django-ca Documentation

Release 1.10.0

Mathias Ertl

Nov 03, 2018

Contents

django-ca is a tool to manage TLS certificate authorities and easily issue and revoke certificates. It is based cryptography and Django. It can be used as an app in an existing Django project or stand-alone with the basic project included. Everything can be managed via the command line via *manage.py* commands - so no webserver is needed, if you're happy with the command-line.

Features:

- Create certificate authorities, issue and revoke certificates in minutes.
- Receive e-mail notifications of certificates about to expire.
- Certificate validation via the included OCSP responder and Certificate Revocation Lists (CRLs).
- Complete, consistent and powerful command line interface.
- Optional web interface for certificate handling (e.g. issuing, revoking, . . .).
- Written in pure Python2.7/Python3.4+, using Django 1.11 or later.

Installation/Configuration:

CHAPTER 1

Installation

You can run **django-ca** as a regular app in any existing Django project of yours, but if you don't have any Django project running, you can run it as a *standalone project*.

Another easy way of running **django-ca** is as a *Docker container*.

1.1 Requirements

- Python 2.7 or Python 3.4+
- Django 1.11+
- Any database supported by Django (sqlite3/MySQL/PostgreSQL/...)
- Python, OpenSSL and libffi development headers

If you're using an older system, you can consult this table to see what versions of Python, Django and cryptography where tested with what release:

django-ca	Python	Django	cryptography
1.4	2.7/3.4 - 3.6	1.8 - 1.10	1.7
1.5	2.7/3.4 - 3.6	1.8 - 1.11	1.7
1.6	2.7/3.4 - 3.6	1.8, 1.10 - 1.11	1.8
1.7	2.7/3.4 - 3.6	1.8, 1.10 - 2.0	2.1 - 2.2
1.8	2.7/3.4 - 3.6	1.11 - 2.0	2.1 - 2.2
1.9	2.7/3.4 - 3.6	1.11 - 2.1	2.1 - 2.3
1.10	2.7/3.4 - 3.7	1.11 - 2.1	2.1 - 2.3

1.2 As Django app (in your existing Django project)

This chapter assumes that you have an already running Django project and know how to use it.

You need various development headers for pyOpenSSL, on Debian/Ubuntu systems, simply install these packages:

```
$ apt-get install gcc python3-dev libffi-dev libssl-dev
```

You can install **django-ca** simply via pip:

```
$ pip install django-ca
```

and add it to your `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    # ... your other apps...

    'django_ca',
]
```

... and configure the *other available settings* to your liking, then simply run:

```
$ python manage.py migrate
$ python manage.py collectstatic

# FINALLY, create the root certificates for your CA:
#      (replace parameters after init_ca with your local details)
$ python manage.py init_ca RootCA \
>      /C=AT/ST=Vienna/L=Vienna/O=Org/OU=OrgUnit/CN=ca.example.com
```

After that, **django-ca** should show up in your admin interface (see *Web interface*) and provide various `manage.py` commands (see *Command-line interface*).

1.3 As standalone project

You can also install **django-ca** as a stand-alone project, if you install it via git. The project provides a *command-line interface* that provides complete functionality. The *web interface* is optional.

Note: If you don't want the private keys of your CAs on the same machine as the web interface, you can also host the web interface on a second server that accesses the same database (CA private keys are hosted on the filesystem, not in the database). You obviously will not be able to sign certificates using the web interface, but you can still e.g. revoke certificates or run a *OCSP responder*.

In the following code-snippet, you'll do all necessary steps to get a basic setup:

```
# install dependencies (adapt to your distro):
$ apt-get install gcc git python3-dev libffi-dev libssl-dev virtualenv

# clone git repository:
$ git clone https://github.com/mathiasertl/django-ca.git

# create virtualenv:
$ cd django-ca
$ virtualenv -p /usr/bin/python3 .
$ source bin/activate

# install Python dependencies:
```

(continues on next page)

(continued from previous page)

```
$ pip install -U pip setuptools
$ pip install -r requirements.txt
```

In the above script, you have created a virtualenv, meaning that all libraries you install with `pip install` are installed in the virtualenv (and don't pollute your system). It also means that before you execute any `manage.py` commands, you'll have to activate your virtualenv, by doing, in the directory of the git checkout:

```
$ source bin/activate
```

1.3.1 Configure django-ca

Before you continue, you have to configure **django-ca**. Django uses a file called `settings.py`, but so you don't have to change any files managed by git, it includes `localsettings.py` in the same directory. So copy the example file and edit it with your favourite editor:

```
$ cp ca/ca/localsettings.py.example ca/ca/localsettings.py
```

The most important settings are documented there, but you can of course use any setting provided by Django.

Warning: The `SECRET_KEY` and `DATABASES` settings are absolutely mandatory. If you use the *Web interface*, the `STATIC_ROOT` setting is also mandatory.

1.3.2 Initialize the project

After you have configured **django-ca**, you need to initialize the project by running a few `manage.py` commands:

```
$ python ca/manage.py migrate

# If you intend to run the webinterface (requires STATIC_ROOT setting!)
$ python ca/manage.py collectstatic

# FINALLY, create a certificate authority:
#     (replace parameters after init_ca with your local details)
$ python manage.py init_ca RootCA /C=AT/ST=Vienna/L=Vienna/O=Org/CN=ca.example.com
```

Please also see *Certificate authority management* for further information on how to create certificate authorities. You can also run `init_ca` with the `-h` parameter for available arguments.

1.3.3 Create manage.py shortcut

If you don't want to always chdir to the git checkout, activate the virtualenv and only then run `manage.py`, you might want to create a shortcut shell script somewhere in your `PATH` (e.g. `/usr/local/bin`):

```
#!/bin/bash

# BASEDIR is the location of your git checkout
BASEDIR=/usr/local/share/ca
PYTHON=${BASEDIR}/bin/python
MANAGE=${BASEDIR}/ca/manage.py
```

(continues on next page)

(continued from previous page)

```
${PYTHON} ${MANAGE} "$@"
```

1.3.4 Setup a webserver

Setting up a webserver and all that comes with it is really out of scope of this document. The WSGI file is located in `ca/ca/wsgi.py`. Django itself provides some info for using Apache and mod_wsgi, or you could use uWSGI and nginx, or any of the many other options available.

1.4 Apache and mod_wsgi

Github user Raoul Thill notes that you need some special configuration variable if you use Apache together with mod_wsgi (see here):

```
WSGIDaemonProcess django_ca processes=1 python-path=/opt/django-ca/ca:/opt/django-ca/
↪ca/ca:/opt/django-ca/lib/python2.7/site-packages threads=5
WSGIProcessGroup django_ca
WSGIApplicationGroup %{GLOBAL}
WSGIScriptAlias / /opt/django-ca/ca/ca/wsgi.py
```

1.5 Regular cronjobs

Some `manage.py` commands are intended to be run as cronjobs:

```
# assuming you cloned the repo at /root/:
HOME=/root/django-ca
PATH=/root/django-ca/bin

# m h  dom mon dow      user  command

# notify watchers about certificates about to expire
* 8    * * *            root  python ca/manage.py notify_expiring_certs

# recreate the CRL and the OCSP index
12 *    * * *           root  python ca/manage.py dump_crl
14 *    * * *           root  python ca/manage.py dump_ocsp_index
```

CHAPTER 2

Docker

There is a **django-ca** Docker container available.

Assuming you have Docker installed, simply start the docker container with:

```
docker run --name=django-ca -p 8000:8000 mathiasertl/django-ca
```

You still need the shell to create one or more root CAs. For the admin interface, we also create a superuser:

```
docker exec -it django-ca python ca/manage.py createsuperuser
docker exec -it django-ca python ca/manage.py init_ca \
   example /C=AT/ST=Vienna/L=Vienna/O=Org/CN=ca.example.com
```

... and visit http://localhost:8000/admin/.

2.1 Configuration

Every environment variable passed to the container that starts with `DJANGO_CA_` is loaded as a normal setting:

```
docker run -e DJANGO_CA_CA_DIGEST_ALGORITHM=sha256 ...
```

This however only works for settings that are supposed to be a string. For more complex settings, you can pass a YAML configuration file. For example, if you create a file `/etc/django-ca/settings.yaml`:

```
# Certificates expire after ten years, default profile is "server":
CA_DEFAULT_EXPIRES: 3650
CA_DEFAULT_PROFILE: server

# The standard Django DATABASES setting, see Django docs:
DATABASES:
   default:
      ENGINE: ...
```

And then start the container with:

```
docker run -v /etc/django-ca/:/etc/django-ca \
   -e DJANGO_CA_SETTINGS=/etc/django-ca/settings.yaml ...
```

... the container will load your settings file.

2.1.1 uWSGI

The container starts a uWSGI instance to let you use the admin interface. To replace the simple default configuration for something else, you can pass `DJANGO_CA_UWSGI_INI` as environment variable to set a different location:

```
docker run -v /etc/django-ca/:/etc/django-ca \
   -e DJANGO_CA_UWSGI_INI=/etc/django-ca/uwsgi.ini ...
```

The docker container comes with different ini files, each located in `/usr/src/django-ca/uwsgi/`:

config	Description
standalone.ini	**Default**. Serves plain HTTP on port 8000, including static files. Suitable for basic setups.
uwsgi.ini	Serves the uwsgi protocol supported by NGINX and Apache. Does not serve static files, has three worker processes.

You can also always pass additional parameters to uWSGI using the `DJANGO_CA_UWSGI_PARAMS` environment variable. For example, to start six worker processes, simply use:

```
docker run -v /etc/django-ca/:/etc/django-ca \
   -e DJANGO_CA_UWSGI_PARAMS="--processes=6" ...
```

2.1.2 Use NGINX or Apache

In more professional setups, uWSGI will not serve HTTP directly, but a webserver like Apache or NGINX will be a proxy to uWSGI communicating via a dedicated protocol. Usually, the webserver serves static files directly and not via uWSGI.

Note: uWSGI supports a variety of webservers: https://uwsgi-docs.readthedocs.io/en/latest/WebServers.html

First, you need to create a directory that you can use as a Docker volume that will contain the static files that are served by the webserver. Note that the process in the container runs with uid/gid of 9000 by default:

```
sudo mkdir /usr/share/django-ca
sudo chown 9000:9000 /usr/share/django-ca
```

Now configure your webserver appropriately, e.g. for NGINX:

```
server {
    # ... everything else

    location / {
        uwsgi_pass 127.0.0.1:8000;
        include uwsgi_params;
    }
```

(continues on next page)

(continued from previous page)

```
    location /static/ {
        alias /home/mati/git/mati/django-ca/static/static/;
    }
}
```

Now all that's left is to start the container with that volume and set `DJANGO_CA_UWSGI_INI` to a different ini file (note that this file is included in the container, see above):

```
docker run \
   -e DJANGO_CA_UWSGI_INI=/usr/src/django-ca/uwsgi/uwsgi.ini \
   -p 8000:8000 --name=django-ca \
   -v /usr/share/django-ca:/usr/share/django-ca \
   django-ca
```

Note that `/usr/share/django-ca` on the host will now contain the static files served by your webserver. If you configured NGINX on port 80, you can now visit e.g. http://localhost/admin/ for the admin interface.

2.1.3 Run as different user

It is possible to run the uWSGI instance inside the container as a different user, *but* you have to make sure that `/var/lib/django-ca/` and `/usr/share/django-ca/` are writable by that user.

> **Warning:** `/var/lib/django-ca/` contains all sensitive data including CA private keys and login credentials to the admin interface. Make sure you protect this directory!

Assuming you want to use uid 3000 and gid 3001, set up appropriate folders on the host:

```
mkdir /var/lib/django-ca/ /usr/share/django-ca/
chown 3000:3001 /var/lib/django-ca/ /usr/share/django-ca/
chmod go-rwx /var/lib/django-ca/
```

If you want to keep any existing data, you now must copy the data for `/var/lib/django-ca/` in the container to the one on the host.

Now you can run the container with the different uid/gid:

```
docker run \
   -p 8000:8000 --name=django-ca \
   -v /usr/share/django-ca:/usr/share/django-ca \
   -v /var/lib/django-ca:/var/lib/django-ca \
   --user 3000:3001 \
   django-ca
```

2.2 Build your own container

If you want to build the container by yourself, simply clone the repository and execute:

```
docker build -t django-ca .
```

CHAPTER 3

Update

Since 1.0.0, this project updates like any other project. First, update the source code, if you use git:

```
git pull origin master
```

or if you installed **django-ca** via pip:

```
pip install -U django-ca
```

then upgrade with these commands:

```
pip install -U -r requirements.txt
python ca/manage.py migrate

# if you use the webinterface
python ca/manage.py collectstatic
```

Warning: If you installed **django-ca** in a virtualenv, don't forget to activate it before executing any python or pip commands using:

```
source bin/activate
```

3.1 Update from 1.0.0b2

If you're updating from a version earlier then 1.0.0 (which was the first real release), you have to first update to 1.0.0.b1 (see below), then to 1.0.0.b2, apply all migrations and reset existing migrations Since all installed instances were probably private, it made sense to start with a clean state.

To update from an earlier git-checkout, to:

- Upgrade to version 1.0.0b2

- Apply all migrations.
- Upgrade to version 1.0.0
- Remove old migrations from the database:

```
python manage.py dbshell
> DELETE FROM django_migrations WHERE app='django_ca';
```

- Fake the first migration:

  python manage.py migrate django_ca 0001 –fake

3.2 Update from pre 1.0.0b1

Prior to 1.0.0, this app was not intended to be reusable and so had a generic name. The app was renamed to *django_ca*, so it can be used in other Django projects (or hopefully stand-alone, someday). Essentially, the upgrade path should work something like this:

```
# backup old data:
python manage.py dumpdata certificate --indent=4 > certs.json

# update source code
git pull origin master

# create initial models in the new app, but only the initial version!
python manage.py migrate django_ca 0001

# update JSON with new model name
sed 's/"certificate.certificate"/"django_ca.certificate"/' > certs-updated.json

# load data
python manage.py loaddata certs-updated.json

# apply any other migrations
python manage.py migrate
```

CHAPTER 4

ChangeLog

4.1 1.10.0 (2018-11-03)

- New dependency: django-object-actions.
- Add ability to resign existing certificates.
- Management command `list_cas` now optionally supports a tree view.
- Use more consistent naming for extensions throughout the code and documentation.
- Renamed the `--tls-features` option of the `sign_cert` command to `--tls-feature`, in line with the actual name of the extension.
- Allow the `TLSFeature` extension in profiles.
- Add link in the admin interface to easily download certificate bundles.
- Support ECC private keys for new Certificate Authorities.
- Store CA private keys in the more secure PKCS8 format.
- The Certificate change view now has a second "Revoke" button as object action next to the "History" button.

4.1.1 Python API

- Add the *Python API* as a fully supported interface to **django-ca**.
- New module *`django_ca.extensions`* to allow easy and consistent handling of X509 extensions.
- Fully document various member attributes of *`CertificateAuthority`* and *`Certificate`*, as well *`Subject`* and as all new Python code.
- The parameters for functions in *`CertificateManager`* and *`CertificateAuthorityManager`* were cleaned up for consistent naming and so that a user no longer needs to use classes from the cryptography libary. Parameters are now optional if default settings exist.
- Variable names have been renamed to be more consistent to make the code more readable.

4.1.2 Testing

- Also test with Python 3.7.0.
- Add configuration for tox.
- Speed up test-suite by using `force_login()` and PASSWORD_HASHERS.
- Load keys and certs in for every testcase instead for every class, improving testcase isolation.
- Add two certificates that include all and no extensions at all respectively to be able to test edge cases more consistently and thoroughly.
- Add function `cmd_e2e` to call `manage.py` scripts in a way that arguments are passed by argparse as if they where called from the command-line. This allows more complete testing including parsing commandline arguments.
- Error on any `warnings` coming from django-ca when running the test-suite.

4.2 1.9.0 (2018-08-25)

- Allow the creation of Certificates with multiple OUs in their subject (command-line only).
- Fix issues with handling CAs with a password on the command-line.
- Fix handling of certificates with no CommonName and/or no x509 extensions.
- Add support for displaying Signed Certificate Timestamps (SCT) Lists, as described in RFC 6962, section 3.3.
- Add limited support for displaying Certificate Policies, as described in RFC 5280, section 4.2.14 and RFC 3647.
- Correctly display extensions with an OID unknown to django-ca or even cryptography.
- Properly escape x509 extensions to prevent any injection attacks.
- Django 2.1 is now fully supported.
- Fix example command to generate a CSR (had a stray '/').
- Run test-suite with template debugging enabled to catch silently skipped template errors.

4.2.1 Docker

- Base the *Docker image* on `python:3-alpine` (instead of `python:3`), yielding a much smaller image (~965MB -> ~235MB).
- Run complete test-suite in a separate build stage when building the image.
- Provide `uwsgi.ini` for fast deployments with the uwsgi protocol.
- Add support for passing additional parameters to uWSGI using the `DJANGO_CA_UWSGI_PARAMS` environment variable.
- Create user/group with a predefined uid/gid of 9000 to allow better sharing of containers.
- Add `/usr/share/django-ca/` as named volume, allowing a setup where an external webserver serves static files.
- Add documentation on how to run the container in combination with an external webserver.
- Add documentation on how to run the container as a different uid/gid.

4.3 1.8.0 (2018-07-08)

- Add *Django signals* to important events to let users add custom actions (such as email notifications etc.) to those events (fixes #39).
- Provide a Docker container for fast deployment of **django-ca**.
- Add the *CA_CUSTOM_APPS* setting to let users that use **django-ca** as a *standalone project* add custom apps, e.g. to register signals.
- Make the `otherName` extension actually usable and tested (see PR47)
- Add the `smartcardLogon` and `msKDC` extended key usage types. They are needed for some AD and OpenLDAP improvements (see PR46)
- Improve compatability with newer `idna` versions (`".com"` now also throws an error).
- Drop support for Django 1.8 and Django 1.10.
- Improve support for yet-to-be-released Django 2.1.
- Fix admin view of certificates with no subjectAltName.

4.4 1.7.0 (2017-12-14)

- Django 2.0 is now fully supported. This release still supports Django 1.8, 1.10 and 1.11.
- Add support for the *TLSFeature* extension.
- Do sanity checks on the "pathlen" attribute when creating Certificate Authorities.
- Add sanity checks when creating CAs:
  - When creating an intermediate CA, check the `pathlen` attribute of the parent CA to make sure that the resulting CA is not invalid.
  - Refuse to add a CRL or OCSP service to root CAs. These attributes are not meaningful there.
- Massively update *documentation for the command-line interface*.
- CAs can now be identified using name or serial (previously: only by serial) in *CA_OCSP_URLS*.
- Make `fab init_demo` a lot more useful by signing certificates with the client CA and include CRL and OCSP links.
- Run `fab init_demo` and documentation generation through Travis-CI.
- Always display all extensions in the django admin interface.
- NameConstraints are now delimited using a `,` instead of a `;`, for consistency with other parameters and so no bash special character is used.

4.4.1 Bugfixes

- Check for permissions when downloading certificates from the admin interface. Previously, users without admin interface access but without permissions to access certificates, where able to guess the URL and download public keys.
- Add a missing migration.
- Fix the value of the crlDistributionPoints x509 extension when signing certificates with Python2.

- The `Content-Type` header of CRL responses now defaults to the correct value regardless of type (DER or PEM) used.
- If a wrong CA is specified in *CA_OCSP_URLS*, an OCSP internal error is returned instead of an uncought exception.
- Fix some edge cases for serial conversion in Python2. Some serials where converted with an "L" prefix in Python 2, because `hex(0L)` returns `"0x0L"`.

4.5 1.6.3 (2017-10-21)

- Fix various operations when `USE_TZ` is `True`.
- Email addresses are now independently validated by `validate_email`. cryptography 2.1 no longer validates email addresses itself.
- Require `cryptography>=2.1`. Older versions should not be broken, but the output changes breaking doctests, meaning they're no longer tested either.
- CA keys are no longer stored with colons in their filename, fixing `init_ca` under Windows.

4.6 1.6.2 (2017-07-18)

- No longer require a strict cryptography version but only `>=1.8`. The previously pinned version is incompatible with Python 3.5.
- Update requirements files to newest versions.
- Update imports to `django.urls.reverse` so they are compatible with Django 2.0 and 1.8.
- Make sure that `manage.py check` exit status is not ignored for `setup.py code_quality`.
- Conform to new sorting restrictions for `isort`.

4.7 1.6.1 (2017-05-05)

- Fix signing of wildcard certificates (thanks RedNixon).
- Add new management commands `import_ca` and `import_cert` so users can import existing CAs and certificates.

4.8 1.6.0 (2017-04-21)

4.8.1 New features and improvements

- Support CSRs in DER format when signing a certificate via `manage.py sign_cert`.
- Support encrypting private keys of CAs with a password.
- Support Django 1.11.
- Allow creating CRLs of disabled CAs via `manage.py dump_crl`.

- Validate DNSNames when parsing general names. This means that signing a certificate with CommonName that is not a valid domain name fails if it should also be added as subjectAltName (see `--cn-in-san` option).
- When configuring *`OCSPView`*, the responder key and certificate are verified during configuration. An erroneous configuration thus throws an error on startup, not during runtime.
- The testsuite now tests certificate signatures itself via `pyOpenSSL`, so an independent library is used for verification.

4.8.2 Bugfixes

- Fix the `authorityKeyIdentifier` extension when signing certificates with an intermediate CA.
- Fix creation of intermediate CAs.

4.9 1.5.1 (2017-03-07)

- Increase minimum field length of serial and common name fields.
- Tests now call full_clean() for created models. SQLite (which is used for testing) does not enforce the `max_length` parameter.

4.10 1.5.0 (2017-03-05)

- Completely remove pyOpenSSL and consistently use cryptography.
- Due to the transitition to cryptography, some features have been removed:
  - The `tlsfeature` extension is no longer supported. It will be again once cryptography adds support.
  - The `msCodeInd`, `msCodeCom`, `msCTLSign`, `msEFS` values for the ExtendedKeyUsage extension are no longer supported. Support for these was largely academic anyway, so they most likely will not be added again.
  - `TEXT` is no longer a supported output format for dumping certificates.
- The `keyUsage` extension is now marked as critical for certificate authorities.
- Add the `privilegeWithdrawn` and `aACompromise` attributes for revocation lists.

4.11 1.4.1 (2017-02-26)

- Update requirements.
- Use Travis CI for continuous integration. **django-ca** is now tested with Python 2.7, 3.4, 3.5, 3.6 and nightly, using Django 1.8, 1.9 and 1.10.
- Fix a few test errors for Django 1.8.
- Examples now consistently use 4096 bit certificates.
- Some functionality is now migrated to `cryptography` in the ongoing process to deprecate pyOpenSSL (which is no longer maintained).
- OCSPView now supports directly passing the public key as bytes. As a consequence, a bad certificate is now only detected at runtime.

4.12 1.4.0 (2016-09-09)

- Make sure that Child CAs never expire after their parents. If the user specifies an expiry after that of the parent, it is silently changed to the parents expiry.
- Make sure that certificates never expire after their CAs. If the user specifies an expiry after that of the parent, throw an error.
- Rename the `--days` parameter of the `sign_cert` command to `--expires` to match what we use for `init_ca`.
- Improve help-output of `--init-ca` and `--sign-cert` by further grouping arguments into argument groups.
- Add ability to add CRL-, OCSP- and Issuer-URLs when creating CAs using the `--ca-*` options.
- Add support for the `nameConstraints` X509 extension when creating CAs. The option to the `init_ca` command is `--name-constraint` and can be given multiple times to indicate multiple constraints.
- Add support for the `tlsfeature` extension, a.k.a. "TLS Must Staple". Since OpenSSL 1.1 is required for this extension, support is currently totally untested.

4.13 1.3.0 (2016-07-09)

- Add links for downloading the certificate in PEM/ASN format in the admin interface.
- Add an extra chapter in documentation on how to create intermediate CAs.
- Correctly set the issuer field when generating intermediate CAs.
- `fab init_demo` now actually creates an intermediate CA.
- Fix help text for the `--parent` parameter for `manage.py init_ca`.

4.14 1.2.2 (2016-06-30)

- Rebuild to remove old migrations accidentally present in previous release.

4.15 1.2.1 (2016-06-06)

- Add the `CA_NOTIFICATION_DAYS` setting so that watchers don't receive too many emails.
- Fix changing a certificate in the admin interface (only watchers can be changed at present).

4.16 1.2.0 (2016-06-05)

- **django-ca** now provides a complete *OCSP responder*.
- Various tests are now run with a pre-computed CA, making tests much fater and output more predictable.
- Update lots of documentation.

4.17 1.1.1 (2016-06-05)

- Fix the `fab init_demo` command.
- Fix installation via `setup.py install`, fixes #2 and #4. Thanks to Jon McKenzie for the fixes!

4.18 1.1.0 (2016-05-08)

- The subject given in the `manage.py init_ca` and `manage.py sign_cert` is now given in the same form that is frequently used by OpenSSL, "/C=AT/L=...".
- On the command line, both CAs and certificates can now be named either by their CommonName or with their serial. The serial can be given with only the first few letters as long as it's unique, as it is matched as long as the serial starts with the given serial.
- Expiry time of CRLs can now be specified in seconds. `manage.py dump_crl` now uses the `--expires` instead of the old `--days` parameter.
- The admin interface now accounts for cases where some or all CAs are not useable because the private key is not accessable. Such a scenario might occur if the private keys are hosted on a different machine.
- The app now provides a generic view to generate CRLs. See *Use generic view to host a CRL* for more information.
- Fix the display of the default value of the –ca args.
- Move this ChangeLog from a top-level .md file to this location.
- Fix shell example when issueing certificates.

4.19 1.0.1 (2016-04-27)

- Officially support Python2.7 again.
- Make sure that certificate authorities cannot be removed via the web interface.

4.20 1.0.0 (2016-04-27)

This represents a massive new release (hence the big version jump). The project now has a new name (**django-ca** instead of just "certificate authority") and is now installable via pip. Since versions prior to this release probably had no users (as it wasn't advertised anywhere), it includes several incompatible changes.

4.20.1 General

- This project now runs under the name **django-ca** instead of just "certificate authority".
- Move the git repository is now hosted at https://github.com/mathiasertl/django-ca.
- This version now absolutely assumes Python3. Python2 is no longer supported.
- Require Django 1.8 or later.
- django-ca is now usable as a stand-alone project (via git) or as a reusable app (via pip).

4.20.2 Functionality

- The main app was renamed from `certificate` to `django_ca`. See below for how to upgrade.

4.20.3 manage.py interface

- `manage.py` commands are now renamed to be more specific:
  - `init` -> `init_ca`
  - `sign` -> `sign_cert`
  - `list` -> `list_certs`
  - `revoke` -> `revoke_cert`
  - `crl` -> `dump_crl`
  - `view` -> `view_cert`
  - `watch` -> `notify_expiring_certs`
  - `watchers` -> `cert_watchers`
- Several new `manage.py` commands:
  - `dump_ca` to dump CA certificates.
  - `dump_cert` to dump certificates to a file.
  - `dump_ocsp_index` for an OCSP responder, `dump_crl` no longer outputs this file.
  - `edit_ca` to edit CA properties from the command line.
  - `list_cas` to list available CAs.
  - `view_ca` to view a CA.
- Removed the `manage.py remove` command.
- `dump_{ca,cert,crl}` can now output DER/ASN1 data to stdout.

4.21 0.2.1 (2015-05-24)

- Signed certificates are valid five minutes in the past to account for possible clock skew.
- Shell-scripts: Correctly pass quoted parameters to manage.py.
- Add documentation on how to test CRLs.
- Improve support for OCSP.

4.22 0.2 (2015-02-08)

- The `watchers` command now takes a serial, like any other command.
- Reworked `view` command for more robustness.
  - Improve output of certificate extensions.
  - Add the `-n`/`--no-pem` option.

  - Add the `-e`/`--extensions` option to print all certificate extensions.
  - Make output clearer.
- The `sign` command now has
  - a `--key-usage` option to override the `keyUsage` extended attribute.
  - a `--ext-key-usage` option to override the `extendedKeyUsage` extended attribute.
  - a `--ocsp` option to sign a certificate for an OCSP server.
- The default `extendedKeyUsage` is now `serverAuth`, not `clientAuth`.
- Update the remove command to take a serial.
- Ensure restrictive file permissions when creating a CA.
- Add requirements-dev.txt

4.23 0.1 (2015-02-07)

- Initial release

CHAPTER 5

Custom settings

You can use any of the settings understood by Django and **django-ca** provides some of its own settings.

From Djangos settings, you especially need to configure `DATABASES`, `SECRET_KEY`, `ALLOWED_HOSTS` and `STATIC_ROOT`.

All settings used by **django-ca** start with the `CA_` prefix. Settings are also documented at `ca/ca/localsettings.py.example` (view on git).

CA_CUSTOM_APPS Default: `[]`

This setting is only used when you use **django-ca** as a standalone project to let you add custom apps to the project, e.g. to add *Signals*.

The list gets appended to the standard `INSTALLED_APPS` setting. If you need more control, you can always override that setting instead.

CA_DEFAULT_ECC_CURVE Default: `"SECP256R1"`

The default elliptic curve used for generating CA private keys when ECC is used.

CA_DEFAULT_EXPIRES Default: `730`

The default time, in days, that any signed certificate expires.

CA_DEFAULT_PROFILE Default: `webserver`

The default profile to use.

CA_DEFAULT_SUBJECT Default: `{}`

The default subject to use. The keys of this dictionary are the valid fields in X509 certificate subjects. Example:

```
CA_DEFAULT_SUBJECT = {
    'C': 'AT',
    'ST': 'Vienna',
    'L': 'Vienna',
    'O': 'HTU Wien',
    'OU': 'Fachschaft Informatik',
```

(continues on next page)

(continued from previous page)

```
    'emailAddress': 'user@example.com',
}
```

CA_DIGEST_ALGORITHM Default: `"sha512"`

The default digest algorithm used to sign certificates. You may want to use `"sha256"` for older (pre-2010) clients. Note that this setting is also used by the `init_ca` command, so if you have any clients that do not understand sha512 hashes, you should change this beforehand.

CA_DIR Default: `"ca/files"`

Where the root certificate is stored. The default is a `files` directory in the same location as your `manage.py` file.

CA_NOTIFICATION_DAYS Default: `[14, 7, 3, 1, ]`

Days before expiry that certificate watchers will receive notifications. By default, watchers will receive notifications 14, seven, three and one days before expiry.

CA_OCSP_URLS Default: `{}`

Configuration for OCSP responders. See *Run a OCSP responder* for more information.

CA_PROFILES Default: `{}`

Profiles determine the default values for the `keyUsage`, `extendedKeyUsage` x509 extensions. In short, they determine how your certificate can be used, be it for server and/or client authentication, e-mail signing or anything else. By default, **django-ca** provides these profiles:

Profile	keyUsage	extendedKeyUsage
client	digitalSignature	clientAuth
server	digitalSignature, keyAgreement keyEncipherment	clientAuth, serverAuth
webserver	digitalSignature, keyAgreement keyEncipherment	serverAuth
enduser	dataEncipherment, digitalSignature, keyEncipherment	clientAuth, emailProtection, codeSigning
ocsp	nonRepudiation, talSignature, keyEncipherment	OCSPSigning

Further more,

- The `keyUsage` attribute is marked as critical.
- The `extendedKeyUsage` attribute is marked as non-critical.

This should be fine for most usecases. But you can use the `CA_PROFILES` setting to either update or disable existing profiles or add new profiles that you like. For that, set `CA_PROFILES` to a dictionary with the keys defining the profile name and the value being either:

- `None` to disable an existing profile.
- A dictionary defining the profile. If the name of the profile is an existing profile, the dictionary is updated, so you can ommit a value to leave it as the default. The possible keys are:

key	Description
`"keyUsage"`	The `keyUsage` X509 extension.
`"extendedKeyUsage"`	The `extendedKeyUsage` X509 extension.
`"desc"`	A human-readable description, shows up with "sing_cert -h" and in the webinterface profile selection.
`"subject"`	The default subject to use. If ommited, `CA_DEFAULT_SUBJECT` is used.
`"cn_in_san"`	If to include the CommonName in the subjectAltName by default. The default value is `True`.

Here is a full example:

```
CA_PROFILES = {
    'client': {
        'desc': _('Nice description.'),
        'keyUsage': {
            'critical': True,
            'value': [
               'digitalSignature',
            ],
        },
        'extendedKeyUsage': {
            'critical': False,
            'value': [
               'clientAuth',
            ],
         },
         'subject': {
            'C': 'AT',
            'L': 'Vienna',
         }
     },

     # We really don't like the "ocsp" profile, so we remove it.
     'ocsp': None,
}
```

CA_PROVIDE_GENERIC_CRL Default: `True`

If set to `False`, `django_ca.urls` will not add a CRL view. See *Use generic view to host a CRL* for more information.

This setting only has effect if you use django_ca as a full project or you include the `django_ca.urls` module somewhere in your URL configuration.

Usage:

CHAPTER 6

Command-line interface

django-ca provides a complete command-line interface for all functionality. It is implemented as subcommands of Djangos `manage.py` script. You can use it for all certificate management operations, and *Certificate authority management* is only possible via the command-line interface for security reasons.

In general, run `manage.py` without any parameters for available subcommands:

```
$ python manage.py

...
[django_ca]
    cert_watchers
    dump_cert
    dump_crl
    ...
```

Creating Certificate Authorities and managing Certificates is documented on individual pages:

6.1 Certificate authority management

django-ca supports managing multiple certificate authorities as well as child certificate authorities.

The *command-line interface* is the only way to create certificate authorities. It is obviously most important that the private keys are never exposed to any attacker, and any web interface would pose an unnecessary risk. Some details, like the x509 extensions used for signing certificates, can be configured using the web interface.

For the same reason, the private key of a certificate authority is stored on the filesystem and not in the database. The initial location of the private key is configured by the *CA_DIR setting*. This also means that you can run your **django-ca** on two hosts, where one host has the private key and only uses the command line, and one with the webinterface that can still be used to revoke certificates.

6.1.1 Index of commands

To manage certificate authorities, use the following *manage.py* commands:

Command	Description
dump_ca	Write the CA certificate to a file.
edit_ca	Edit a certificate authority.
import_ca	Import an existing certificate authority.
init_ca	Create a new certificate authority.
list_cas	List all currently configured certificate authorities.
view_ca	View details of a certificate authority.

Like all *manage.py* subcommands, you can run `manage.py <subcomand> -h` to get a list of availabble parameters.

6.1.2 Create a new CA

You should be very careful when creating a new certificate authority, especially if it is used by a large number of clients. If you make a mistake here, it could make your CA unusable and you have to redistribute new public keys to all clients, which is usually a lot of work.

Please think carefully about how you want to run your CA: Do you want intermediate CAs? Do you want to use CRLs and/or run an OCSP responder?

`pathlen` attribute

The `pathlen` attribute says how many levels of intermediate CAs can be used below a given CA. If present, it is an integer attribute (>= 0) meaning how many intermediate CAs can be below this CA. If *not* present, the number is unlimited. For a valid setup, all `pathlen` attributes of all intermediate CAs must be correct. Here is a typical (correct) example:

```
root    # pathlen: 2
|- child_A  # pathlen 1
   |- child_A.1  # pathlen 0
|- child_B  # pathlen 0
```

In this example, *root* and *child_A* can have intermediate CAs, while *child_B* and *child_A.1* can not.

The default value for the `pathlen` attribute is `0`, meaning that any CA cannot have any intermediate CAs. You can use the `--pathlen` parameter to set a different value or the `--no-pathlen` parameter if you don't want to set the attribute:

```
# Two sublevels of intermediate CAs:
python manage.py init_ca --pathlen=2 ...

# unlimited number of intermediate CAs:
python manage.py init_ca --no-pathlen ...
```

CRL URLs

Certificate Revocation Lists (CRLs) are signed files that contain a list of all revoked certificates. Certificates (including those for CAs) can contain pointers to CRLs, usually a single URL, in the `crlDistributionPoints` extension. Clients that support this extension can query the URL and refuse to establish a connection if the certificate is revoked.

Since a CRL has to be signed by the issuing CA, root CAs cannot sensibly contain a CRL: You could only revoke the root CA with it, and it would have to be signed by the (compromised) root CA.

django-ca supports adding CRLs to (intermediate) CAs as well as end-user certificates. The former cannot be changed later, while the latter can be changed at any time for future certificates using the `edit_ca` subcommand or via the web interface.

> **Warning:** If you decide to add a CRL to CAs/certificates, you must also provide the CRLs at the given URL. **django-ca** provides everything you need, please see *Host a Certificate Revocation List (CRL)* for more information.

For certificates to be signed by this CA, use the `--crl-url` option:

```
python manage.py init_ca --ca-url http://ca.example.com/example.crl ...
```

To add a CRL url for an intermediate CA, use the `--ca-crl-url` option:

```
python manage.py init_ca \
   --parent root
   --ca-url http://ca.example.com/root.crl
   ...
```

OCSP responder

The Online Certificate Status Protocol or OCSP is a service (called "OCSP responder") run by a certificate authority that allows clients to query for revoked certificates. It is an improvement over CRLs particulary for larger CAs because a full CRL can grow quite big.

The same restrictions as for CRLs apply: You cannot add a OCSP URL to a root CA, it runs via HTTP (not HTTPS) and if you decide to add such URLs, you also have to actually run that service, or clients will refuse to connect. **django-ca** includes a somewhat tested OCSP responder, see *Run a OCSP responder* for more information.

To add a OCSP URL to certificates to be signed by this CA, use the `--ocsp-url` option:

```
python manage.py --ocsp-url http://ocsp.ca.example.com/example ...
```

To add a OCSP URL to intermediate CAs, use the `--ca-ocsp-url` option:

```
python manage.py init_ca \
   --parent root \
   --ca-ocsp-url http://ocsp.ca.example.com/root \
   ...
```

Name constraints

NameConstraints are a little-used extension (see RFC 5280, section 4.2.1.10 that allows you to create CAs that are limited to issuing certificates for a particular set of addresses. The parsing of this syntax is quite complex, see e.g. this blog post for a good explanation.

> **Warning:** This extension is marked as "critical". Any client that does not understand this extension will refuse a connection.

To add name constraints to a CA, use the `--name-constraint` option, which can be given multiple times. Values are any valid name, see *Names on the command-line* for detailed documentation. Prefix the value with either `permitted,` or `excluded,` to add them to the Permitted or Excluded subtree:

```
python manage.py init_ca \
   --name-constraint permitted,DNS:com
   --name-constraint permitted,DNS:net
   --name-constraint excluded,DNS:evil.com
   ...
```

This will restrict the CA to issuing certificates for .com and .net subdomains, except for evil.com, which obviously should never have a certificate (evil.net is good, though).

Examples

Here is a shell session that illustrates the respective manage.py commands:

```
$ python manage.py init_ca --pathlen=2
>     --crl-url=http://ca.example.com/crl \
>     --ocsp-url=http://ocsp.ca.example.com \
>     --issuer-url=http://ca.example.com/ca.crt \
>     TestCA /C=AT/L=Vienna/L=Vienna/O=Example/OU=ExampleUnit/CN=ca.example.com
$ python manage.py list_cas
BD:5B:AB:5B:A2:1C:49:0D:9A:B2:AA:BC:68:ED:ED:7D - TestCA

$ python manage.py view_ca BD:5B:AB:5B:A2
...
* OCSP URL: http://ocsp.ca.example.com
$ python manage.py edit_ca --ocsp-url=http://new-ocsp.ca.example.com \
>     BD:5B:AB:5B:A2
$ python manage.py view_ca BD:5B:AB:5B:A2
...
* OCSP URL: http://new-ocsp.ca.example.com
```

Note that you can just use the start of a serial to identify the CA, as long as that still uniquely identifies the CA.

6.1.3 Create intermediate CAs

Intermediate CAs are created, just like normal CAs, using `manage.py init_ca`. For intermediate CAs to be valid, CAs however must have a correct `pathlen` x509 extension. Its value is an integer describing how many levels of intermediate CAs a CA may have. A `pathlen` of "0" means that a CA cannot have any intermediate CAs, if it is not present, a CA may have an infinite number of intermediate CAs.

Note: **django-ca** by default sets a `pathlen` of "0", as it aims to be secure by default. The `pathlen` attribute cannot be changed in hindsight (not without resigning the CA). If you plan to create intermediate CAs, you have to consider this when creating the root CA.

So for example, if you want two levels of intermediate CAs, , you'd need the following `pathlen` values (the `pathlen` value is the minimum value, it could always be a larger number):

index	CA	pathlen	description
1	example.com	2	Your root CA.
2	sub1.example.com	1	Your first intermediate CA, a sub-CA from (1).
3	sub2.example.com	0	A second intermediate CA, also a sub-CA from (1).
4	sub.sub1.example.com	0	An intermediate CA of (2).

If in the above example, CA (1) had `pathlen` of "1" or CA (2) had a `pathlen` of "0", CA (4) would no longer be a valid CA.

By default, **django-ca** sets a `pathlen` of 0, so CAs will not be able to have any intermediate CAs. You can configure the value by passing `--pathlen` to `init_ca`:

```
$ python manage.py init_ca --pathlen=2 ...
```

When creating a sub-ca, you must name its parent using the `--parent` parameter:

```
$ python manage.py list_cas
BD:5B:AB:5B:A2:1C:49:0D:9A:B2:AA:BC:68:ED:ED:7D - Root CA
$ python manage.py init_ca --parent=BD:5B:AB:5B ...
```

Note: Just like throughout the system, you can always just give the start of the serial, as long as it still is a unique identifier for the CA.

6.2 Managing certificates

All certificate operations can be done via the command line. You do not have to use this interface, all functionality is also available via the *Web interface*, if it has access to the private key of the certificate authority.

6.2.1 Index of commands

To manage certificate, use the following manage.py commands:

Command	Description
cert_watchers	Add/remove addresses to be notified of an expiring certificate.
dump_cert	Dump a certificate to a file.
import_cert	Import an existing certificate.
list_certs	List all certificates.
notify_expiring_certs	Send notifications about expiring certificates to watchers.
revoke_cert	Revoke a certificate.
sign_cert	Sign a certificate.
view_cert	View a certificate.

Like all *manage.py* subcommands, you can run `manage.py <subcomand> -h` to get a list of availabble parameters.

6.2.2 Signing certificates

Signing certificates is done using `manage.py sign_cert`. The only requirements are that you provide either a full subject and/or one or more subjectAltNames. Obviously, you also need to create at least one certificate authority first (*documentation*).

Like any good certificate authority, **django-ca** never handles private keys of signed certificates. Instead, you sign certificates from a Certificate Signing Request (CSR) that you generate from the private key. Using the OpenSSL command-line tools, you can create a CSR *on the host that should use the certificate*:

```
$ openssl genrsa -out example.key 4096
$ openssl req -new -key example.key -out example.csr -utf8
```

Next, simply copy the CSR file (`example.csr` in the above example) to the host where you installed **django-ca**. You can now create a signed certificate using:

```
$ python manage.py sign_cert --alt example.com --csr example.csr --out example.pub
```

If you have defined multiple CAs, you also have to name the CA:

```
$ python manage.py list_cas
4E:1E:2A:29:F9:4C:45:CF:12:2F:2B:17:9E:BF:D4:80:29:C6:37:C7 - Root CA
32:BE:A9:E8:7E:21:BF:3E:E9:A1:F3:F9:E4:06:14:B4:C4:9D:B2:6C - Child CA
$ python manage.py sign_cert --ca 32:BE:A9 --alt example.com --csr example.csr --out␣
↪example.pub
```

Subject and subjectAltName

The Certificate's Subject (that is, it's CommonName) and the names given in the `subjectAltName` extension define where the certificate is valid.

The CommonName is usually added to the `subjectAltName` extension as well and vice versa. This means that these two will give the same CommonName and `subjectAltName`:

```
$ python manage.py sign_cert --subject /C=AT/.../CN=example.com
$ python manage.py sign_cert --alt example.com
```

A given CommonName is only added as `subjectAltName` if it is a valid *name*. If you give multiple names via `--alt` but no CommonName, the first one will be used as CommonName. Names passed via `alt` are parsed as *names*, so you can also use e.g.:

```
$ python manage.py sign_cert --alt IP:127.0.0.1
```

You can also disable adding the CommonName as `subjectAltName`:

```
$ python manage.py sign_cert --cn-not-in-san --subject /C=AT/.../CN=example.com --
↪alt=example.net
```

... this will only have "example.net" but not example.com as `subjectAltName`.

Using profiles

Certificates have extensions that define certain aspects of how/why/where/when a certificate can be used. Some extensions are added based on how the Certificate Authority is configured, e.g. CRL/OCSP URLs. Extensions that define for what purposes are a certificate can be used can be configured on a per-certificate basis.

The easiest way is to use profiles that define what extensions are added to any certificate. **django-ca** adds these predefined profiles:

Name	Purpose
client	Allows the certificate to be used on the client-side of a TLS connection.
server	Allows the certificate to be used on the client- and server-side of a connections.
enduser	Allows client authentication and code and email signing.
webserver	Allows only the server-side of a TLS connection, it can't be used as a client certificate.
ocsp	Allows the certificate to be used for signing OCSP responses.

You can add and modify profiles using the *CA_PROFILES* setting. The default profile is configured by the *CA_DEFAULT_PROFILE* setting.

Override extensions

You can override some extensions using command-line parameters. Currently, this includes `keyUsage`, `extendedKeyUsage` and `TLSFeature`. In every case, prefixing the value with `critical` marks the extension as critical (meaning a TLS client that does not understand the extension will reject the connection):

```
$ python manage.py sign_cert \
   --key-usage critical,keyCertSign \
   --ext-key-usage serverAuth,clientAuth \
   --tls-feature OCSPMustStaple \
   ...
```

For more information on these extensions, their meaning and typical values, see *x509 extensions*.

6.2.3 Revoke certificates

To revoke a certificate, use:

```
$ python manage.py list_certs
49:BC:F2:FE:FA:31:03:B6:E0:CC:3D:16:93:4E:2D:B0:8A:D2:C5:87 - localhost (expires:␣
↪2019-04-18)
...
$ python manage.py revoke_cert␣
↪49:BC:F2:FE:FA:31:03:B6:E0:CC:3D:16:93:4E:2D:B0:8A:D2:C5:87
```

6.2.4 Expiring certificates

You can add email addresses to be notified of expiring certificates using the `--watch` parameter:

```
$ python manage.py --sign-cert --watch user@example.com --watch user@example.net ...
```

Or modify to add/remove watchers later:

```
$ python manage.py list_certs
49:BC:F2:FE:FA:31:03:B6:E0:CC:3D:16:93:4E:2D:B0:8A:D2:C5:87 - localhost (expires:␣
↪2019-04-18)
...
$ python manage.py cert_watchers -a add@example.com -r user@example.net 49:BC:F2
```

Note: Consider *creating a bash script* to easily access your manage.py script.

6.3 Index of existing commands

`manage.py` subcommands for *certificate authority management*:

Command	Description
dump_ca	Write the CA certificate to a file.
edit_ca	Edit an existing certificate authority.
import_ca	Import an existing certificate authority.
init_ca	Create a new certificate authority.
list_cas	List currently configured certificate authorities.
view_ca	View details of a certificate authority.

`manage.py` subcommands for *certificate management*:

Command	Description
cert_watchers	Add/remove addresses to be notified of an expiring certificate.
dump_cert	Dump a certificate to a file.
import_cert	Import an existing certificate.
list_certs	List all certificates.
notify_expiring_certs	Send notifications about expiring certificates to watchers.
revoke_cert	Revoke a certificate.
sign_cert	Sign a certificate.
view_cert	View a certificate.

Miscellaneous `manage.py` subcommands:

Command	Description
dump_crl	Write the certificate revocation list (CRL), see *Host a Certificate Revocation List (CRL)*.
dump_ocsp_index	Write an OCSP index file, see *Run a OCSP responder*.

6.4 Names on the command-line

The most common use case for certificates is to issue certificates for domains. For example, a certificate for "example.com" is valid for exactly that domain and no other. But certificates can be valid for various other names as well, e.g. email addresses or URLs. Those names also occur in other places, like in the *Name constraints* extension for CAs.

On the command-line, **django-ca** will do its best to guess what you want. This example would issue a certificate valid for one domain and and one email address:

```
$ python manage.py sign_cert --alt example.com --alt user@example.net ...
```

If the name you're giving might be ambigious or you just want to make sure that the value is interpreted correctly, you can always use a prefix to force a particular type. This is equivalent to the above example:

```
$ python manage.py sign_cert --alt DNS:example.com --alt email:user@example.net ...
```

Valid prefixes right now are:

Prefix	Meaning
DNS	A DNS name, the most common use case.
email	An email address (e.g. used when using S/MIME to sign emails).
dirname	An LDAP-style directory name, e.g. "/C=AT/L=Vienna/CN=example.at".
URI	A URI, e.g. https://example.com.
IP	An IP address, both IPv4 and IPv6 are supported.
RID	A "Registered ID". No real-world examples are known, you're on your own.
otherName	Anything not covered in the above values. Same restrictions as for RID apply.

6.4.1 Wildcard names

In some cases you might want to use a wildcard in DNS names. The most common use cases are "wildcard certificates", which are valid for all given subdomains. Creating such certificates is simple:

```
$ python manage.py sign_cert --alt *.example.com ...
```

6.4.2 IP addresses

Both IPv4 and IPv6 addresses are supported, e.g. this certificate is valid for localhost on both IPv4 and IPv6:

```
python manage.py sign_cert --alt ::1 --alt 127.0.0.1 ...
```

CHAPTER 7

Web interface

The web interface allows you to perform the most common tasks necessary when running certificate authority. It is implemented using Djangos admin interface. You can:

- Issue and revoke certificates.
- Modify the x509 extensions used when signing certificates.
- Modify who is notified about expiring certificates.

The django project in the git repository (e.g. if you installed **django-ca** as *a standalone project*) already enables the admin interface and it's usable as soon as you enabled the webserver (tip: Create a user for login using `manage.py createsuperuser`). If you installed **django-ca** as an app, the admin interface is automatically included.

CHAPTER 8

Host a Certificate Revocation List (CRL)

A Certificate Revocation List (CRL) contains all revoked certificates signed by a certificate authority. Having a CRL is completely optional (e.g. Let's Encrypt certificates don't have one).

A URL to the CRL is usually included in the certificates (in the `crlDistributionPoints` x509 extension) so clients can fetch the CRL and verify that the certificate has not been revoked. Some services (e.g. OpenVPN) also just keep a local copy of a CRL.

Note: CRLs are usually hosted via HTTP, **not** HTTPS. CRLs are always signed, so hosting them via HTTP is not a security vulnerability. Further, you cannot verify the the certificate used when fetching the CRL anyway, since you would need the CRL for that.

8.1 Add CRL URL to new certificates

To include the URL to a CRL in newly issued certificates (you cannot add it to already issued certificates, obviously), either set it in the admin interface or via the command line:

```
$ python manage.py list_cas
34:D6:02:B5:B8:27:4F:51:9A:16:0C:B8:56:B7:79:3F - Root CA
$ python manage.py edit_ca --crl-url=http://ca.example.com/crl.pem \
>     34:D6:02:B5:B8:27:4F:51:9A:16:0C:B8:56:B7:79:3F
```

8.2 Use generic view to host a CRL

django-ca provides the generic view *`CertificateRevocationListView`* to provide CRLs via HTTP.

If you installed **django-ca** as a full project, a default CRL is already available for all CAs. If you installed django-ca on "ca.example.com", the CRL is available at `http://ca.example.com/django_ca/crl/<serial>/`. If

you installed django-ca as an app, you only need to include `django_ca.urls` in your URL conf at the appropriate location.

The default CRL is in the ASN1/DER format, signed with sha512 and refreshed every ten minutes. This is fine for TLS clients that use CRLs and is in fact similar to what public CAs use (see *crlDistributionPoints*). If you want to change any of these settings, you can override them as parameters in a URL conf:

```
from OpenSSL import crypto
from django_ca.views import CertificateRevocationListView

urlpatterns = [
    # ... your other patterns

    # We need a CRL in PEM format with a sha256 digest
    url(r'^crl/(?P<serial>[0-9A-F:]+)/$',
        CertificateRevocationListView.as_view(
            type=crypto.FILETYPE_PEM,
            digest='sha256',
            content_type='text/plain',
        ),
        name='sha256-crl')),
]
```

If you do not want to include the automatically hosted CRL, please set `CA_PROVIDE_GENERIC_CRL` to `False` in your settings.

class `django_ca.views.`**CertificateRevocationListView**(***kwargs*)

Generic view that provides Certificate Revocation Lists (CRLs).

ca_crl = False

If set to `True`, return a CRL for child CAs instead.

content_type = None

Value of the Content-Type header used in the response. For CRLs in PEM format, use `text/plain`.

digest = <cryptography.hazmat.primitives.hashes.SHA512 object>

Digest used for generating the CRL.

expires = 600

CRL expires in this many seconds.

password = None

Password used to load the private key of the certificate authority. If not set, the private key is assumed to be unencrypted.

type = 'DER'

Filetype for CRL.

8.3 Write a CRL to a file

You can generate the CRL with the `manage.py dump_crl` command:

```
$ python manage.py dump_crl -f PEM /var/www/crl.pem
```

Note: The `dump_crl` command uses the first enabled CA by default, you can force a particular CA with `--ca=<serial>`.

CRLs expire after a certain time (default: one day, configure with `--expires=SECS`), so you must periodically regenerate it, e.g. via a cron-job.

How and where to host the file is entirely up to you. If you run a Django project with a webserver already, one possibility is to dump it to your `MEDIA_ROOT` directory.

CHAPTER 9

Run a OCSP responder

OCSP, or the Online Certificate Status Protocol provides a second method (besides *CRLs*) for a client to find out if a certificate has been revoked.

9.1 Configure OCSP with django-ca

django-ca provides generic HTTP endpoints for an OCSP service for your certificate authorities. The setup involves:

1. *Creating a responder certificate*
2. *Configure generic views*
3. *Add a OCSP URL to the new certificate*

New in version 1.2: Before version 1.2, django-ca was not able to host its own OCSP responder.

9.1.1 Create an OCSP responser certificate

To run an OCSP responder, you first need a certificate with some special properties. Luckily, **django-ca** has a profile predefined for you:

```
$ openssl genrsa -out ocsp.key 4096
$ openssl req -new -key ocsp.key -out ocsp.csr -utf8 -batch
$ python manage.py sign_cert --csr=ocsp.csr --out=ocsp.pem \
>     --subject /CN=ocsp.example.com --ocsp
```

Warning: The CommonName in the certificates subject must match the domain where you host your **django-ca** installation.

9.1.2 Configure generic views

The final step in configuring an OCSP responder for the CA is configuring the HTTP endpoint. If you've installed django-ca as a full project or include `django_ca.urls` in your root URL config, configure the `CA_OCSP_URLS` setting. It's a dictionary configuring instances of *OCSPView*. Keys become part of the URL pattern, the value is a dictionary for the arguments of the view. For example:

```
CA_OCSP_URLS = {
    'Root CA': {
        'responder_key': '/usr/share/django-ca/ocsp.key',
        'responder_cert': '/usr/share/django-ca/ocsp.pem',

        # optional: The name or serial of the CA. By default, the dictionary key (
↪"Root CA" in
        #           this example is assumed to be the CA name or serial.
        #'ca': '34:D6:02:B5:B8:27:4F:51:9A:16:0C:B8:56:B7:79:3F',

        # optional: How long OCSP responses are valid
        #'expires': 3600,
    },

    # This URL can be added to any intermediate CA using the --ca-ocsp-url parameter
    'Root CA - intermediate': {
        # Dictionary key is not the name of the root CA, so we pass a serial instead:
        'ca': '34:D6:02:B5:B8:27:4F:51:9A:16:0C:B8:56:B7:79:3F',
        'responder_key': '/usr/share/django-ca/ocsp.key',
        'responder_cert': '/usr/share/django-ca/ocsp.pem',

        # optional: This URL serves OCSP responses for Child CAs, not signed enduser␣
↪certs:
        #'ca_ocsp': True,
    }
}
```

This would mean that your OCSP responder would be located at `/django_ca/ocsp/root/` at whatever domain you have configured your WSGI daemon. If you're using your own URL configuration, pass the same parameters to the `as_view()` method.

class `django_ca.views.`**OCSPView**(***kwargs*)

View to provide an OCSP responder.

See also:

This is heavily inspired by https://github.com/threema-ch/ocspresponder/blob/master/ocspresponder/__init__.py.

ca = None

The name or serial of your Certificate Authority.

ca_ocsp = False

If set to `True`, validate child CAs instead.

expires = 600

Time in seconds that the responses remain valid. The default is 600 seconds or ten minutes.

responder_cert = None

Absolute path to the public key used for signing OCSP responses. May also be a serial identifying a certificate from the database.

responder_key = None
Absolute path to the private key used for signing OCSP responses.

9.1.3 Add OCSP URL to new certificates

To include the URL to an OCSP service to newly issued certificates (you cannot add it to already issued certificates, obviously), either set it in the admin interface or via the command line:

```
$ python manage.py list_cas
34:D6:02:B5:B8:27:4F:51:9A:16:0C:B8:56:B7:79:3F - Root CA
$ python manage.py edit_ca --ocsp-url=http://ocsp.example.com/ \
>     34:D6:02:B5:B8:27:4F:51:9A:16:0C:B8:56:B7:79:3F
```

9.2 Run an OCSP responser with `openssl ocsp`

OpenSSL ships with the `openssl ocsp` command that allows you to run an OCSP responser, but note that the manpage says **"only useful for test and demonstration purposes"**.

To use the command, generate an index:

```
$ python manage.py dump_ocsp_index ocsp.index
```

OpenSSL itself allows you to run an OCSP responder with this command:

```
$ openssl ocsp -index ocsp.index -port 8888 -rsigner ocsp.pem \
>     -rkey ocsp.example.com.key -CA files/ca.crt -text
```

Python API:

CHAPTER 10

Python API

django-ca provides a Python API for everyone that wants to extend the functionality or build your own solution on top.

Note: This project is developed using Python and Django. Using the Python API requires knowledge in both. If you need help, both projects provide excellend documentation.

10.1 General

django-ca is a standard Django App. Using it requires a basic Django environment. You do not have to provide any special settings, default settings should be fine.

If you plan on using this project in standalone scripts, Django has some hints to get you started. But note that you still have to configure all of the basic Django settings and there is virtually no functionality without a database.

In some environments, e.g. where **django-ca** is exclusively used with commandline scripts, it might we worth it to use the default SQLite database backend.

10.2 Certificate Authorities

Certificate Authorities are represented by the *CertificateAuthority* model. It is a standard Django model, which means you can use the QuerySet API to retrieve and manipulate CAs:

```
>>> from django_ca.models import CertificateAuthority
>>> ca = CertificateAuthority.objects.get(serial='...')
>>> ca.enabled = False
>>> ca.save()
```

To create a new CA, you have to *init()*, this example creates a minimal CA:

```
>>> from django_ca.models import CertificateAuthority
>>> from django_ca.subject import Subject
>>> from datetime import datetime
>>> ca = CertificateAuthority.objects.init(
...     name='A new CA', subject=Subject('/CN=ca.example.com'), key_size=2048)
```

Please refer to the autogenerated documentation for *CertificateAuthority* and *CertificateAuthorityManager* for a detailed list of models.

10.3 Certificates

Certificates are represented by the *Certificate* model, they too are a standard Django model:

```
>>> from django_ca.models import Certificate
>>> cert = Certificate.objects.get(serial='...')
>>> cert.revoke()  # this already calls save()
```

Much like with certificate authorities, creating a new certificate requires a management method:

```
>>> from django_ca.models import Certificate
>>> cert = Certificate.objects.init(ca, csr, ...)
```

10.4 Subjects and Extensions

django-ca provides a set of convenience classes to allow you to handle subjects and extensions for certificates more easily. These classes are easy to instantiate and provide convenient access methods:

- *django_ca.subject.Subject* handles x509 subjects, e.g. as used in a certificate subject.
- *django_ca.extensions* is a module that includes various classes that represent x509 extensions.

10.5 Signals

Signals are a way for a developer to execute code whenever an event happens, for example to send out an email whenever a new certificate is issued. **django-ca** provides some *custom signals*.

CHAPTER 11

Signals

django-ca adds a few custom Django signals to important events to let you execute custom actions when these events happen. Please see Djangos documentation on signals for further information on how to use signals.

If you use **django-ca** as *standalone project*, use the *CA_CUSTOM_APPS* setting to add a custom django app. Please see the Django documentation on apps if you need help on writing Django apps.

`django_ca.signals.`**`post_create_ca = <django.dispatch.dispatcher.Signal object>`**
Called after a new certificate authority was created.

Parameters

ca [*CertificateAuthority*] The certificate authority that was just created.

`django_ca.signals.`**`post_issue_cert = <django.dispatch.dispatcher.Signal object>`**
Called after a new certificate was issued.

Parameters

cert [*Certificate*] The certificate that was just issued.

`django_ca.signals.`**`post_revoke_cert = <django.dispatch.dispatcher.Signal object>`**
Called after a certificate was revoked

Parameters

cert [*Certificate*] The certificate that was just revoked.

`django_ca.signals.`**`pre_create_ca = <django.dispatch.dispatcher.Signal object>`**
Called before a new certificate authority is created.

Parameters

name [str] The name of the future CA.

****kwargs**

`django_ca.signals.`**`pre_issue_cert = <django.dispatch.dispatcher.Signal object>`**
Called before a new certificate is issued.

Parameters

ca

csr

****kwargs**

django_ca.signals.**pre_revoke_cert = <django.dispatch.dispatcher.Signal object>**

Called before a certificate is revoked.

Parameters

ca

csr

****kwargs**

CHAPTER 12

django_ca.extensions - X509 extensions

class django_ca.extensions.**Extension**(*value*)

Convenience class to handle X509 Extensions.

The class is designed to take whatever format an extension might occur, essentially providing a convertible format for extensions that is used in many places throughout the code. It accepts `str` if e.g. the value was received from the commandline:

```
>>> KeyUsage('keyAgreement,keyEncipherment')
<KeyUsage: ['keyAgreement', 'keyEncipherment'], critical=False>
>>> KeyUsage('critical,keyAgreement,keyEncipherment')
<KeyUsage: ['keyAgreement', 'keyEncipherment'], critical=True>
```

It also accepts a `list`/`tuple` of two elements, the first being the "critical" flag, the second being a value (e.g. from a MultiValueField from a form):

```
>>> KeyUsage((False, ['keyAgreement', 'keyEncipherment']))
<KeyUsage: ['keyAgreement', 'keyEncipherment'], critical=False>
>>> KeyUsage((True, ['keyAgreement', 'keyEncipherment']))
<KeyUsage: ['keyAgreement', 'keyEncipherment'], critical=True>
```

Or it can be a `dict` as used by the *CA_PROFILES* setting:

```
>>> KeyUsage({'value': ['keyAgreement', 'keyEncipherment']})
<KeyUsage: ['keyAgreement', 'keyEncipherment'], critical=False>
>>> KeyUsage({'critical': True, 'value': ['keyAgreement', 'keyEncipherment']})
<KeyUsage: ['keyAgreement', 'keyEncipherment'], critical=True>
```

... and finally it can also use a subclass of `ExtensionType` from `cryptography`:

```
>>> from cryptography import x509
>>> ExtendedKeyUsage(x509.extensions.Extension(
...     oid=ExtensionOID.EXTENDED_KEY_USAGE,
...     critical=False,
...     value=x509.ExtendedKeyUsage([ExtendedKeyUsageOID.SERVER_AUTH])
```

(continues on next page)

(continued from previous page)

```
... ))
<ExtendedKeyUsage: ['serverAuth'], critical=False>
```

Parameters

value [list or tuple or dict or str or `ExtensionType`] The value of the extension, the description provides further details.

Attributes

name A human readable name of this extension.

value Raw value for this extension. The type various from subclass to subclass.

add_colons(*s*)
Add colons to a string.

TODO: duplicate from utils :-(

as_extension()
This extension as `ExtensionType`.

as_text()
Human-readable version of the *value*, not including the "critical" flag.

extension_type
The extension_type for this value.

for_builder()
Return kwargs suitable for a `CertificateBuilder`.

Example:

```
>>> kwargs = KeyUsage('keyAgreement,keyEncipherment').for_builder()
>>> builder.add_extension(**kwargs)
```

name
A human readable name of this extension.

class django_ca.extensions.**KeyIdExtension**(*value*)
Bases: *django_ca.extensions.Extension*

Base class for extensions that contain a KeyID as value.

class django_ca.extensions.**MultiValueExtension**(*value*)
Bases: *django_ca.extensions.Extension*

A generic base class for extensions that have multiple values.

Instances of this class have a `len()` and can be used with the `in` operator:

```
>>> ku = KeyUsage((False, ['keyAgreement', 'keyEncipherment']))
>>> 'keyAgreement' in ku
True
>>> len(ku)
2
```

Known values are set in the `KNOWN_VALUES` attribute for each class. The constructor will raise `ValueError` if an unknown value is passed.

12.1 Concrete extensions

class django_ca.extensions.**AuthorityKeyIdentifier**(*value*)
Bases: *django_ca.extensions.KeyIdExtension*

Class representing a AuthorityKeyIdentifier extension.

class django_ca.extensions.**ExtendedKeyUsage**(*value*)
Bases: *django_ca.extensions.MultiValueExtension*

Class representing a ExtendedKeyUsage extension.

KNOWN_VALUES = {'OCSPSigning', 'clientAuth', 'codeSigning', 'emailProtection', 'msKDC'
Known values for this extension.

class django_ca.extensions.**KeyUsage**(**args*, ***kwargs*)
Bases: *django_ca.extensions.MultiValueExtension*

Class representing a KeyUsage extension.

KNOWN_VALUES = {'cRLSign', 'dataEncipherment', 'decipherOnly', 'digitalSignature', 'en
Known values for this extension.

class django_ca.extensions.**SubjectKeyIdentifier**(*value*)
Bases: *django_ca.extensions.KeyIdExtension*

Class representing a SubjectKeyIdentifier extension.

class django_ca.extensions.**TLSFeature**(*value*)
Bases: *django_ca.extensions.MultiValueExtension*

Class representing a TLSFeature extension.

KNOWN_VALUES = {'MultipleCertStatusRequest', 'OCSPMustStaple'}
Known values for this extension.

CHAPTER 13

django_ca.models - django-ca models

Note that both *CertificateAuthority* and *Certificate* inherit from *X509CertMixin*, which provides many convenience methods.

13.1 CertificateAuthority

class django_ca.models.**CertificateAuthority**(*id*, *created*, *expires*, *pub*, *cn*, *serial*, *revoked*, *revoked_date*, *revoked_reason*, *name*, *enabled*, *parent*, *private_key_path*, *crl_url*, *issuer_url*, *ocsp_url*, *issuer_alt_name*)

allows_intermediate_ca
Wether this CA allows creating intermediate CAs.

bundle
A list of any parent CAs, including this CA.

The list is ordered so the Root CA will be the first.

max_pathlen
The maximum pathlen for any intermediate CAs signed by this CA.

This value is either `None`, if this and all parent CAs don't have a `pathlen` attribute, or an `int` if any parent CA has the attribute.

name
Human-readable name of the CA, only used for displaying the CA.

pathlen
The `pathlen` attribute of the `BasicConstraints` extension (either an `int` or `None`).

13.1.1 Manager methods

CertificateAuthorityManager is the default manager for *CertificateAuthority*, meaning you can

access it using `CertificateAuthority.objects`, e.g.:

```
>>> from django_ca.models import CertificateAuthority
>>> CertificateAuthority.objects.init(...)
```

class django_ca.managers.**CertificateAuthorityManager**

init(*name*, *subject*, *expires=None*, *algorithm=None*, *parent=None*, *pathlen=None*, *issuer_url=None*, *issuer_alt_name=None*, *crl_url=None*, *ocsp_url=None*, *ca_issuer_url=None*, *ca_crl_url=None*, *ca_ocsp_url=None*, *name_constraints=None*, *password=None*, *parent_password=None*, *ecc_curve=None*, *key_type='RSA'*, *key_size=None*)

Create a new certificate authority.

Parameters

name [str] The name of the CA. This can be a human-readable string and is used for administrative purposes only.

algorithm [str or `HashAlgorithm`, optional] Hash algorithm used when signing the certificate. If a string is passed, it must be the name of one of the hashes in `hashes`, e.g. `"SHA512"`. This method also accepts instances of `HashAlgorithm`, e.g. `SHA512`. The default is the `CA_DIGEST_ALGORITHM` setting.

subject [*Subject*] Subject string, e.g. `Subject("/CN=example.com")`.

expires [datetime, optional] Datetime for when this certificate authority will expire, defaults to the `CA_DEFAULT_EXPIRES` setting.

parent [*CertificateAuthority*, optional] Parent certificate authority for the new CA. This means that this CA will be an intermediate authority.

pathlen [int, optional]

password [bytes, optional] Password to encrypt the private key with.

parent_password [bytes, optional] Password that the private key of the parent CA is encrypted with.

ecc_curve [str or EllipticCurve, optional] The elliptic curve to use for ECC type keys, passed verbatim to *parse_key_curve()*.

key_type: str, optional The type of private key to generate, must be one of `"RSA"`, `"DSA"` or `"ECC"`, with `"RSA"` being the default.

key_size [int, optional] Integer specifying the key size, must be a power of two (e.g. 2048, 4096, ...) unused if `key_type="ECC"` but required otherwise.

Raises

ValueError For various cases of wrong input data (e.g. `key_size` not being the power of two).

PermissionError If the private key file cannot be written to disk.

13.2 Certificate

class django_ca.models.**Certificate**(*id*, *created*, *expires*, *pub*, *cn*, *serial*, *revoked*, *revoked_date*, *revoked_reason*, *ca*, *csr*)

bundle
The complete certificate bundle. This includes all CAs as well as the certificates itself.

13.2.1 Manager methods

CertificateManager is the default manager for *Certificate*, meaning you can access it using `Certificate.objects`, e.g.:

```
>>> from django_ca.models import Certificate
>>> Certificate.objects.init(...)
```

class django_ca.managers.**CertificateManager**

init(*ca*, *csr*, ***kwargs*)
Create a signed certificate from a CSR and store it to the database.

All parameters are passed on to *Certificate.objects.sign_cert()*.

sign_cert(*ca*, *csr*, *expires=None*, *algorithm=None*, *subject=None*, *cn_in_san=True*, *csr_format=<Encoding.PEM: 'PEM'>*, *subjectAltName=None*, *key_usage=None*, *extended_key_usage=None*, *tls_feature=None*, *password=None*)
Create a signed certificate from a CSR.

PLEASE NOTE: This function creates the raw certificate and is usually not invoked directly. It is called by *Certificate.objects.init()*, which passes along all parameters unchanged and saves the raw certificate to the database.

Parameters

ca [*CertificateAuthority*] The certificate authority to sign the certificate with.

csr [str] A valid CSR. The format is given by the `csr_format` parameter.

expires [datetime, optional] Datetime for when this certificate will expire, defaults to the `CA_DEFAULT_EXPIRES` setting.

algorithm [str or `HashAlgorithm`, optional] Hash algorithm used when signing the certificate. If a string is passed, it must be the name of one of the hashes in `hashes`, e.g. `"SHA512"`. This method also accepts instances of `HashAlgorithm`, e.g. `SHA512`. The default is the `CA_DIGEST_ALGORITHM` setting.

subject [*Subject*, optional] The Subject to use in the certificate. If this value is not passed or if the value does not contain a CommonName, the first value of the `subjectAltName` parameter is used as CommonName.

cn_in_san [bool, optional] Wether the CommonName should also be included as subjectAlternativeName. The default is `True`, but the parameter is ignored if no CommonName is given. This is typically set to `False` when creating a client certificate, where the subjects CommonName has no meaningful value as subjectAltName.

csr_format [`Encoding`, optional] The format of the CSR. The default is `PEM`.

subjectAltName [list of str, optional] A list of values for the subjectAltName extension. Values are passed to *parse_general_name()*, see function documentation for how this value is parsed.

key_usage [*KeyUsage*, optional] Value for the `keyUsage` X509 extension.

extended_key_usage [*ExtendedKeyUsage*, optional] Value for the `extendedKeyUsage` X509 extension.

tls_feature [*TLSFeature*, optional] Value for the `TLSFeature` X509 extension.

password [bytes, optional] Password used to load the private key of the certificate authority. If not passed, the private key is assumed to be unencrypted.

Returns

cryptography.x509.Certificate The signed certificate.

13.3 X509CertMixin

X509CertMixin is a common base class to both *CertificateAuthority* and *Certificate* and provides many convenience attributes.

class `django_ca.models.`**X509CertMixin**(**args*, ***kwargs*)

authority_key_identifier
The *AuthorityKeyIdentifier* extension, or `None` if it doesn't exist.

extended_key_usage
The *ExtendedKeyUsage* extension, or `None` if it doesn't exist.

issuer
The certificate issuer field as *Subject*.

key_usage
The *KeyUsage* extension, or `None` if it doesn't exist.

not_after
Date/Time this certificate expires.

not_before
Date/Time this certificate was created

subject
The certificates subject as *Subject*.

subject_key_identifier
The *SubjectKeyIdentifier* extension, or `None` if it doesn't exist.

tls_feature
The *TLSFeature* extension, or `None` if it doesn't exist.

x509
The underlying `cryptography.x509.Certificate`.

CHAPTER 14

django_ca.subject - X509 Subject

class django_ca.subject.**Subject**(*subject=None*)

Convenience class to handle X509 Subjects.

This class accepts a variety of values and intelligently parses them:

```
>>> Subject('/CN=example.com')
Subject("/CN=example.com")
>>> Subject({'CN': 'example.com'})
Subject("/CN=example.com")
>>> Subject([('CN', 'example.com'), ])
Subject("/CN=example.com")
```

In many respects, this class handles like a `dict`:

```
>>> s = Subject('/CN=example.com')
>>> 'CN' in s
True
>>> s.get('OU', 'Default OU')
'Default OU'
>>> s.setdefault('C', 'AT')
>>> s['C'], s['CN']
('AT', 'example.com')
```

fields

This subject as a list of `NameOID` instances.

```
>>> list(Subject('/C=AT/CN=example.com').fields)  # doctest: +NORMALIZE_
↪WHITESPACE
[(<ObjectIdentifier(oid=2.5.4.6, name=countryName)>, 'AT'),
 (<ObjectIdentifier(oid=2.5.4.3, name=commonName)>, 'example.com')]
```

name

This subject as `x509.Name`.

```
>>> Subject('/C=AT/CN=example.com').name  # doctest: +NORMALIZE_WHITESPACE
<Name([<NameAttribute(oid=<ObjectIdentifier(oid=2.5.4.6, name=countryName)>,␣
↪value='AT')>,
       <NameAttribute(oid=<ObjectIdentifier(oid=2.5.4.3, name=commonName)>,␣
↪value='example.com')>])>
```

CHAPTER 15

django_ca.utils - utility functions

Central functions to load CA key and cert as PKey/X509 objects.

django_ca.utils.**GENERAL_NAME_RE = re.compile('^(email|URI|IP|DNS|RID|dirName|otherName):(.**
Regular expression to match general names.

class django_ca.utils.**LazyEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)
Encoder that also encodes strings translated with ugettext_lazy.

django_ca.utils.**NAME_RE = re.compile('(?:/+|\\A)\\s*(?P<field>[^\\s]*?)\\s*=(?P<quote>[**
Regular expression to match RDNs out of a full x509 name.

django_ca.utils.**OID_NAME_MAPPINGS = {<ObjectIdentifier(oid=2.5.4.7, name=localityName)>:**
Map OID objects to IDs used in subject strings

django_ca.utils.**add_colons**(*s*)
Add colons after every second digit.

This function is used in functions to prettify serials.

```
>>> add_colons('teststring')
'te:st:st:ri:ng'
```

django_ca.utils.**format_general_name**(*name*)
Format a single general name.

```
>>> import ipaddress
>>> format_general_name(x509.DNSName('example.com'))
'DNS:example.com'
>>> format_general_name(x509.IPAddress(ipaddress.IPv4Address('127.0.0.1')))
'IP:127.0.0.1'
```

django_ca.utils.**format_general_names**(*names*)
Format a list of general names.

```
>>> import ipaddress
>>> format_general_names([x509.DNSName('example.com')])
'DNS:example.com'
>>> format_general_names([x509.IPAddress(ipaddress.IPv4Address('127.0.0.1'))])
'IP:127.0.0.1'
>>> format_general_names([x509.DirectoryName(
...     x509.Name([x509.NameAttribute(x509.oid.NameOID.COMMON_NAME, 'example.com
↪')]))])
'dirname:/CN=example.com'
>>> format_general_names([x509.DNSName('example.com'), x509.DNSName('example.net
↪')])
'DNS:example.com, DNS:example.net'
```

django_ca.utils.**format_name**(*subject*)
Convert a subject into the canonical form for distinguished names.

This function does not take care of sorting the subject in any meaningful order.

Examples:

```
>>> format_name([('CN', 'example.com'), ])
'/CN=example.com'
>>> format_name([('CN', 'example.com'), ('O', "My Organization"), ])
'/CN=example.com/O=My Organization'
```

django_ca.utils.**get_cert_builder**(*expires*)
Get a basic X509 cert builder object.

Parameters

expires [datetime] When this certificate will expire.

django_ca.utils.**get_cert_profile_kwargs**(*name=None*)
Get kwargs suitable for get_cert X509 keyword arguments from the given profile.

django_ca.utils.**get_default_subject**(*name*)
Get the default subject for the given profile.

django_ca.utils.**int_to_hex**(*i*)
Create a hex-representation of the given serial.

```
>>> int_to_hex(12345678)
'BC:61:4E'
```

django_ca.utils.**is_power2**(*num*)
Return True if num is a power of 2.

```
>>> is_power2(4)
True
>>> is_power2(3)
False
```

django_ca.utils.**multiline_url_validator**(*value*)
Validate that a TextField contains one valid URL per line.

See also:

https://docs.djangoproject.com/en/1.9/ref/validators/

django_ca.utils.**parse_general_name**(*name*)
Parse a general name from user input.

This function will do its best to detect the intended type of any value passed to it:

```
>>> parse_general_name('example.com')
<DNSName(value='example.com')>
>>> parse_general_name('*.example.com')
<DNSName(value='*.example.com')>
>>> parse_general_name('.example.com')  # Syntax used e.g. for NameConstraints:␣
↪All levels of subdomains
<DNSName(value='.example.com')>
>>> parse_general_name('user@example.com')
<RFC822Name(value='user@example.com')>
>>> parse_general_name('https://example.com')
<UniformResourceIdentifier(value='https://example.com')>
>>> parse_general_name('1.2.3.4')
<IPAddress(value=1.2.3.4)>
>>> parse_general_name('fd00::1')
<IPAddress(value=fd00::1)>
>>> parse_general_name('/CN=example.com')  # doctest: +NORMALIZE_WHITESPACE
<DirectoryName(value=<Name([<NameAttribute(oid=<ObjectIdentifier(oid=2.5.4.3,␣
↪name=commonName)>,
                                            value='example.com')>])>)>
```

The default fallback is to assume a `DNSName`. If this doesn't work, an exception will be raised:

```
>>> parse_general_name('foo..bar`*123')
Traceback (most recent call last):
    ...
idna.core.IDNAError: The label b'' is not a valid A-label
>>> parse_general_name('foo bar')
Traceback (most recent call last):
    ...
idna.core.IDNAError: The label b'foo bar' is not a valid A-label
```

If you want to override detection, you can prefix the name to match *GENERAL_NAME_RE*:

```
>>> parse_general_name('email:user@example.com')
<RFC822Name(value='user@example.com')>
>>> parse_general_name('URI:https://example.com')
<UniformResourceIdentifier(value='https://example.com')>
>>> parse_general_name('dirname:/CN=example.com')  # doctest: +NORMALIZE_
↪WHITESPACE
<DirectoryName(value=<Name([<NameAttribute(oid=<ObjectIdentifier(oid=2.5.4.3,␣
↪name=commonName)>,
                                            value='example.com')>])>)>
```

Some more exotic values can only be generated by using this prefix:

```
>>> parse_general_name('rid:2.5.4.3')
<RegisteredID(value=<ObjectIdentifier(oid=2.5.4.3, name=commonName)>)>
>>> parse_general_name('otherName:2.5.4.3;UTF8:example.com')
<OtherName(type_id=<ObjectIdentifier(oid=2.5.4.3, name=commonName)>, value=b
↪'example.com')>
```

If you give a prefixed value, this function is less forgiving of any typos and does not catch any exceptions:

```
>>> parse_general_name('email:foo@bar com')
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
    ...
ValueError: Invalid domain: bar com
```

django_ca.utils.**parse_hash_algorithm**(*value=None*)

Parse a hash algorithm value.

The most common use case is to pass a str naming a class in `hashes`.

For convenience, passing `None` will return the value of *CA_DIGEST_ALGORITHM*, and passing an `HashAlgorithm` will return that instance unchanged.

Example usage:

```
>>> parse_hash_algorithm()
<cryptography.hazmat.primitives.hashes.SHA512 object at ...>
>>> parse_hash_algorithm('SHA512')
<cryptography.hazmat.primitives.hashes.SHA512 object at ...>
>>> parse_hash_algorithm(' SHA512 ')
<cryptography.hazmat.primitives.hashes.SHA512 object at ...>
>>> parse_hash_algorithm(hashes.SHA512)
<cryptography.hazmat.primitives.hashes.SHA512 object at ...>
>>> parse_hash_algorithm(hashes.SHA512())
<cryptography.hazmat.primitives.hashes.SHA512 object at ...>
>>> parse_hash_algorithm('Wrong')
Traceback (most recent call last):
    ...
ValueError: Unknown hash algorithm: Wrong
>>> parse_hash_algorithm(object())
Traceback (most recent call last):
    ...
ValueError: Unknown type passed: object
```

Parameters

value [str or `HashAlgorithm`, optional] The value to parse, the function description on how possible values are used.

Returns

algorithm A `HashAlgorithm` instance.

Raises

ValueError If an unknown object is passed or if `value` does not name a known algorithm.

django_ca.utils.**parse_key_curve**(*value=None*)

Parse an elliptic curve value.

This function uses a value identifying an elliptic curve to return an `EllipticCurve` instance. The name must match a class name of one of the classes named under "Elliptic Curves" in `Elliptic curve cryptography`.

For convenience, passing `None` will return the value of *CA_DEFAULT_ECC_CURVE*, and passing an `EllipticCurve` will return that instance unchanged.

Example usage:

```
>>> parse_key_curve('SECP256R1')
<cryptography.hazmat.primitives.asymmetric.ec.SECP256R1 object at ...>
```

(continues on next page)

(continued from previous page)

```
>>> parse_key_curve('SECP384R1')
<cryptography.hazmat.primitives.asymmetric.ec.SECP384R1 object at ...>
>>> parse_key_curve(ec.SECP256R1())
<cryptography.hazmat.primitives.asymmetric.ec.SECP256R1 object at ...>
>>> parse_key_curve()
<cryptography.hazmat.primitives.asymmetric.ec.SECP256R1 object at ...>
```

Parameters

value [str, otional] The name of the curve or `None` to return the default curve.

Returns

curve An `EllipticCurve` instance.

Raises

ValueError If the named curve is not supported.

`django_ca.utils.`**`parse_name`**(*name*)

Parses a subject string as used in OpenSSLs command line utilities.

The `name` is expected to be close to the subject format commonly used by OpenSSL, for example `/C=AT/L=Vienna/CN=example.com/emailAddress=user@example.com`. The function does its best to be lenient on deviations from the format, object identifiers are case-insensitive (e.g. `cn` is the same as `CN`, whitespace at the start and end is stripped and the subject does not have to start with a slash (`/`).

```
>>> parse_name('/CN=example.com')
[('CN', 'example.com')]
>>> parse_name('c=AT/l= Vienna/o="ex org"/CN=example.com')
[('C', 'AT'), ('L', 'Vienna'), ('O', 'ex org'), ('CN', 'example.com')]
```

Dictionary keys are normalized to the values of *OID_NAME_MAPPINGS* and keys will be sorted based on x509 name specifications regardless of the given order:

```
>>> parse_name('L="Vienna / District"/EMAILaddress=user@example.com')
[('L', 'Vienna / District'), ('emailAddress', 'user@example.com')]
>>> parse_name('/C=AT/CN=example.com') == parse_name('/CN=example.com/C=AT')
True
```

Due to the magic of *NAME_RE*, the function even supports quoting strings and including slashes, so strings like `/OU="Org / Org Unit"/CN=example.com` will work as expected.

```
>>> parse_name('L="Vienna / District"/CN=example.com')
[('L', 'Vienna / District'), ('CN', 'example.com')]
```

But note that it's still easy to trick this function, if you really want to. The following example is *not* a valid subject, the location is just bogus, and whatever you were expecting as output, it's certainly different:

```
>>> parse_name('L="Vienna " District"/CN=example.com')
[('L', 'Vienna'), ('CN', 'example.com')]
```

Examples of where this string is used are:

```
# openssl req -new -key priv.key -out csr -utf8 -batch -sha256 -subj '/C=AT/
↪CN=example.com'
# openssl x509 -in cert.pem -noout -subject -nameopt compat
/C=AT/L=Vienna/CN=example.com
```

django_ca.utils.**sort_name**(*subject*)
Returns the subject in the correct order for a x509 subject.

django_ca.utils.**validate_email**(*addr*)
Validate an email address.

This function raises `ValueError` if the email address is not valid.

```
>>> validate_email('foo@bar.com')
'foo@bar.com'
>>> validate_email('foo@bar com')
Traceback (most recent call last):
    ...
ValueError: Invalid domain: bar com
```

django_ca.utils.**write_private_file**(*path*, *data*)
Function to write binary data to a file that will only be readable to the user.

django_ca.utils.**x509_name**(*name*)
Parses a subject into a `x509.Name`.

If `name` is a string, *`parse_name()`* is used to parse it.

```
>>> x509_name('/C=AT/CN=example.com')  # doctest: +NORMALIZE_WHITESPACE
<Name([<NameAttribute(oid=<ObjectIdentifier(oid=2.5.4.6, name=countryName)>,␣
↪value='AT')>,
       <NameAttribute(oid=<ObjectIdentifier(oid=2.5.4.3, name=commonName)>, value=
↪'example.com')>])>
>>> x509_name([('C', 'AT'), ('CN', 'example.com')])  # doctest: +NORMALIZE_
↪WHITESPACE
<Name([<NameAttribute(oid=<ObjectIdentifier(oid=2.5.4.6, name=countryName)>,␣
↪value='AT')>,
       <NameAttribute(oid=<ObjectIdentifier(oid=2.5.4.3, name=commonName)>, value=
↪'example.com')>])>
```

Development documentation:

CHAPTER 16

Development

16.1 Setup demo

You can set up a demo using `fab init_demo`. First create a minimal `localsettings.py` file (in `ca/ca/localsettings.py`):

```
DEBUG = True
SECRET_KEY = "whatever"
```

And then simply run `fab init_demo` from the root directory of your project.

16.1.1 Development webserver via SSL

To test a certificate in your webserver, first install the root certificate authority in your browser, then run `stunnel4` and `manage.py runserver` in two separate shells:

```
$ stunnel4 .stunnel4.conf
```

There is also a second config file using a revoked certificate. If you use it, browsers will display an error.

```
$ stunnel4  .stunnel4-revoked.conf
```

You can now start your development webserver normally:

```
$ DJANGO_SETTINGS_MODULE=ca.demosettings python manage.py runserver
```

. . . and visit https://localhost:8443.

16.2 Run test-suite

To run the test-suite, simply execute:

```
python setup.py test
```

... or just run some of the tests:

```
python setup.py test --suite=tests_command_dump_crl
```

To generate a coverate report:

```
python setup.py coverage
```

16.3 Useful OpenSSL commands

16.3.1 Verification

Verify a certificate signed by a root CA (`cert.crt` could also be an intermediate CA):

```
openssl verify -CAfile ca.crt cert.crt
```

If you have an intermediate CA:

```
openssl verify -CAfile ca.crt -untrusted intermediate.crt cert.crt
```

16.3.2 CRLs

Convert a CRL to text on stdout:

```
openssl crl -inform der -in sfsca.crl -noout -text
```

Convert a CRL to PEM to a file:

```
openssl crl -inform der -in sfsca.crl -outform pem -out test.pem
```

Verify a certificate using a CRL:

```
openssl verify -CAfile files/ca_crl.pem -crl_check cert.pem
```

16.3.3 OCSP

Run a OCSP responder:

```
openssl ocsp -index files/ocsp_index.txt -port 8888 \
   -rsigner files/localhost.pem -rkey files/localhost.key \
   -CA ca.pem -text
```

Verify a certificate using OCSP:

```
openssl ocsp -CAfile ca.pem -issuer ca.pem -cert cert.pem \
    -url http://localhost:8888 -resp_text
```

16.3.4 Conversion

Convert a PEM formatted public key to DER:

```
openssl x509 -in pub.pem -outform der -out pub.der
```

Convert a PEM formatted **private** key to DER:

```
openssl rsa -in priv.pem -outform der -out priv.der
```

Convert a p7c/pkcs7 file to PEM (Let's Encrypt CA Issuer field) (see also *pkcs7(1SSL)* - online):

```
openssl pkcs7 -inform der -in letsencrypt.p7c -print_certs \
   -outform pem -out letsencrypt.pem
```

CHAPTER 17

Contribute

Please also see *Development* for how to setup a development environment.

To contribute to **django-ca** simply do a fork on on github and submit a pull request when you're happy.

When doing a pull request, please make sure to explain what your improvement does or what bug is fixed by it and how to reproduce this locally.

17.1 Code quality

This project is very rigorous about code quality standards. That means that the source code is checked with Flake8 and import order is checked with isort. Before you submit a pull request, please make sure that all tests pass by executing:

```
python setup.py code_quality
```

Naturally, I also expect the test suite to still pass. Please make sure you test in at least your local Python2 and Python3 environments:

```
python setup.py test
```

17.2 Write tests

Please write tests for any new functionality. If you provide a bugfix, write a test that tests the fix, which means that the test should fail on current master and pass on your pull request.

If a function is also covered with doctests, please consider adding an example there as well, if it affects handling a parameter or something.

17.3 Code coverage

Generate a coverage report and make sure that your code is covered by tests.

Warning: Code coverage is not a catch all tool for "yes, this code is well-tested". It's a tool to catch missed spots, but you must still think for yourself about what and how to test.

CHAPTER 18

Release process

18.1 Before release

- Update `requirements*.txt` (use `pip list -o`).
- Make sure that `setup.py` has proper requirements.
- Check `.travis.yaml` if the proper Django and cryptography versions are tested.
- Check test coverage (`setup.py coverage`).
- Update `version` parameter in `setup.py`.
- Update `version` and `release` in `docs/source/conf.py`.
- Make sure that `docs/source/changelog.rst` is up to date.
- Push the last commit and make sure that Travis and Read The Docs are updated.

18.2 Docker image

Create a docker image:

```
docker build --no-cache -t django-ca .
docker run -d --name=django-ca -p 8000:8000 django-ca
docker exec -it django-ca python ca/manage.py createsuperuser
docker exec -it django-ca python ca/manage.py init_ca \
   example /C=AT/ST=Vienna/L=Vienna/O=Org/CN=ca.example.com
```

... and browse http://localhost:8000/admin.

18.3 Release process

- Tag the release: `git tag -s $version`
- Push the tag: `git push origin --tags`
- Create a release on GitHub.
- Upload release to PyPI: `python setup.py sdist bdist_wheel upload`.
- Tag and upload the docker image (note that we create a image revision by appending `-1`):

```
docker tag django-ca mathiasertl/django-ca
docker tag django-ca mathiasertl/django-ca:$version-1
docker push mathiasertl/django-ca
docker push mathiasertl/django-ca:$version-1
```

CHAPTER 19

x509 extensions in other CAs

This page documents the x509 extensions (e.g. for CRLs, etc.) set by other CAs. The information here is used by **django-ca** to initialize and sign certificate authorities and certificates.

Helpful descriptions of the meaning of various extensions can also be found in *x509v3_config(5SSL)* (online).

19.1 CommonName

Of course not an extension, but included here for completeness.

CA	Value
Let's Encrypt X1	C=US, O=Let's Encrypt, CN=Let's Encrypt Authority X1
Let's Encrypt X3	C=US, O=Let's Encrypt, CN=Let's Encrypt Authority X3
StartSSL	C=IL, O=StartCom Ltd., OU=Secure Digital Certificate Signing, CN=StartCom Certification Authority
StartSSL Class 2	C=IL, O=StartCom Ltd., OU=Secure Digital Certificate Signing, CN=StartCom Class 2 Primary Intermediate Server CA
StartSSL Class 3	C=IL, O=StartCom Ltd., OU=StartCom Certification Authority, CN=StartCom Class 3 OV Server CA
GeoTrust Global	C=US, O=GeoTrust Inc., CN=GeoTrust Global CA
RapidSSL G3	C=US, O=GeoTrust Inc., CN=RapidSSL SHA256 CA - G3
Comodo	C=GB, ST=Greater Manchester, L=Salford, O=COMODO CA Limited, CN=COMODO RSA Certification Authority
Comodo DV	C=GB, ST=Greater Manchester, L=Salford, O=COMODO CA Limited, CN=COMODO RSA Domain Validation Secure Server CA
GlobalSign	C=BE, O=GlobalSign nv-sa, OU=Root CA, CN=GlobalSign Root CA
GlobalSign DV	C=BE, O=GlobalSign nv-sa, CN=GlobalSign Domain Validation CA - SHA256 - G2

19.2 authorityInfoAccess

See also:

https://tools.ietf.org/html/rfc5280#section-4.2.2.1

The "CA Issuers" is a URI pointing to the signing certificate. The certificate is in DER/ASN1 format and has a `Content-Type: application/x-x509-ca-cert` header (except where noted).

19.2.1 In CA certificates

Let's Encrypt is notable here because its CA Issuers field points to a pkcs7 file and the HTTP response returns a `Content-Type: application/x-pkcs7-mime` header.

The certificate pointed to by the CA Issuers field is the root certificate (so the Comodo DV CA points to the AddTrust CA that signed the Comodo Root CA).

CA	Value
Let's Encrypt X1	• OCSP - URI:http://isrg.trustid.ocsp.identrust.com • CA Issuers - URI:http://apps.identrust.com/roots/dstrootcax3.p7c
Let's Encrypt X3	• OCSP - URI:http://isrg.trustid.ocsp.identrust.com • CA Issuers - URI:http://apps.identrust.com/roots/dstrootcax3.p7c
StartSSL	(not present)
StartSSL Class 2	• OCSP - URI:http://ocsp.startssl.com/ca • CA Issuers - URI:http://aia.startssl.com/certs/ca.crt
StartSSL Class 3	• OCSP - URI:http://ocsp.startssl.com • CA Issuers - URI:http://aia.startssl.com/certs/ca.crt
GeoTrust Global	(not present)
RapidSSL G3	OCSP - URI:http://g.symcd.com
Comodo	OCSP - URI:http://ocsp.usertrust.com
Comodo DV	• CA Issuers - URI:http://crt.comodoca.com/COMODORSAAddTrust • OCSP - URI:http://ocsp.comodoca.com
GlobalSign	(not present)
GlobalSign DV	OCSP - URI:http://ocsp.globalsign.com/rootr1

19.2.2 In signed certificates

Let's Encrypt is again special in that the response has a `Content-Type: application/pkix-cert` header (but at least it's in DER format like every other certificate). RapidSSL uses `Content-Type: text/plain`.

The CA Issuers field sometimes points to the signing certificate (e.g. StartSSL) or to the root CA (e.g. Comodo DV, which points to the AddTrust Root CA)

CA	Value
Let's Encrypt X1	• OCSP - URI:http://ocsp.int-x1.letsencrypt.org/ • CA Issuers - URI:http://cert.int-x1.letsencrypt.org
Let's Encrypt X3	• OCSP - URI:http://ocsp.int-x3.letsencrypt.org/ • CA Issuers - URI:http://cert.int-x3.letsencrypt.org/
StartSSL Class 2	• OCSP - URI:http://ocsp.startssl.com/sub/class2/server/ca • CA Issuers - URI:http://aia.startssl.com/certs/sub.class2.server.ca.crt
StartSSL Class 3	• OCSP - URI:http://ocsp.startssl.com • CA Issuers - URI:http://aia.startssl.com/certs/sca.server3.crt
RapidSSL G3	• OCSP - URI:http://gv.symcd.com • CA Issuers - URI:http://gv.symcb.com/gv.crt
Comodo DV	• CA Issuers - URI:http://crt.comodoca.com/COMODORSADomainVa • OCSP - URI:http://ocsp.comodoca.com
GlobalSign DV	• CA Issuers - URI:http://secure.globalsign.com/cacert/gsdomainvalsha • OCSP - URI:http://ocsp2.globalsign.com/gsdomainvalsha2g2

19.3 authorityKeyIdentifier

See also:

https://tools.ietf.org/html/rfc5280#section-4.2.1.1

A hash identifying the CA used to sign the certificate. In theory the identifier may also be based on the issuer name and serial number, but in the wild, all certificates reference the *subjectKeyIdentifier*. Self-signed certificates (e.g. Root CAs, like StartSSL and Comodo below) will reference themself, while signed certificates reference the signed CA, e.g.:

Name	subjectKeyIdentifier	authorityKeyIdentifier
Root CA	foo	keyid:foo
Intermediate CA	bar	keyid:foo
Client Cert	bla	keyid:bar

19.3.1 In CA certificates

CA	Value
Let's Encrypt X1	keyid:C4:A7:B1:A4:7B:2C:71:FA:DB:E1:4B:90:75:FF:C4:15:60:85:89:10
Let's Encrypt X3	keyid:C4:A7:B1:A4:7B:2C:71:FA:DB:E1:4B:90:75:FF:C4:15:60:85:89:10
StartSSL	keyid:4E:0B:EF:1A:A4:40:5B:A5:17:69:87:30:CA:34:68:43:D0:41:AE:F2
StartSSL Class 2	keyid:4E:0B:EF:1A:A4:40:5B:A5:17:69:87:30:CA:34:68:43:D0:41:AE:F2
StartSSL Class 3	keyid:4E:0B:EF:1A:A4:40:5B:A5:17:69:87:30:CA:34:68:43:D0:41:AE:F2
GeoTrust Global	keyid:C0:7A:98:68:8D:89:FB:AB:05:64:0C:11:7D:AA:7D:65:B8:CA:CC:4E
RapidSSL G3	keyid:C0:7A:98:68:8D:89:FB:AB:05:64:0C:11:7D:AA:7D:65:B8:CA:CC:4E
Comodo	keyid:AD:BD:98:7A:34:B4:26:F7:FA:C4:26:54:EF:03:BD:E0:24:CB:54:1A
Comodo DV	keyid:BB:AF:7E:02:3D:FA:A6:F1:3C:84:8E:AD:EE:38:98:EC:D9:32:32:D4
GlobalSign	(not present)
GlobalSign DV	keyid:60:7B:66:1A:45:0D:97:CA:89:50:2F:7D:04:CD:34:A8:FF:FC:FD:4B

19.3.2 In signed certificates

CA	Value
Let's Encrypt X1	keyid:A8:4A:6A:63:04:7D:DD:BA:E6:D1:39:B7:A6:45:65:EF:F3:A8:EC:A1
Let's Encrypt X3	keyid:A8:4A:6A:63:04:7D:DD:BA:E6:D1:39:B7:A6:45:65:EF:F3:A8:EC:A1
StartSSL Class 2	keyid:11:DB:23:45:FD:54:CC:6A:71:6F:84:8A:03:D7:BE:F7:01:2F:26:86
StartSSL Class 3	keyid:B1:3F:1C:92:7B:92:B0:5A:25:B3:38:FB:9C:07:A4:26:50:32:E3:51
RapidSSL G3	keyid:C3:9C:F3:FC:D3:46:08:34:BB:CE:46:7F:A0:7C:5B:F3:E2:08:CB:59
Comodo DV	keyid:90:AF:6A:3A:94:5A:0B:D8:90:EA:12:56:73:DF:43:B4:3A:28:DA:E7
GlobalSign DV	keyid:EA:4E:7C:D4:80:2D:E5:15:81:86:26:8C:82:6D:C0:98:A4:CF:97:0F

19.4 basicConstraints

See also:

https://tools.ietf.org/html/rfc5280#section-4.2.1.9

The `basicConstraints` extension specifies if the certificate can be used as a certificate authority. It is always marked as critical. The `pathlen` attribute specifies the levels of possible intermediate CAs. If not present, the level of intermediate CAs is unlimited, a `pathlen:0` means that the CA itself can not issue certificates with `CA:TRUE` itself.

19.4.1 In CA certificates

CA	Value
Let's Encrypt X1	(critical) CA:TRUE, pathlen:0
Let's Encrypt X3	(critical) CA:TRUE, pathlen:0
StartSSL	(critical) CA:TRUE
StartSSL Class 2	(critical) CA:TRUE, pathlen:0
StartSSL Class 3	(critical) CA:TRUE, pathlen:0
GeoTrust Global	(critical) CA:TRUE
RapidSSL G3	(critical) CA:TRUE, pathlen:0
Comodo	(critical) CA:TRUE
Comodo DV	(critical) CA:TRUE, pathlen:0
GlobalSign	(critical) CA:TRUE
GlobalSign DV	(critical) CA:TRUE, pathlen:0

19.4.2 In signed certificates

CA	Value
Let's Encrypt X1	(critical) CA:FALSE
Let's Encrypt X3	(critical) CA:FALSE
StartSSL Class 2	(critical) CA:FALSE
StartSSL Class 3	CA:FALSE
RapidSSL G3	(critical) CA:FALSE
Comodo DV	(critical) CA:FALSE
GlobalSign DV	CA:FALSE

19.5 crlDistributionPoints

See also:

https://tools.ietf.org/html/rfc5280#section-4.2.1.13

In theory a complex multi-valued extension, this extension usually just holds a URI pointing to a Certificate Revokation List (CRL).

Root certificate authorities (StartSSL, GeoTrust Global, GlobalSign) do not set this field. This usually isn't a problem since clients have a list of trusted root certificates anyway, and browsers and distributions should get regular updates on the list of trusted certificates.

All CRLs linked here are all in DER/ASN1 format, and the `Content-Type` header in the response is set to `application/pkix-crl`. Only Comodo uses `application/x-pkcs7-crl`, but it is also in DER/ASN1 format.

19.5.1 In CA certificates

CA	Value	Content-Type
Let's Encrypt X1	URI:http://crl.identrust.com/DSTROOTCAX3CRL.crl	application/pkix-crl
Let's Encrypt X3	URI:http://crl.identrust.com/DSTROOTCAX3CRL.crl	application/pkix-crl
StartSSL	(not present)	
StartSSL Class 2	URI:http://crl.startssl.com/sfsca.crl	application/pkix-crl
StartSSL Class 3	URI:http://crl.startssl.com/sfsca.crl	application/pkix-crl
GeoTrust Global	(not present)	
RapidSSL G3	URI:http://g.symcb.com/crls/gtglobal.crl	application/pkix-crl
Comodo	URI:http://crl.usertrust.com/AddTrustExternalCARoot.crl	application/x-pkcs7-crl
Comodo DV	URI:http://crl.comodoca.com/COMODORSACertificationAuthority.crl	application/x-pkcs7-crl
GlobalSign	(not present)	
GlobalSign DV	URI:http://crl.globalsign.net/root.crl	application/pkix-crl

19.5.2 In signed certificates

Let's Encrypt is so far the only CA that does not maintain a CRL for signed certificates. Major CAs usually don't fancy CRLs much because they are a large file (e.g. Comodos CRL is 1.5MB) containing all certificates and cause major traffic for CAs. OCSP is just better in every way.

CA	Value	Content-Type
Let's Encrypt	(not present)	
StartSSL Class 2	URI:http://crl.startssl.com/crt2-crl.crl	application/pkix-crl
StartSSL Class 3	URI:http://crl.startssl.com/sca-server3.crl	application/pkix-crl
RapidSSL G3	URI:http://gv.symcb.com/gv.crl	application/pkix-crl
Comodo DV	URI:http://crl.comodoca.com/COMODORSADomainValidationSecureServerCA.crl	application/x-pkcs7-crl
GlobalSign DV	URI:http://crl.globalsign.com/gs/gsdomainvalsha2g2.crl	application/pkix-crl

19.6 extendedKeyUsage

A list of purposes for which the certificate can be used for. CA certificates usually do not set this field.

19.6.1 In CA certificates

CA	Value
Let's Encrypt X1	(not present)
Let's Encrypt X3	(not present)
StartSSL	(not present)
StartSSL Class 2	(not present)
StartSSL Class 3	TLS Web Client Authentication, TLS Web Server Authentication
GeoTrust Global	(not present)
RapidSSL G3	(not present)
Comodo	(not present)
Comodo DV	TLS Web Server Authentication, TLS Web Client Authentication
GlobalSign	(not present)
GlobalSign DV	(not present)

19.6.2 In signed certificates

CA	Value
Let's Encrypt X1	TLS Web Server Authentication, TLS Web Client Authentication
Let's Encrypt X3	TLS Web Server Authentication, TLS Web Client Authentication
StartSSL Class 2	TLS Web Client Authentication, TLS Web Server Authentication
StartSSL Class 3	TLS Web Client Authentication, TLS Web Server Authentication
RapidSSL G3	TLS Web Server Authentication, TLS Web Client Authentication
Comodo DV	TLS Web Server Authentication, TLS Web Client Authentication
GlobalSign DV	TLS Web Server Authentication, TLS Web Client Authentication

19.7 issuerAltName

See also:

https://tools.ietf.org/html/rfc5280#section-4.2.1.7

Only StartSSL sets this field in its signed certificates. It's a URI pointing to their homepage.

19.7.1 In CA certificates

CA	Value
Let's Encrypt	(not present)
StartSSL	(not present)
StartSSL Class 2	(not present)
StartSSL Class 3	(not present)
GeoTrust Global	(not present)
RapidSSL G3	(not present)
Comodo	(not present)
Comodo DV	(not present)
GlobalSign	(not present)
GlobalSign DV	(not present)

19.7.2 In signed certificates

CA	Value
Let's Encrypt	(not present)
StartSSL Class 2	URI:http://www.startssl.com/
StartSSL Class 3	URI:http://www.startssl.com/
RapidSSL G3	(not present)
Comodo DV	(not present)
GlobalSign DV	(not present)

19.8 keyUsage

See also:

https://tools.ietf.org/html/rfc5280#section-4.2.1.3

List of permitted key usages. Usually marked as critical, except for certificates signed by StartSSL.

19.8.1 In CA certificates

CA	Value
Let's Encrypt X1	(critical) Digital Signature, Certificate Sign, CRL Sign
Let's Encrypt X3	(critical) Digital Signature, Certificate Sign, CRL Sign
StartSSL	(critical) Certificate Sign, CRL Sign
StartSSL Class 2	(critical) Certificate Sign, CRL Sign
StartSSL Class 3	(critical) Certificate Sign, CRL Sign
GeoTrust Global	(critical) Certificate Sign, CRL Sign
RapidSSL G3	(critical) Certificate Sign, CRL Sign
Comodo	(critical) Digital Signature, Certificate Sign, CRL Sign
Comodo DV	(critical) Digital Signature, Certificate Sign, CRL Sign
GlobalSign	(critical) Certificate Sign, CRL Sign
GlobalSign DV	(critical) Certificate Sign, CRL Sign

19.8.2 In signed certificates

CA	Value
Let's Encrypt X1	(critical) Digital Signature, Key Encipherment
Let's Encrypt X3	(critical) Digital Signature, Key Encipherment
StartSSL Class 2	Digital Signature, Key Encipherment, Key Agreement
StartSSL Class 3	Digital Signature, Key Encipherment
RapidSSL G3	(critical) Digital Signature, Key Encipherment
Comodo DV	(critical) Digital Signature, Key Encipherment
GlobalSign DV	(critical) Digital Signature, Key Encipherment

19.9 subjectAltName

The `subjectAltName` extension is not present in any CA certificate, and of course whatever the customer requests in signed certificates.

19.9.1 In CA certificates

CA	Value
Let's Encrypt	•
StartSSL	•
StartSSL Class 2	•
StartSSL Class 3	•
GeoTrust Global	•
RapidSSL G3	•
Comodo	•
Comodo DV	•
GlobalSign	•
GlobalSign DV	•

19.10 subjectKeyIdentifier

See also:

https://tools.ietf.org/html/rfc5280#section-4.2.1.2

The subjectKeyIdentifier extension provides a means of identifying certificates. It is a mandatory extension for CA certificates. Currently only RapidSSL does not set this for signed certificates.

The value of the subjectKeyIdentifier extension reappears in the *authorityKeyIdentifier* extension (prefixed with `keyid:`).

19.10.1 In CA certificates

CA	Value
Let's Encrypt X1	A8:4A:6A:63:04:7D:DD:BA:E6:D1:39:B7:A6:45:65:EF:F3:A8:EC:A1
Let's Encrypt X3	A8:4A:6A:63:04:7D:DD:BA:E6:D1:39:B7:A6:45:65:EF:F3:A8:EC:A1
StartSSL	4E:0B:EF:1A:A4:40:5B:A5:17:69:87:30:CA:34:68:43:D0:41:AE:F2
StartSSL Class 2	11:DB:23:45:FD:54:CC:6A:71:6F:84:8A:03:D7:BE:F7:01:2F:26:86
StartSSL Class 3	B1:3F:1C:92:7B:92:B0:5A:25:B3:38:FB:9C:07:A4:26:50:32:E3:51
GeoTrust Global	C0:7A:98:68:8D:89:FB:AB:05:64:0C:11:7D:AA:7D:65:B8:CA:CC:4E
RapidSSL G3	C3:9C:F3:FC:D3:46:08:34:BB:CE:46:7F:A0:7C:5B:F3:E2:08:CB:59
Comodo	BB:AF:7E:02:3D:FA:A6:F1:3C:84:8E:AD:EE:38:98:EC:D9:32:32:D4
Comodo DV	90:AF:6A:3A:94:5A:0B:D8:90:EA:12:56:73:DF:43:B4:3A:28:DA:E7
GlobalSign	60:7B:66:1A:45:0D:97:CA:89:50:2F:7D:04:CD:34:A8:FF:FC:FD:4B
GlobalSign DV	EA:4E:7C:D4:80:2D:E5:15:81:86:26:8C:82:6D:C0:98:A4:CF:97:0F

19.10.2 In signed certificates

CA	Value
Let's Encrypt X1	F4:F3:B8:F5:43:90:2E:A2:7F:DD:51:4A:5F:3E:AC:FB:F1:33:EE:95
Let's Encrypt X3	71:57:F2:DC:D2:02:5C:00:5E:74:28:57:4C:7E:61:43:44:44:AF:84
StartSSL Class 2	C7:AA:D9:A4:F0:BC:D1:C1:1B:05:D2:19:71:0A:86:F8:58:0F:F0:99
StartSSL Class 3	F0:72:65:5E:21:AA:16:76:2C:6F:D0:63:53:0C:68:D5:89:50:2A:73
RapidSSL G3	(not present)
Comodo DV	F2:CB:1F:E9:6E:D5:43:E3:85:75:98:5F:97:7C:B0:59:7F:D5:C0:C0
GlobalSign DV	52:5A:45:5B:D4:9D:AC:65:30:BD:67:80:6C:D1:A1:3E:09:F7:FD:92

19.11 Other extensions

Extensions used by certificates encountered in the wild that django-ca does not (yet) support in any way.

19.11.1 In CA certificates

CA	Value
Let's Encrypt X1	X509v3 Certificate Policies, X509v3 Name Constraints
Let's Encrypt X3	X509v3 Certificate Policies
StartSSL	X509v3 Certificate Policies, Netscape Cert Type, Netscape Comment
StartSSL Class 2	X509v3 Certificate Policies
StartSSL Class 3	X509v3 Certificate Policies
GeoTrust Global	(none)
RapidSSL G3	X509v3 Certificate Policies
Comodo	X509v3 Certificate Policies
Comodo DV	X509v3 Certificate Policies
GlobalSign	(none)
GlobalSign DV	X509v3 Certificate Policies

19.11.2 In signed certificates

CA	Value
Let's Encrypt X1	X509v3 Certificate Policies
Let's Encrypt X3	X509v3 Certificate Policies
StartSSL Class 2	X509v3 Certificate Policies
StartSSL Class 3	X509v3 Certificate Policies
RapidSSL G3	X509v3 Certificate Policies
Comodo DV	X509v3 Certificate Policies
GlobalSign DV	X509v3 Certificate Policies

CHAPTER 20

x509 extensions

This page provides a list of supported TLS extensions. They can be selected in the admin interface or via the command line. Please see *Override extensions* for more information on how to set these extensions in the command line.

20.1 keyUsage

The `keyUsage` extension defines the basic purpose of the certificate. It is defined in RFC5280, section 4.2.1.3. The extension is usually defined as critical.

Name	Used for
cRLSign	
dataEncipherment	email encryption
decipherOnly	
digitalSignature	TLS connections (client and server), email and code signing, OCSP responder
encipherOnly	
keyAgreement	TLS server connections
keyCertSign	
keyEncipherment	TLS server connections, email encryption, OCSP responder
nonRepudiation	OCSP responder

Currently, the default profiles (see *CA_PROFILES* setting) use these values:

value	client	server	webserver	enduser	ocsp
cRLSign					
dataEncipherment				✓	
decipherOnly					
digitalSignature	✓	✓	✓	✓	✓
encipherOnly					
keyAgreement		✓	✓		
keyCertSign					
keyEncipherment		✓	✓	✓	✓
nonRepudiation					✓

20.2 extendedKeyUsage

The `extendedKeyUsage` extension refines the `keyUsage` extension and is defined in RFC5280, section 4.2.1.12. The extension is usually not defined as critical.

Name	Used for
serverAuth	TLS server connections
clientAuth	TLS client connections
codeSigning	Code signing
emailProtection	Email signing/encryption
timeStamping	
OCSPSigning	Running an OCSP responder
smartcardLogon	Required for user certificates on smartcards for PKINIT logon on Windows
msKDC	Required for Domain Controller certificates to authorise them for PKINIT logon on Windows

Currently, the default profiles (see *CA_PROFILES* setting) use these values:

value	client	server	webserver	enduser	ocsp
serverAuth		✓	✓	✓	
clientAuth	✓	✓		✓	
codeSigning				✓	
emailProtection					
timeStamping					
OCSPSigning					✓
smartcardLogon					
msKDC					

20.3 TLSFeature

The `TLSFeature` extension is defined in RFC7633. This extension should not be marked as critical.

Name	Description
OCSPMustStaple	TLS connections *must* include a stapled OCSP response, defined in RFC6066.
MultipleCertStatusRequest	Not commonly used, defined in RFC6961.

The use of this extension is currently discouraged. Current OCSP stapling implementation are still poor, making OCSPMustStaple a dangerous extension.

CHAPTER 21

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index