_access.ext, 13
invenio_access.factory, 14
invenio_access.models, 15
invenio_access.proxies, 17
invenio_access.utils, 17
Index

A
ACCESS_ACTION_CACHE_PREFIX (in module invenio_access.config), 5
ACCESS_CACHE (in module invenio_access.config), 5
ACCESS_LOAD_SYSTEM_ROLE_NEEDS (in module invenio_access.config), 5
action (invenio_access.models.ActionNeedMixin attribute), 15
action_factory () (in module invenio_access.factory), 14
ActionNeedMixin (class in invenio_access.models), 15
ActionRoles (class in invenio_access.models), 16
ActionSystemRoles (class in invenio_access.models), 16
ActionUsers (class in invenio_access.models), 16
allow () (invenio_access.models.ActionNeedMixin class method), 15
allow_action (in module invenio_access.cli), 18
allow_by_default (invenio_access.permissions.Permission attribute), 14
any_user (in module invenio_access.permissions), 15
argument (invenio_access.models.ActionNeedMixin attribute), 15
authenticated_user (in module invenio_access.permissions), 15

C
changed_action () (in module invenio_access.models), 17
create () (invenio_access.models.ActionNeedMixin class method), 15
create () (invenio_access.models.ActionSystemRoles class method), 16
current_access (in module invenio_access.proxies), 17

D
deny () (invenio_access.models.ActionNeedMixin class method), 15
deny_action (in module invenio_access.cli), 18

E
eclude (invenio_access.models.ActionNeedMixin attribute), 16
excludes (invenio_access.permissions.Permission attribute), 14

G
get_action_cache_key () (in module invenio_access.models), 17
get_identity () (in module invenio_access.utils), 17

I
id (invenio_access.models.ActionNeedMixin attribute), 16
init_app () (invenio_access.ext.InvenioAccess method), 13
init_config () (invenio_access.ext.InvenioAccess method), 13
invenio_access (module), 5
invenio_access.cli (module), 17
invenio_access.config (module), 5
invenio_access.ext (module), 13
invenio_access.factory (module), 14
invenio_access.models (module), 15
invenio_access.proxies (module), 17
invenio_access.utils (module), 17
InvenioAccess (class in invenio_access.ext), 13

L
lazy_result () (in module invenio_access.cli), 17
list_actions (in module invenio_access.cli), 18
load_permissions_on_identity_loaded (in module invenio_access.loaders), 15

N
need (invenio_access.models.ActionNeedMixin attribute), 16
need (invenio_access.models.ActionRoles attribute), 16
need (invenio_access.models.ActionSystemRoles attribute), 16
need (invenio_access.models.ActionUsers attribute), 17
needs (invenio_access.permissions.Permission attribute), 14

P
ParameterizedActionNeed (in module invenio_access.permissions), 14
Permission (class in invenio_access.permissions), 14
process_action () (in module invenio_access.cli), 17
process_allow_action () (in module invenio_access.cli), 17
process_deny_action () (in module invenio_access.cli), 17
process_email () (in module invenio_access.cli), 17
process_remove_action () (in module invenio_access.cli), 17
process_role () (in module invenio_access.cli), 18

Q
query_by_action () (in module invenio_access.models.ActionNeedMixin class method), 16

R
removed_or_inserted_action () (in module invenio_access.models), 17

S
superuser_access (in module invenio_access.permissions), 15
SystemRoleNeed (in module invenio_access.permissions), 15

V
validate_role_name () (in module invenio_access.models.ActionSystemRoles method), 16