# OpenXPKI Documentation

***Release 1.2***

**The OpenXPKI Project**

2017-02-23

Contents

*An open, enterprise-grade PKI/Trustcenter*

CHAPTER 1

# Table of Contents:

## Introduction

This manual describes the installation and use of the OpenXPKI software, an Open Source trustcenter solution written by The OpenXPKI Project.

The intended audience are CA administrators and operators. We assume that readers are familiar working on a Unix shell and have enough background knowledge about Public Key Infrastructures to understand the relevant terms.

The OpenXPKI manual is split into the following sections:

- The introduction, which you are reading right now. Following this abstract, you will learn more about where to get the software, where to get help. Furthermore, a high-level overview of the system design and some key concepts will be presented.
- The *Quickstart guide* lays the emphasis on getting a minimal Certificate Authority (CA) without any bells and whistles running. Reference to further configuration options is provided inline, so that you know where to look if you want to configure more advanced features. Please note that setting up a working CA is a complex task and thus the ''quick" in "quick start" may be a bit euphemistic.
- The *Setup and Configuration* chapter gives a more detailed introduction with brief configuration examples to adjust the system to your needs based on the existing, pre-defined workflows and code.
- In the *Technical Operation Manual* chapter we describe commands and tools which are required for setting up the non-config items (e.g. the crypto tokens) and to perform daily operation.
- If you need to extend the system, read the *Extending OpenXPKI* section to learn how to use the workflow engine and the connector layer to make the system fit your needs.

### Key features

Assuming this is your first contact with OpenXPKI here is a quick summary of what it is and what it is capable of.

OpenXPKI aims to be an enterprise-scale Public Key Infrastructure (PKI) solution, supporting well established infrastructure components like RDBMS and Hardware Security Modules (HSMs). It started as the successor of OpenCA, and builds on the experience gained while developing it as well as on our experience in large public key infrastructures.

- *CA rollover:* "Normal" trust center software usually does not account for the installment of a new CA certificate, thus if the CA certificate becomes invalid, a complete re-deployment has to be undertaken. OpenXPKI solves this problem by automatically deciding which CA certificate to use at a certain point in time.
- *Support for multiple so-called PKI realms:* Different CA instances can be run in a single installation without any interaction between them, so one machine can be used for different CAs.

- *Private key support both in hardware and software:* OpenXPKI has support for professional Hardware Security Modules such as the nCipher nShield or the Safenet Luna CA modules. If such modules are not available, access to a key can be protected by using a threshold secret sharing algorithm.
- *Professional database support:* The user can choose from a range of database backends, including commercial ones such as Oracle which are typically used in enterprise scenarios.
- *Many different interfaces to the server:* Humans can access the CA server using a web-interface. Embedded devices such as routers can also use the Simple Certificate Enrollment Protocol (SCEP) to talk to the server and apply for certificates - including automatic renewal. If integration into existing systems is required, REST and SOAP interfaces are also available.
- *Workflow Engine:* OpenXPKI aims to be extremly customizable by allowing the definition of workflows for any process you can think of in the PKI area. Typical workflows such as editing and approving certificate signing requests, certificate and CRL issuance are already implemented. Implementing your own idea is normally pretty easy by defining a workflow in a YAML configuaration file and (maybe) implementing a few lines in Perl.
- *I18N:* Localization of the application and interfaces is easily possible.
- *Self-Service application for smartcard/token personalization:* A web application which allows a user to easily create and install certificates to a smartcard is available (commercial third party component required).
- *Template-based certificate details:* Contrary to the typical CA system, your users do not need to know about how you would like the subject to look like - you can just ask them for the information they know (for example a hostname and port) and OpenXPKI will create the corresponding subject and subjectAlternativeNames for you. Regular expression support allows you to enforce certificate naming conventions easily.
- *Interchangeble notifation backends* We can of course send eMail notifications to customers and operators but if you have a heavy load of certificate requests that need additional communication with the requesters, you can attach a ticket system like [http://www.bestpractical.com/rt/ Request Tracker], which will receive updates on the certificate status.

# Quickstart guide

## Vagrant

We have a vagrant setup for debian jessie and ubuntu trusty. If you have vagrant you can just checkout the git repo, go to vagrant/debian and run "vagrant up test". Provisioning takes some minutes and will give you a ready to run OXI install available at http://localhost:8080/openxpki/.

## Debian/Ubuntu Builds

**Packages are for 64bit systems (arch amd64), make sure that the en_US.utf8 locale is installed as the translation stuff will crash otherwise!**

Start with a debian minimal install, we recommend to add "SSH Server" and "Web Server" in the package selection menu, as this will speed up the install later:

Current release is 1.16 which is out for debian jessie on the package mirror at http://packages.openxpki.org/.

Add the repository to your source list (jessie):

```
echo "deb http://packages.openxpki.org/debian/ jessie release" > /etc/apt/sources.list.d/openxpki
aptitude update
```

or ubuntu trusty (still on 1.13 due to compatiblity issues):

```
echo "deb http://packages.openxpki.org/ubuntu/ dists/trusty/release/binary-amd64/" > /etc/apt/sou
aptitude update
```

To avoid an "untrusted package" warning, you should add our package signing key (works only on debian yet):

```
wget http://packages.openxpki.org/debian/Release.key -O - | apt-key add -
```

As the init script uses mysql as default, but does not force it as a dependency, it is crucial that you have the mysql server and the perl mysql binding installed before you pull the OpenXPKI package:

```
aptitude install mysql-server libdbd-mysql-perl
```

We strongly recommend to use a fastcgi module as it speeds up the UI, we recommend mod_fcgid as it is in the official main repository (mod_fastcgi will also work but is only available in the non-free repo):

```
aptitude install libapache2-mod-fcgid
```

Note, fastcgi module should be enabled explicitly, otherwise, .fcgi file will be treated as plain text (this is usually done by the installer already):

```
a2enmod fcgid
```

Some people reported that a2enmod is not available on their system, in this case try to install the apache2.2-common package.

*Ubuntu*: The provided CGI.pm does not support para_multi, either install a recent version from CPAN or manually grab an install the package from our package server!

Now install the OpenXPKI core package and the translation package:

```
aptitude install libopenxpki-perl openxpki-i18n
```

*Note: It looks like we solved this for debian but the ubuntu signatures are still "broken" and installing on ubuntu causes a "gpg error". We are still working on this issue (see #181).*

You should now restart the apache server to activate the new config:

```
service apache2 restart
```

use the openxpkiadm command to verify if the system was installed correctly:

```
openxpkiadm version
Version (core): 1.0.0
```

Now, create an empty database and assign a database user:

```
CREATE DATABASE openxpki CHARSET utf8;
CREATE USER 'openxpki'@'localhost' IDENTIFIED BY 'openxpki';
GRANT ALL ON openxpki.* TO 'openxpki'@'localhost';
flush privileges;
```

...and put the used credentials into /etc/openxpki/config.d/system/database.yaml:

```
main:
   debug: 0
   type: MySQL
   name: openxpki
   host: localhost
   port: 3306
   user: openxpki
   passwd: openxpki
```

Starting with v1.13, the "initdb" command is deprecated, please create the empty database schema from the provided schema file (currently only available for mysql).

Example call when debian packages are installed:

```
zcat /usr/share/doc/libopenxpki-perl/examples/schema-mysql.sql.gz | \
    mysql -u root -p openxpki
```

If you do not use debian packages, you can get a copy from the config/sql/ folder of the repository.

### Setup base certificates

The debian package comes with a shell script `sampleconfig.sh` that does all the work for you (look in /usr/share/doc/libopenxpki-perl/examples/). The script will create a two stage ca with a root ca certificate and below your issuing ca and certs for SCEP and the internal datasafe.

The sample script provides certs for a quickstart but should never be used for production systems (it has the fixed passphrase *root* for all keys ;) and no policy/crl, etc config ).

Here is what you need to do if you *dont* use the sampleconfig script.

1. Create a key/certificate as signer certificate (ca = true)
2. Create a key/certificate for the internal datavault (ca = false, can be below the ca but can also be self-signed).
3. Create a key/certificate for the scep service (ca = false, can be below the ca but can also be self-signed or from other ca).

Move the key files to /etc/openxpki/ssl/ca-one/ and name them ca-one-signer-1.pem, ca-one-vault-1.pem, ca-one-scep-1.pem. The key files must be readable by the openxpki user, so we recommend to make them owned by the openxpki user with mode 0400.

Now import the certificates to the database. The signer token is used exclusive in the current realm, so we can use a shortcut and import and reference it with one command.

```
openxpkiadm certificate import  --file ca-root-1.crt

openxpkiadm certificate import  --file ca-one-signer-1.crt \
    --realm ca-one --token certsign
```

As we might want to reuse SCEP and Vault token across the realms, we import them in to the global namespace and just create an alias in the current realm:

```
openxpkiadm certificate import  --file ca-one-vault-1.crt
openxpkiadm certificate import  --file ca-one-scep-1.crt

openxpkiadm alias --realm ca-one --token datasafe \
    --identifier `openxpkiadm certificate id --file ca-one-vault-1.crt`

openxpkiadm alias --realm ca-one --token scep \
    --identifier `openxpkiadm certificate id --file ca-one-scep-1.crt`
```

If the import went smooth, you should see something like this (ids and times will vary):

```
$ openxpkiadm alias --realm ca-one

=== functional token ===
ca-one-scep (scep):
Alias     : ca-one-scep-1
Identifier: YsBNZ7JYTbx89F_-Z4jn_RPFFWo
NotBefore : 2015-01-30 20:44:40
NotAfter  : 2016-01-30 20:44:40

ca-one-vault (datasafe):
Alias     : ca-one-vault-1
Identifier: lZILS1l6Km5aIGS6pA7P7azAJic
NotBefore : 2015-01-30 20:44:40
NotAfter  : 2016-01-30 20:44:40

ca-one-signer (certsign):
Alias     : ca-one-signer-1
Identifier: Sw_IY7AdoGUp28F_cFEdhbtI9pE
NotBefore : 2015-01-30 20:44:40
NotAfter  : 2018-01-29 20:44:40

=== root ca ===
```

```
current root ca:
Alias     : root-1
Identifier: fVrqJAlpotPaisOAsnxa9cglXCc
NotBefore : 2015-01-30 20:44:39
NotAfter  : 2020-01-30 20:44:39

upcoming root ca:
  not set
```

Now it is time to see if anything is fine:

```
$ openxpkictl start

Starting OpenXPKI...
OpenXPKI Server is running and accepting requests.
DONE.
```

In the process list, you should see two process running:

```
14302 ?        S      0:00 openxpki watchdog ( main )
14303 ?        S      0:00 openxpki server ( main )
```

If this is not the case, check */var/openxpki/stderr.log*.

### Adding the Webclient

The new webclient is included in the core packages now. Just open your browser and navigate to *http://yourhost/openxpki/*. You should see the main authentication page. If you get an internal server error, make sure you have the en_US.utf8 locale installed (*locale -a | grep en_US*)!

You can log in as user with any username/password combination, the operator login has two preconfigured operator accounts raop and raop2 with password openxpki.

### Testdrive

1. Login as User (Username: bob, Password: <any>)
2. Go to "Request", select "Request new certificate"
3. Complete the pages until you get to the status "PENDING" (gray box on the right)
4. Logout and re-login as RA Operator (Username: raop, Password: openxpki )
5. Select "Home / My tasks", there should be a table with one request pending
6. Select your Request by clicking the line, change the request or use the "approve" button
7. After some seconds, your first certificate is ready :)
8. You can download the certificate by clicking on the link in the first row field "certificate"
9. You can now login with your username and fetch the certificate

### Enabling the SCEP service

**Note: You need to manually install the openca-tools package which is available from our package server in order to use the scep service.**

The SCEP logic is already included in the core distribution. The package installs a wrapper script into /usr/lib/cgi-bin/ and creates a suitable alias in the apache config redirecting all requests to `http://host/scep/<any value>` to the wrapper. A default config is placed at /etc/openxpki/scep/default.conf. For a testdrive, there is no need for any configuration, just call `http://host/scep/scep`.

The system supports getcacert, getcert, getcacaps, getnextca and enroll/renew - the shipped workflow is configured to allow enrollment with password or signer on behalf. The password has to be set in `scep.yaml`, the default is 'SecretChallenge'. For signing on behalf, use the UI to create a certificate with the 'SCEP Client' profile - there is no password necessary. Advanced configuration is described in the scep workflow section.

The best way for testing the service is the sscep command line tool (available at e.g. https://github.com/certnanny/sscep).

Check if the service is working properly at all:

```
mkdir tmp
./sscep getca -c tmp/cacert -u http://yourhost/scep/scep
```

Should show and download a list of the root certificates to the tmp folder.

To test an enrollment:

```
openssl req -new -keyout tmp/scep-test.key -out tmp/scep-test.csr -newkey rsa:2048 -nodes
./sscep enroll -u http://yourhost/scep/scep \
    -k tmp/scep-test.key -r tmp/scep-test.csr \
    -c tmp/cacert-0 \
    -l tmp/scep-test.crt \
    -t 10 -n 1
```

Make sure you set the challenge password when prompted (default: 'SecretChallenge'). On current desktop hardware the issue workflow will take approx. 15 seconds to finish and you should end up with a certificate matching your request in the tmp folder.

### Support for Java Keystore

OpenXPKI can assemble server generated keys into java keystores for immediate use with java based applications like tomcat. This requires a recent version of java `keytool` installed. On debian, this is provided by the package `openjdk-7-jre`. Note: You can set the location of the keytool binary in `system.crypto.token.javajks`, the default is /usr/bin/keytool.

# Upgrading OpenXPKI

We try hard to build releases that do not break old installations but sometimes we are forced to make changes that require manual adjustment of existing config or even the database schema.

This page provides a summary of recommended and mandatory changes. Recommended items should be done but the installation will continue to work. Mandatory items MUST be done, as otherwise the system will not behave correctly or even wont start.

For a quick overview of config changes, you should always check the config repository at https://github.com/openxpki/openxpki-config.

## Release v1.13

The default config now uses /var/log/openxpki/ as log directory. It is no problem to leave your log files where there are but you need to fix the permissions on the frontend logs after running the update:

```
cd /var/openxpki/; chown www-data webui.log scep.log soap.log rpc.log
```

We will fix this in the debian update with the next release.

## Release v1.11

We put access to workflow log/history/context under access control. If you want your users/operators to have access to those items, you MUST add the new acl items to your workflow definitions:

```
acl:
  RA Operator:
    creator: any
    fail: 1
    resume: 1
    wakeup: 1
    history: 1
    techlog: 1
    context: 1
```

If you are using the SOAP revocation interface or want to use the new RPC revocation interface, you MUST add a new field to the inital action.

Add the file config.d/realm/ca-one/workflow/global/field/interface.yaml to your config tree. In config.d/realm/ca-one/workflow/def/certificate_revocation_request_v2.yaml add the field "interface" to the list of "input" fields of "create_crr".

## Release v1.10

Please update your database schema:

```
DROP TABLE IF EXISTS `seq_application_log`;
CREATE TABLE IF NOT EXISTS `seq_application_log` (
  `seq_number` bigint(20) UNSIGNED NOT NULL AUTO_INCREMENT,
  `dummy` int(11) DEFAULT NULL,
  PRIMARY KEY (`seq_number`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8;

DROP TABLE IF EXISTS `application_log`;
CREATE TABLE IF NOT EXISTS `application_log` (
  `application_log_id` bigint(20) unsigned NOT NULL,
  `logtimestamp` bigint(20) unsigned DEFAULT NULL,
  `workflow_id` decimal(49,0) NOT NULL,
  `priority` int(11) DEFAULT 999,
  `category` varchar(255) NOT NULL,
  `message` longtext,
  PRIMARY KEY (`application_log_id`),
  KEY (`workflow_id`),
  KEY (`workflow_id`,`priority`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8;
```

Append "DBI" for the application logger in /etc/openxpki/log.conf:

```
log4perl.category.openxpki.application = INFO, Logfile, DBI
```

Setup and Configuration

# Setup and Configuration

# Writing a workflow activity

## Perl Code

We strongly advise to create a custom namespace for your workflow classes! You **should** derive your classes from *OpenXPKI::Server::Workflow::Activity*, but you **must** derive if you want to use the pause/resume/autofail functionality.

A skeleton class OpenXPKI::Server::Workflow::Activity::Skeleton is provided as a point to start with. Here is some pseudocode to show the basic idea on the scheduler system.

### pause a workflow

If you need to wait for some other things to happen before you can continue, you can simply pause the workflow by calling:

```
if ($need_to_wait) {
    $self->pause('Waiting for Godot');
}
```

The activity will return immediately and the workflow stops. After the retry_interval elapsed, the scheduler reinstantiates the workflow, calls the *wake_up* funktion and afterwards the *execute* method again.

You can also set the retry parameters from inside the action:

```
# Set the interval after three invalid tries to 1 hour
if ($workflow->get_retry_count() > 3) {
    $self->set_retry_intervall('+00000001');
}
```

### resume from an exception

It might happen that your code raised an exception and you need to clean up the context or check/rollback some other things before you will continue the normal operation. Such code should be placed into the *resume* method which gets executed after an exception was thrown.

## Predefined context values and namespaces

The UI and workflow engine looks for certain context values to render error and status messages. Those values *should* be used to provide information. You *must not* use such predefined keys for any other purpose.

### certificate / workflow relations

Please record workflows that change a certificates status in the certificate attributes table, using the prefix *system_workflow_XXXX*. The core uses system_workflow_csr, system_workflow_crr, system_workflow_crl to link a certificate to the appropriate workflows for issue, revoke and crl creation.

Those workflows are shown on the “related information” page.

### XML Configuration

# TODO - Update for YAML! XML is gone!

To use an activity in the workflow system, you must create a mapping between the perl class and a symbolic name. It is allowed to reuse the same class with different names (and likely different parameters):

```
<action name="I18N_OPENXPKI_WF_ACTION_MY_ACTION"
    class="OpenXPKI::Server::Workflow::Custom::Activity::MyAction">
</action>
```

For detailed information see the perldoc of Workflow.pm.

The OpenXPKI server comes with a scheduler to detach workflows from the frontend and continue them in the background.:

```
<actions>
    <action name="I18N_OPENXPKI_WF_ACTION_MY_ACTION"
        class="OpenXPKI::Server::Workflow::Custom::Activity::MyAction"
            retry_count="5"
            retry_interval="+0000000030">
    </action>
</actions>
```

You can set the retry parameters also in the workflow definition. Those settings are superior to the once in the action definition:

```
<state name="STEP3">
    <description></description>

    <action name="I18N_OPENXPKI_WF_ACTION_MY_ACTION"
        resulting_state="SUCCESS"
        retry_count="7"
        autofail="yes">
    </action>
  </state>
```

This configuration will run the action up to 7 times with a pause of approx 30 minutes between the retries. If the method does not finish while its seventh loop, the workflow is stopped with an error. As we have the *autofail* flag set, the workflow is immediately send into the *FAILURE* state and set to finished. Note that the autofail flag is only allowed in the <state> block, but not in the activity definition itself!

#### Important Precondition

Do not use activities with pause together with conditions that might change over time! Workflows are always resumed by state and if your paused activity is linked with a condition, it gets re-evaluated. If the result is false now, the workflow layer will block access to the (formerly paused) activity so the watchdog can not rerun it.

To solve such issues, either create two seperate states for the evaluation and the paused activity (using a NOOP activity) or write your condition in a way that it will not change, e.g. by persisting the relevant parameters into the context on the first run.

## OpenXPKI Configuration

### Main configuration

The configuration of OpenXPKI consists of two, fundamental different, parts. There is one global system configuration, which holds information about database, filesystem, etc. where the system lives. The second part are the realm configurations, which define the properties of the certificates within the realm. Each pki realm has its own,

independant configuration and is isolated from other realm, so you can run instances with different behaviour with one single OpenXPKI server.

We ship the software with a set of YaML files, and we recommend to keep the given layout. The following documentation uses some notations you should know about.

1. Configuration items are read as a path, which is denoted as a string with the path elements seperated by a dot, e.g. `system.database.main.type`.
2. The path is assembled from the directory, the name of the configuration file, the path of the element in the YaML notation. The value from the example above can be found in the directory `system`, file `database.yaml`, section `main`, key `type`.
3. All paths except those starting with *system* or *realm* refer to the configuration of a particular realm. The root node for building the path is the realm's directory found at `realm/<name of the realm>`.

## Config versioning

This idea was dropped, configuration is now read freshly from the filesystem at every restart of the daemon.

### Activiate the new configuration

To activate a new config without a restart, you need to do a reload:

```
openxpkicli reload
```

You can also just send a `SIGHUP` to the main process or restart the dameon.

**IMPORTANT**

Those parts of the system are preloaded on server init and need a restart to load a new configuration.

- worflow configuration
- authentication handlers
- database settings
- daemon settings (**never** change anything below *system.server* while the dameon is running as you might screw up your system!)

## Logging

OpenXPKI uses Log4perl as its primary system log. Logging during startup and in critical situations is done via STDERR.

# Global configuration

This document give a brief overview over the system configuration items. You find those settings in the files in the `<configfir>/config.d/system` directory.

## Database

Configure the settings for the database layer. You need at least a configuration for the *main* database. It is possible to provide a seperate database for logging purposes, just copy the complete block from `system.database.main` to `system.database.logging` and adjust as required. If you don't configure a logging database, the main database is used.

The general configuration block looks like:

```
main:
    type: supported driver name
    name: name of the database
    host: host
    port: port
    user: database user
    passwd: database credential
    namespace: namespace (to be used with oracle)
    environment:
        db_driver_env_key: value
```

OpenXPKI supports MySQL, PostgreSQL, Oracle and DB2. The *namespace* parameter is used only by the Oracle driver. DB2 uses only the *name* parameter and reads other settings from the environment, which are passed as a key/value list below the *environment* key.

Check perldoc OpenXPKI::Server::DBI::Driver::<type> for more info on the parameters.

## System

Settings about filesystem, daemon and services to start. Located at `system.server`

**os related stuff**

i18n locale settings:

```
i18n:
    locale_directory: path to the gettext locales on your system
    default_language: supported locale (e.g. en_US.utf8)
```

Location of the locale files and the default language used. If you set another language than `C`, make sure you have the correct po-files installed, otherwise OpenXPKI won't even start! This usually only affects logging and system messages as most of the client related output uses the locale settings from the client session. We recommend using *C* as default.

**daemon settings**

Those settings determine the properties of the OpenXPKI daemon *openxpkid*.:

```
name:          label for your process list, useful if you are running multiple servers.
user:          Unix user to run as (numeric or name)
group:         Group to run as (numeric or name)

socket_file:   Location of the communication socket.
pid_file:      Location of the pid file.
environment:
    key: value

log4perl:      path to your Log4perl configuration file (the primary system logger).
stderr:        File to redirect stderr to after dettaching from console.
tmpdir:        Location for temporary files, writable by the daemon.

session:
    directory: Directory to store the session information.
    lifetime:  Lifetime of the sessions on the server side.
```

The socket, pidfile and stderr are created during startup while running as root. The directory must exist, be writeable by root and accessible by the user the daemon runs as. The *tmpdir* must be writable by the daemon user, it can be a ramfs but can grow large in high volume environments.

**system internals**

```
transport:
    Simple: 1
```

The transport setting is reserved for future use, leave it untouched.

```
service:
    Default:
        enabled: 1
        timeout: 120

    SCEP:
        enabled: 1
```

The *service* block lists all services to be enabled, the key is the name of the service, the *enabled* key is supported by all services, for all other parameters consult the concrete service documentation (perldoc OpenXPKI::Service::<ServiceName>).

**multi-node support**

```
shift: 8
node:
    id: 0

data_exchange:
```

TODO - this is not used yet

## Watchdog

The openxpkid daemon forks a watchdog process to take care of background processes. It is initialised with default settings, but you can provide your own values by setting them at `system.watchdog`.

```
# How to deal with exceptions
max_exception_threshhold: 10
interval_sleep_exception: 60
max_tries_hanging_workflows:  3

# Control the wait intervals
interval_wait_initial: 60
interval_loop_idle: 5
interval_loop_run: 1

# You should not change this unless you know what you are doing
max_instance_count: 1
disabled: 0
```

Please see perldoc OpenXPKI::Server::Watchdog for details.

## Crypto layer (global)

Define several parameters for the basic crypto tools.

**api settings**

You should not need to touch this unless you are developing your own crypto classes.

```
tokenapi:
    certsign:       OpenXPKI::Crypto::Backend::API
    datasafe:       OpenXPKI::Crypto::Backend::API
    scep:           OpenXPKI::Crypto::Tool::SCEP::API
```

The setting denotes the name of the perl module used as backend class when using a token of the given class. Default tokens are *certsign*, is used for all ca operations, and *datasafe*, used to internally´ encrypt data. Any tokens that are not defined here, use *OpenXPKI::Crypto::Backend::API* by default. If you run a scep server, you must add the line for the scep module, as it does not work with the default.

**configuration of the default tokens**

```
token:
    default:
        backend: OpenXPKI::Crypto::Backend::OpenSSL
        api:     OpenXPKI::Crypto::Backend::API
        engine:  OpenSSL
        key_store: OPENXPKI

        # OpenSSL binary location
        shell: /usr/bin/openssl

        # OpenSSL binary call gets wrapped with this command
        wrapper: ''

        # random file to use for OpenSSL
        randfile: /var/openxpki/rand

    pkcs7:
        backend: OpenXPKI::Crypto::Tool::PKCS7
        api: OpenXPKI::Crypto::Tool::PKCS7::API

    javaks:
        backend: OpenXPKI::Crypto::Tool::CreateJavaKeystore
        api: OpenXPKI::Crypto::Tool::CreateJavaKeystore::API
```

If you have non-standard file locations, you might want to change the OpenSSL relevant settings here, the *wrapper* allows you to provide the name of a wrapper command which is commonly necessary if you use hardware security modules or other special OpenSSL eninges for your crypto operations. See the section about using HSMs for more details.

Developer note: See OpenXPKI::Crypto::TokenManager::get_system_token

### PKI Realms

The detailed settings of each realm are given in the specific realm configuration. To use a realm you need to specify and enable it at `system.realms`.

```
ca-one:
    label: This is just a verbose label for your CA
```

You should use only 7bit word characters and no spaces as name for the realm.

## Realm configuration

To create a new realm for your OpenXPKI installation, you need to create a configuration for it. The fastest way is to copy `config.d/realm.tpl` to `config.d/<realm_name>`. *realm_name* must match the name you gave the realm in `system.realms`.

The realm configuration consists of five major parts:

**Authentication** Configure which authentication mechanisms to use for the realm. If you use internal authentication methods, this also holds your user databases.

**Crypto layer** Define name and path of your keyfiles and settings for the crypto tokens used.

**Profiles** Anything related to your certificate and crl profiles

**Publishing** How and where to publish certificates and crls

**Workflow** Configuration data of the internal workflow engine.

## Authentication

Authentication is done via authentication handlers, multiple handlers are combined to an authentication stack.

### Stack

The authentication stacks are set below `auth.stack`:

```
User:
    description: I18N_OPENXPKI_CONFIG_AUTH_STACK_DESCRIPTION_USER
    handler:
    - User Password
    - Operator Password
```

The code above defines a stack *User* which internally uses two handlers for authentication. You can define any number of stacks and reuse the same handlers inside. You must define at least one stack.

### Handler

A handler consists of a perl module, that provides the authentication mechanism. The name of the handler is used to be referenced in the stack definition, mandatory entries of each handler are *type* and *label*. All handlers are defined below OpenXPKI::Server::Authentication, where *type* is equal to the name of the module.

Here is a list of the default handlers and their configuration sets.

**anonymous user**

If you just need an anonymous connection, you can use the *Anonymous* handler.

```
Anonymous:
    type: Anonymous
    label: Anonymous

System:
    type: Anonymous
    label: System
    role: System
```

If no role is provided, you get the anonymous role. **Do never set any other role than system, unless you exactly know what you are doing!**

**x509 based authentication**

There are two handlers using x509 certificates for authentication. *X509Client* uses the SSL client authentication feature of apache/mod_ssl while *X509Challenge* sends a challenge to be signed by the browser. Both handlers pass the signer certificate to a validation function that cryptographically checks the chain and tests the chain against a list of trusted anchors.

The configuration is the same for both handlers (apart from the class name):

```
Certificate:
    type: ClientX509/ChallengeX509
    label: Certificate
    description: I18N_OPENXPKI_CONFIG_AUTH_HANDLER_DESCRIPTION_CERTIFICATE_WEBSERVER
    role:
        default: User
        handler@: connector:auth.connector.role
        argument: cn
    realm:
    - ca-one
    cacert:
    - cert_identifier of external ca cert
```

The role assignment is done by querying the connector specified by *handler* using the certificates component *argument*. Possible arguments are "cn", "subject" and "serial". The value given by *default* is assigned if no match is found by the handler. If you do not specify a handler but a default role, you get a static role assignment for any matching certificate.

For the trust anchor you have consider two different situations:

1. If the certificates originate from the OpenXPKI instance itself, list the realms which issue them below *realm*.
2. If you have certificates from an external ca, import the ca certificate with the `openxpkiadm` utility and put its certificate identifier below *cacert*.

Both lists can be combined and accept any number of items.

**Note**: OpenXPKI uses a third party tool named openca-sv to check the x509 signature. You need to build that by your own and put it into /usr/bin. The source is available at http://www.openca.org/projects/openca/tools-sources.shtml.

**password database handler**

The password database handler allows to specify user/password/role pairs directly inside the configuration.

```
Password:
    type: Password
    label: User Password
    description: I18N_OPENXPKI_CONFIG_AUTH_HANDLER_DESCRIPTION_PASSWORD
    user:
        John Doe:
            digest: "{SSHA}TZXM/aqflDDQAmSWVxSDVWnH+NhxNU5w"
            role: User
        root:
            digest: "{SSHA}+u48F1BajP3ycfY/azvTBqprsStuUnhM"
            role: CA Operator
        raop:
            digest: "{SSHA}ejZpY22dFwjVI48z14y2jYuToPRjOXRP"
            role: RA Operator
```

The passwords are hashed, the used hash algorithm is given as prefix inside the curly brackets. You should use only *SSHA* which is "salted sha1". For compatibility we support plain sha (sha1), md5, smd5 (salted md5) and crypt. You can created the salted passwords using the openxpkiadm CLI tool.

If you plan to use static passwords for a larger amount of users, you should consider to use a connector instead:

```
Password:
    type: Password
    label: User Password
    description: I18N_OPENXPKI_CONFIG_AUTH_HANDLER_DESCRIPTION_PASSWORD
    user@: connector:auth.connector.userdb
```

Define the user database file inside auth.connector.yaml:

```
userdb:
    class: Connector::Proxy::YAML
    LOCATION: /home/pkiadm/ca-one-userdb.yaml
```

The user file has the same structure as the *user* section above, the user names are the on the top level:

```
root:
    digest: "{SSHA}+u48F1BajP3ycfY/azvTBqprsStuUnhM"
    role: CA Operator
raop:
    digest: "{SSHA}ejZpY22dFwjVI48z14y2jYuToPRjOXRP"
    role: RA Operator
```

You can share a user database file within realms.

**authentication connectors**

There is a family of authentication connectors. The main difference against other connector is, that the password is passed as a parameter and is not part of the path. Check for connectors starting with Connector::Builtin::Authentication. The connector only validates the password, therefore the role must be set in the configuration (same for all users handled by this item):

```
Password Connector:
    type: Connector
    label: User Password
    description: I18N_OPENXPKI_CONFIG_AUTH_HANDLER_DESCRIPTION_PASSWORD
    role: User
    source@: connector:auth.connector.localuser
```

An example config to authenticate RA Operators against ActiveDirectory using their company mail address and windows password including check of a group membership (this is just the authentication, set the role in the handler config):

```
raop-ad:
    class: Connector::Builtin::Authentication::LDAP
    LOCATION: ldap://ad.company.com
    base: dc=company,dc=loc
    binddn@: cn=binduser
    password@: secret
    filter: "(&(mail=[% LOGIN %])(memberOf=CN=RA Operator,OU=SecurityGroups,DC=company,DC=loc))"
```

**external authentication**

If you have a proxy or sso system in front of your OpenXPKI server that authenticates your users, the external handler can be used to set the user information:

```
External Dynamic Role:
    type: External
    label: External Dynamic Role
    description: I18N_OPENXPKI_CONFIG_AUTH_HANDLER_DESCRIPTION_EXTERNAL
    command: echo -n $PASSWD
    # if this field is empty then the role is determined dynamically -->
    role: ''
    pattern: x
    replacement: x
    env:
       LOGIN: __USER__
       PASSWD: __PASSWD__
```

TODO: This needs some useful example code.

### Workflow ACL

The Workflow-ACL set is located at `auth.wfacl` and controls which workflows a user can access. The rules are based on the role of the user and distinguish between creating a new and accessing an exisiting workflow.

**workflow creation**

To determine what workflows a user can create, just list the names of the workflows under the create key.

```
User:
    create:
    - I18N_OPENXPKI_WF_TYPE_CERTIFICATE_RENEWAL_REQUEST
    - I18N_OPENXPKI_WF_TYPE_CERTIFICATE_REVOCATION_REQUEST
    - I18N_OPENXPKI_WF_TYPE_CERTIFICATE_SIGNING_REQUEST
    - I18N_OPENXPKI_WF_TYPE_PASSWORD_SAFE
```

**unconditional workflow access**

The access privileg takes the workflow creator into account. To get access to all existing workflows regardless of the creator, use a wildcard pattern:

```
User:
    access:
        I18N_OPENXPKI_WF_TYPE_CERTIFICATE_RENEWAL_REQUEST:
            creator: .*
```

**conditional workflow access**

To show a user only his own workflows, use the special word *self*:

```
User:
    access:
        I18N_OPENXPKI_WF_TYPE_CERTIFICATE_RENEWAL_REQUEST:
            creator: self
```

**workflow context filter**

Sometimes the workflow context contains items, you don't want to show to the user. You can specify a regular expression to show or hide certain entries. The regex is applied to the context key:

```
User:
    access:
        I18N_OPENXPKI_WF_TYPE_PASSWORD_SAFE:
            creator: self
            context:
                show: .*
                hide: encrypted_.*
```

The given example shows everything but any context items that begin with "encrypted_". The filters are additive, so a key must match the show expression but must not match the hide expression to show up. *Note*: No setting or an empty string for *show* results in no filtering! To hide the whole context set a wildcard ".*" for *hide*.

## Crypto layer

### group assignment

You must provide a list of token group names at `crypto.type` to tell the system which token group it should use for a certain task. The keys are the same as used in `system.crypto.tokenapi` (see Crypto layer (global)). See TODO for a detailed view how the token assignment works.

```
type:
  certsign: ca-one-certsign
  datasafe: ca-one-vault
  scep: ca-one-scep
```

### token setup

Any token used within OpenXKI needs a corresponding entry in the realm's token configuration at `crypto.token`. The name of the token is the alias name you used while registering the correspondig certificate.

```
token:
  ca-one-certsign:
    backend: OpenXPKI::Crypto::Backend::OpenSSL

    key: /etc/openxpki/ssl/ca-one/ca-one-certsign-1.pem

    # possible values are OpenSSL, nCipher, LunaCA
    engine:         OpenSSL
    engine_section: ''
    engine_usage:   ''
    key_store:      OPENXPKI
```

```
    # OpenSSL binary location
    shell: /usr/bin/openssl

    # OpenSSL binary call gets wrapped with this command
    wrapper: ''

    # random file to use for OpenSSL
    randfile: /var/openxpki/rand

    # Secret group
    secret: default
```

The most important setting here is *key* which must be the absolute filesystem path to the keyfile. The key must be in PEM format and is protected by a password. The password is taken from the secret group mentioned by *secret*. See TODO for the meaning of the other options.

**using inheritance**

Usually the tokens in a system share a lot of properties. To simplify the configuration, it is possible to use inheritance in the configuration:

```
token:
    default:
        backend: OpenXPKI::Crypto::Backend::OpenSSL
        ......
        secret: default

    server-ca-1:
        inherit: default
        key: /etc/openxpki/ssl/ca-one/ca-one-certsign-1.pem
        secret: gen1pass

    server-ca-2:
        inherit: default
        key: /etc/openxpki/ssl/ca-one/ca-one-certsign-2.pem
```

Inheritance can daisy chain profiles. Note that inheritance works top-down and each step replaces all values that have not been defined earlier but are defined on the current level. Therefore you should not use undef values but the empty string to declare an empty setting.

You can use template toolkit to autoconfigure the `key` property, this way you can roll over your key without modifying your configuration.

The example above will then look like:

```
token:
    default:
        backend: OpenXPKI::Crypto::Backend::OpenSSL
        key: /etc/openxpki/ssl/ca-one/[% ALIAS %].pem
        ......
        secret: default

    server-ca-1:
        inherit: default
        secret: gen1key

    server-ca-2:
        inherit: default
```

If you need to name your keys according to a custom scheme, you also have GROUP (ca-one-certsign) and GENERATION (1) available for substitution.

### secret groups

A secret group maintains the password cache for your keys and PINs. You need to setup at least one secret group for each realm. The most common version is the plain password:

```
secret:
  default:
    label: One Piece Password
    method: plain
    cache: daemon
```

This tells the OpenXPKI daemon to ask for the default only once and then store it "forever". If you want to have the secret cleared at the end of the session, set *cache: session*.

To increase the security of your key material, you can configure secret splitting (k of n).

```
secret:
  ngkey:
    label: Split secret Password
    method: split
    total_shares: 5
    required_shares: 3
    cache: daemon
```

TODO: How to create the password segments?

If you have a good reason to put your password into the configuration, use the *literal* type:

```
secret:
  insecure:
    label: A useless Password
    method: literal
    value: my_not_so_secret_password
    cache: daemon
```

You can also use the secret groups for other purposes, in this case you need to add "export: 1" to the group. This allows you to use the get_secret method of the TokenManager (OpenXPKI::Crypto::TokenManager) to retrieve the plain value of the secret.

## Profiles

### certificates

There is a TODO:link seperate section about certificate profile configuration.

### certificate revocation list

A basic setup must provide at least a minimum profile for crl generation at `crl.default`:

```
digest: sha1
validity:
    nextupdate: +000014
    renewal: +000003
```

The *nextupdate* value gives the validity of the created crl (14 days). The *renewal* value tells OpenXPKI how long before the expiry date of the current crl the system is allowed to create a new one. If you set this to a value larger than *nextupdate*, a new crl is created every time you trigger a new crl creation workflow. Note: If a certificate becomes revoked, the renewal interval is not checked.

**crl at "end of life"**

Once your ca certificate exceeds its validity, you are no longer able to create new crls (at least if you are using the shell modell). OpenXPKI allows you to define a different validity for the last crl, which is taken if the next calculated renewal time will exceed the validity of the ca certificate:

```
validity:
    nextupdate: +000014
    renewal: +000003
    lastcrl: 20301231235900
```

**crl extensions**

The following code shows the full set of supported extensions, you can skip what you do not need:

```
extensions:
    authority_info_access:
        critical: 0
        ca_issuers: http://myca.mycompany.com/[% CAALIAS.ALIAS %]/cacert.pem
        ocsp:
        - http://ocsp1.mycompany.com/
        - http://ocsp2.mycompany.com/

    authority_key_identifier:
        critical: 0
        keyid:  1
        issuer: 1


    issuer_alt_name:
        critical: 0
        # If the issuer has no subject alternative name, copying returns
        # an empty extension, which is problematic with both RSA SecurId
        # tokens and Cisco devices!
        copy: 0
```

There are two specialities in handling the *ca_issuers* and *ocsp* entries in the *authority_info_access* section:

1. You can pass either a list or a single scalar to each item.
2. For each item, template expansion based on the signing ca certificate is available. See TODO:link for details.

The `CAALIAS` hash also offers the components of the alias in GENERATION and GROUP.

## Publishing

Publishing of certificates and crl is done via connectors (TODO:link). The default workflows look for targets at `publishing.entity` and `publishing.crl`. Each target can contain a list of key-value pairs where the value points to a valid connector item while the keys are used for internal logging:

```
entity:
    int-repo@: connector:publishing.connectors.ldap
    ext-repo@: connector:publishing.connectors.ldap-ext

crl:
    crl@: connector:publishing.connectors.cdp
```

**certificate publishing**

The OpenXPKI packages ship with a sample configuration for LDAP publication but you might include any other connector. The publication workflow appends the common name of the certificate to the connector path and passes a hash containing the subject (*subject*) and the DER (*der*) and PEM (*pem*) encoded certificate.

The configuration block looks like this:

```
connectors:
    ldap-ext:
        class: Connector::Proxy::Net::LDAP::Single
        LOCATION: ldap://localhost:389
        base: ou=people,dc=mycompany,dc=com
        filter: (|(mail=[% ARG %]) (objectCategory=person))
        binddn: cn=admin,dc=mycompany,dc=com
        password: secret
        attrmap:
            der: usercertificate;binary

        create:
            basedn: ou=people,dc=mycompany,dc=com
            rdnkey: cn

        schema:
            cn:
                objectclass: inetOrgPerson
                values:
                    sn: copy:self
                    ou: IT Department
```

Let's explain the parts.

```
class: Connector::Proxy::Net::LDAP::Single
LOCATION: ldap://localhost:389
base: ou=people,dc=mycompany,dc=com
filter: (|(mail=[% ARG %]) (objectCategory=person))
binddn: cn=admin,dc=mycompany,dc=com
password: secret
```

Use the Connector::Proxy::Net::LDAP::Single package and use *cn=admin,dc=mycompany,dc=com* and *secret* to connect with the ldap server at *ldap://localhost:389* using *ou=people,dc=mycompany,dc=com* as the basedn. Look for an entry of class person where the mailadress is equal to the common name of the certificate.

```
attrmap:
    der: usercertificate;binary
```

Publish the content of the internal key *der* to the ldap attribute *usercertificate;binary*.

```
create:
    basedn: ou=people,dc=mycompany,dc=com
    rdnkey: cn
```

This enables the auto-creation of non-existing nodes. The dn of the new node is create from the basedn and the new component of class "cn" set to the path-item which was passed to the connector (in our example the mailadress). You also need to pass the structural information for the node to create.

```
schema:
    cn:
        objectclass: inetOrgPerson
        values:
            sn: copy:self
            ou: IT Department
```

**crl publishing**

The crl publication workflow appends the common name of the ca certificate to the connector path and passes a hash containing the subject (*subject*), the components of the parsed subject as hash (*subject_hash*) and the DER (*der*) and PEM (*pem*) encoded crl.

The default configuration comes with a text-file publisher for the crl:

```
cdp:
    class: Connector::Builtin::File::Path
    LOCATION: /var/www/openxpki/myrealm/crls/
    file: "[% ARGS %].crl"
    content: "[% pem %]"
```

If the dn of your current ca certificate is like "cn=My CA1,ou=ca,o=My Company,c=us", this connector writes the PEM encoded crl to the file */var/www/openxpki/myrealm/crls/My CA1.crl*

## Notification

Notifications are triggered from within a workflow. The workflow just calls the notification layer with the name of the message which should be send, which can result in no message or multiple messages on different communication channels.

The configuration is done per realm at `notification`. Supported connectors are Mail via SMTP (plain and html) and RT Request Tracker (using the RT::Client::REST module from CPAN). You can use an arbitrary number of backends, where each one has its own configuration at `notification.mybackend`.

Most parts of the messages are customized using the Template Toolkit. The list of available variables is given at the end of this section.

### Sending mails using SMTP

You first need to configure the SMTP backend parameters:

```
backend:
    class: OpenXPKI::Server::Notification::SMTP
    host: localhost
    port: 25
    username: smtpuser
    password: smtpsecret
    debug: 0
    use_html: 0
```

Class is the only mandatory parameter, the default is localhost:25 without authentication. Debug enables the Debug option from Net::SMTP writing to the stderr.log which can help you to test/debug mail delivery. To use html formatted mails, you need to install *MIME::Lite* and set *use_html: 1*. The handler will fall back to plain text if MIME::Lite can not be loaded.

The mail templates are read from disk from, you need to set a base directory:

```
template:
    dir:   /home/pkiadm/ca-one/email/
```

Below is the complete message configuration as shipped with the default issuance workflow:

```
default:
    from: no-reply@mycompany.com
    reply: helpdesk@mycompany.com
    to: "[% cert_info.requestor_email %]"
    cc: helpdesk@mycompany.com

message:
    csr_created:   # The message Id as referenced in the activity
        user:   # The internal handle for this thread
            template: csr_created_user
            subject: CSR for [% cert_subject %]
            prefix: PKI-Ticket [% meta_wf_id %]
            images:
                banner: head.png
```

```
                footer: foot.png

        raop:       # Another internal handle for a second thread
            template: csr_created_raop  # Suffix .txt is always added!
            to: reg-office@mycompany.com
            cc: ''
            reply: "[% cert_info.requestor_email %]"
            subject: CSR for [% cert_subject %]

    csr_rejected:
        user:
            template: csr_rejected
            subject: CSR rejected for [% cert_subject %]

    cert_issued:
        user:
            template: cert_issued
            subject: certificate issued for [% cert_subject %]
```

The *default* section is not necessary but useful to keep your config short and readable. These options are merged with the local ones, so any local variable is possible and you can overwrite any default at the local configuration (to clear a setting use an empty string, the images hash is NOT merged recursively).

**the idea of threads**

You might have recognized that there are two blocks below `messages.csr_created`. Those are so called *threads*, which combine messages sent at different times to share some common settings. With the first message of a thread the values given for to, cc and prefix are persisted so you can ensure that all messages that belong to a certain thread go to the same receipients using the same subject prefix. **Note, that settings to those options in later messages are ignored!**

**receipient information**

The primary receipient and a from address are mandatory:

- to: The primary receipient, single value, parsed using TT
- from: single value, NOT parsed

Additional receipients and a seperate Reply-To header are optional:

- cc: comma seperated list, parsed using TT
- reply: single value, NOT parsed

All values need to be rfc822 compliant full addresses.

**composing the subject**

The subject is parsed using TT. If you have specified a prefix, it is automatically prepended.

**composing the message body**

The body of a message is read from the filename specified by *template*, where the suffix '.txt' is always apppended. So the full path for the message at `messages.csr_created.user` is */home/pkiadm/ca-one/email/csr_created_user.txt*.

**html messages**

If you use the html backend, the template for the html part is read from *csr_created_user.html*. It is allowed to provide either a text or a html template, if both files are found you will get a multipart message with both message parts set. Make sure that the content is the same to avoid funny issues ;)

It is possible to use inline images by listing the image files with the *images* key as key/value list. The key is the internal identifier, to be used in the html template, the value is the name of the image file on disk.

With a config of:

```
user:
    template: csr_created_user
    ....
    images:
        banner: head.png
        footer: foot.png
```

You need to reference the image in the html template like this:

```
<body>
    <img src="cid:banner" title="My Company Logo Banner" />
    .....
    <img src="cid:footer" title="My Company Logo Footer" />
</body>
```

The images are pulled from the folder *images* below the template directory, e.g. */home/pkiadm/ca-one/email/images/head.png*. The files must end on gif/png/jpg as the suffix is used to detect the correct image type.

### RT Request Tracker

The RT handler can open, modify and close tickets in a remote RT system using the REST interface. You need to install RT::Client::REST from CPAN and setup the connection:

```
backend:
    class: OpenXPKI::Server::Notification::RT
    server: http://rt.mycompany.com/
    username: pkiuser
    password: secret
    timeout: 30
```

The timeout value is optional with a default of 30 seconds.

As the SMTP backend, it uses templates on disk to build the ticket contents, so we also need to set the template directory:

```
template:
    dir:   /home/pkiadm/ca-one/rt/
```

You can share the templates for SMTP and RT handler and reuse most parts of your configuration, but note that the syntax is slightly different from SMTP. Here is the complete message configuration as shipped with the default issuance workflow:

```
message:
    csr_created:  # The message Id as referenced in the activity
        main:     # The internal handle for this ticket
            - action: open
              queue: PKI
              owner: pki-team
              subject: New CSR for [% cert_subject %]
              to: "[% cert_info.requestor_email %]"
              template: csr_created
              priority: 1

            - action: comment
              template: csr_created_comment
              status: open

    csr_approved:
        main:
            - action: update
              status: working
```

```
    csr_rejected:
        main:
            - action: correspond
              template: csr_rejected
              priority: 10

    cert_issued:
        main:
            - action: comment
              template: cert_issued_internals

            - action: correspond
              template: cert_issued
              status: resolved
```

The RT handler also makes use of threads, where each thread is equal to one ticket in the RT system. The example uses only one thread = one ticket. Each message can have multiple threads and each thread consists of at least one action.

**Create a new ticket**

You should make sure that a ticket is created before you work with it! The minimum information required to open a ticket is:

```
action: open
queue: PKI
owner: pki-team
subject: New CSR for [% cert_subject %]
to: "[% cert_info.requestor_email %]"
```

The *to* field must be an email address, which is used to fill the *requestor* field in RT.

Additional fields are:

- cc: comma sep. list of email addresses to be assigned to the ticket, parsed with TT
- template: filename for a TT template, used as inital text for the ticket (.txt suffix is added)
- priority: priority level, usually a numeric value
- status: ticket status, usually one of “new”, “open”, “resolved”, “stalled”, “rejected”, and “deleted”.

**comment or correspond to a ticket**

The maximum configuration is:

```
action:   comment  # or "correspond"
status:   open     # optional
priority: 5        # optional
template: csr_created_comment  # .txt is added
```

For *comment* the result of the parsed template is added to the ticket history.

For *correspond* the result is also mailed to the ticket receipients (this is a feature of RT, we dont send any mails).

Note: If the template parser returns an empty string, no operation is done on the ticket.

**update status/priority without text**

The *update* action allows you to set status/priority without creating a text entry in the history:

```
action: update
status: stalled
priority: 0
```

You can call update with either status or priority or both.

**setting custom fields**

You can set custom field values using the update action. Any key/value pair in the block (except the ones above) is considered to be a custom field. The values are parsed using TT:

```
action: update
priority: 3
custom-field1: My custom value
custom-field2: My other custom value
```

Note: This feature is untested!

**closing a ticket**

You can close a ticket with the above commands by setting the status-flag. For convenience there is a shortcut, setting the status to "resolved":

```
action: close
```

### Template Variables

The notification handler injects those values into the template parser on any invocation.

**realm info**

- meta_pki_realm (key of the current realm)
- meta_label (verbose realm name as defined at `system.realms.$realm.label`)
- meta_baseurl (baseurl as defined at `system.realms.$realm.baseurl`)

**request related context values (scalars)**

- csr_serial
- cert_subject
- cert_identifier
- cert_profile

**request related context values (hashes)**

- cert_subject_parts
- cert_subject_alt_name
- cert_info
- approvals

**misc**

- creator
- requestor (real name of the requestor, if available assembled from cert_info.requestor_gname + requestor_name, otherwise the word "unknown")

**Certificate Info Plugin**

The default install also provides a plugin to get detailed informations on a certificate:

```
[% USE Certificate %]

Serial: [% Certificate.serial(cert_identifier) %]
Hex Serial: [% Certificate.serial_hex(cert_identifier) %]
CSR: [% Certificate.csr_serial(cert_identifier) %]
Issuer: [% Certificate.issuer(cert_identifier) %]
Status: [% Certificate.status(cert_identifier) %]

Body-Subject: [% Certificate.body(cert_identifier, 'Subject') %]
```

The body method will return any field of the body structure offered by the get_cert api method. Fore further info check the modules documentation (OpenXPKI::Template::Plugin::Certificate).

### Workflow

The definition of the workflows is still in the older xml format, already used in older OpenXPKI releases but its management is included into the connector now. The XML files are located in the folder named *_workflow* (**note the underscore!**) in the top level direcotry of the realm. If you are upgrading from an older installation, you can just move your old workflow*.xml files here *and* add an outer "openxpki" tag to the *workflow.xml* file.

## Configuration of the Workflow Engine with UI

All files mentioned here are below the node `workflow` inside the realm configuration. Filenames are all lowercased.

### Workflow Definitions

Each workflow is represented by a file or directory structure below `workflow.def.<name>`. The name of the file is equal to the internal name of the workflow. Each such file must have the following structure, not all attributes are mandatory or useful in all situations:

```
head:
    label: The verbose name of the workflow, shown on the UI
    description: The verbose description of the workflow, shown on the UI
    prefix: internal short name, used to prefix the actions, must be unique.

state:
    name_of_state:  (used as literal name in the engine)
        autorun: 0/1
        autofail: 0/1
        label: visible name
        description: the text for the page head
        action:
          - name_of_action > state_on_success ? condition_name
          - name_of_other_action > other_state_on_success !condition_name
        hint:
            name_of_action: A verbose text shown aside of the button
            name_of_other_action: A verbose text shown aside of the button

action:
    name_of_action: (as used above)
        label: Verbose name, shown as label on the button
        tooltip: Hint to show as tooltip on the button
        description: Verbose description, show on UI page
        class: Name of the implementation class
        abort: state to jump to on abort (UI button, optional) # not implemented yet
        resume: state to jump to on resume (after exception, optional) # not implemented yet
        validator:
          - name_of_validator (defined below)
        input:
          - name_of_field (defined below)
          - name_of_other_field
        param:
            key: value - passed as params to the action class

field:
    field_name: (as used above)
        name: key used in context
```

```
        label: The fields label
        placeholder: Hint text shown in empty form elements
        tooltip: Text for "tooltip help"
        type:     Type of form element (default is input)
        required: 0|1
        default:  default value
        more_key: other_value  (depends on form type)

validator:
    class: OpenXPKI::Server::Workflow::Validator::CertIdentifierExists
    param:
        emptyok: 1
    arg:
      - $cert_identifier
```

Note: All entity names must contain only letters (lower ascii), digits and the underscore.

Below is a simple, but working workflow config (no conditions, no validators, the global action is defined outside this file):

```
head:
    label: I am a Test
    description: This is a Workflow for Testing
    prefix: test

state:
    INITIAL:
        label: initial state
        description: This is where everything starts
        action: run_test1 > PENDING

    PENDING:
        label: pending state
        description: We hold here for a while
        action: global_run_test2 > SUCCESS

    SUCCESS:
        label: finals state
        description: It's done - really!


action:
    run_test1:
    label: The first Action
    description: I am first!
    class: Workflow::Action::Null
    input: comment
    param:
        message: "Hi, I am a log message"

field:
    comment: (as used above)
        name: comment
        label: Your Comment
        placeholder: Please enter a comment here
        tooltip: Tell us what you think about it!
        type: textarea
        required: 1
        default: ''
```

### Workflow Head

### States

The `action` attribute is a list (or scalar) holding the action name and the follow up state. Put the name of the action and the expected state on success, seperated by the > sign (is greater than).

### Action

### Field

*Select Field with options*

type: select option:

**item:**

- unspecified
- keyCompromise
- CACompromise
- affiliationChanged
- superseded
- cessationOfOperation

label: I18N_OPENXPKI_UI_WORKFLOW_FIELD_REASON_CODE_OPTION

If the label tag is given (below option!), the values in the drop down are i18n strings made from label + uppercase(key), e.g I18N_OPENXPKI_UI_WORKFLOW_FIELD_REASON_CODE_OPTION_UNSPECIFIED

## UI Rendering

The UI uses information from the workflow definition to render display and input pages. There are two different kinds of pages, switches and inputs.

### Action Switch Page

Used when the workflow comes to a state with more than one possible action.

*headline*

Concated string from state.label + workflow.label

*descriptive intro*

String as defined in state.description, can contain HTML tags

*workflow context*

By default a plain dump of the context using key/values, array/hash values are converted to a html list/dd-list. You can define a custom output table with labels, formatted values and even links, etc - see the section "Workflow Output Formatting" fore details.

*button bar / simple layout*

One button is created for each available action, the button label is taken from action.label. The value of action.tooltip becomes a mouse-over label.

*button bar / advanced layout*

If you set the state.hint attribute, each button is drawn on its own row with a help text shown aside.

#### Form Input Page

Used when the workflow comes to a state where only one action is available or where one action was choosen.

*headline*

Concated string from action.label (if none is given: state.label ) + workflow.label

*descriptive intro*

String as defined in action.description, can contain HTML tags

*form fields*

The field itself is created from label, placeholder and tooltip. If at least one form field has the description attribute set, an explanatory block for the fields is added to the bottom of the page.

### Global Entities

You can define entities for action, condition and validator for global use in the corresponding files below `workflow.global.`. The format is the same as described below, the "**global_**" prefix is added by the system.

### Creating Macros (not implemented yet!)

If you have a sequence of states/actions you need in multiple workflows, you can define them globally as macro. Just put the necessary state and action sections as written above into a file below `workflow.macros.<name>`. You need to have one state named `INITIAL` and one `FINAL`.

To reference such a macro, create an action in your main workflow and replace the `class` atttribute with `macro`. Note that this is NOT an extension to the workflow engine but only merges the definitions from the macro file with those of the current workflow. After successful execution, the workflow will be in the state passed in the `success` attribute ofthe surrounding action.

## Workflow Output Formatting

### Buttons

### Key/Value Grid

#### redirect

Creates an immediate redirect command with value as location. The location must be an ember route, e.g. workflow!start!system_status.

## Profile Configuration

A certificate profile is the blueprint that determines all technical aspects of the certificate such as subject pattern, key usages and other extensions.

## Naming

The internal name of the profile is the name of the node in the configuration layer. If you keep the sample structure each profile is in a single file in the profile directory, so the name of the profile is the name of the file.

You can add *label* and *description* to the profile, which is used for display purpose on the WebUI frontend only, it has no effect on the actual certificate.

## Validity

The validity is usually defined by a relative time specification of the format +YYMMDDhhmmss, (e.g. +0006 for six month, see OpenXPKI::DateTime).

**validity:** notafter: +0006

The actual value is determined at the moment the PKI really signs the request. You can also add a notbefore date, in this case the notafter date is calculated relative to notbefore!

**validity:** notafter: +000001 notafter: +0006

Above example will create a certificate with a notbefore 24 hours ahead of the time of issuance and ends 6 months + 1 day later.

It is also possible to give an absolute date as YYYYMMDDhhmmss.

## Styles

t.b.d

## Subject and Process Information

OpenXPKI can collect meta information based on the selected profile and has a templating engine to build subject and subject alternative name sections (SAN) from the input data in many different ways.

The input fields are summarized in three groups: subject, san and info:

```
00_basic_style:
    label: I18N_OPENXPKI_UI_PROFILE_BASIC_STYLE_LABEL
    description: I18N_OPENXPKI_UI_PROFILE_BASIC_STYLE_DESC
    ui:
        subject:
            - hostname
            - hostname2
            - port
        san:
            - san_ipv4
        info:
            - requestor_gname
            - requestor_name
            - requestor_email
            - requestor_affiliation
            - comment
```

Each section can hold any number of fields, each field is defined by a set of options:

```
id: hostname
label: I18N_OPENXPKI_UI_PROFILE_HOSTNAME
description: I18N_OPENXPKI_UI_PROFILE_HOSTNAME_DESC
type: freetext
preset: "[% CN.0.replace(':.*','') %]"
match: \A [A-Za-z\d\-\.]+ \z
```

```
width: 60
default: fully.qualified.example.com
```

The definition can be placed in the node *template* inside the profile file or globally in the template directory.

#### Field Definition

**id** the key used when this item is written into the workflow

**label** the label shown next to the input field

**description** description, shown as tooltip

**type** the type of the field, freetext or select

**option** list of options for the select field (used value is equal to the label)

**preset** in case a CSR is uploaded by the user, you can use parts of it to prefill the fields. Items from the subject can be referenced by the name of the component, items from the subject alternative name section are prefixed with the string *SAN_*, e.g. *SAN_DNS*. Note that all keys are uppercased and all items are arrays regardless of the number of items found!

There are several options for preprocessing the items.

First item of type CN:

```
preset: CN.0
```

All items of type OU (creates on field per item):

```
preset: OU.X
```

Use templating to extract left side of CN up to the first colon:

```
preset: "[% CN.0.replace(':.*','') %]"
```

Use templating to create a list of items, the pipe symbol is used as seperator:

```
preset: "[% FOREACH ou = OU %][% ou %]|[% END %]"
```

**match** a regex pattern that is applied to the user input for validation

**width** size of the field - not implemented yet, definition might change.

**default** A text which is shown as placeholder in the input field (this value is NOT a default value for the field)

#### Subject Rendering

### Extensions

## Configuration of the SCEP Workflow

Before you can use the SCEP subsystem, you need to enable the SCEP Server in the general configuration. This section explains the options for the scep enrollment workflow.

### Workflow Logic

The workflow validates incoming requests against five stages. Parameters are described in detail in the section on policy settings.

**technical parameters:** Check if key algorithm, key size and hash algorithm match the policy. If any of those checks fails, the request is rejected.

**authentication:** A request can either be self-signed and provide a challenge password or is signed by a trusted certificate (renewal or "signer on behalf"). You can also disable authentication or dispatch unauthorized requests to an operator for review.

**subject duplicate check:** The database is checked for valid certificates with the same subject. If issuing the certificate would exceed the configured maximum count, the request is dispatched to an operator wo can either reject the request or take actions to meet the policy.

**eligibility:** The basic idea is to check requests based on the subject or additonal parameters against an external source to see if enrollment is possible. The check counts against the approval point counter, the workflow does not take any special action if the check fails.

**approval point:** The first four stages are usually run in one step when the request hits the server. Before the certificate is issued, the request must have a sufficient number of approval points. Each operator approval is worth one point. A passed eligibility check is also worth one.

## Sample Configuration

The workflow fetches all information from the configuration system at `scep.<servername>` where the servername is taken from the scep wrapper configuration.

Here is a complete sample configuration:

```
key_size:
    rsaEncryption: 1020-2048

hash_type:
  - sha1

authorized_signer_on_behalf:
    technicans:
        subject: CN=.*DC=SCEP Signer CA,DC=mycompany,DC=com
        profile: I18N_OPENXPKI_PROFILE_SCEP_SIGNER
    blackbox:
        identifier: JNHN5Hnje34HcltluuzooKVqxss

challenge:
   value: SecretChallenge

renewal_period: 000014
replace_period: 05
revoke_on_replace:
    reason_code: keyCompromise
    invalidity_time: +000014

eligible:
    initial:
       value: 0
    renewal:
       value: 1

policy:
    allow_anon_enroll: 0
    allow_man_authen: 1
    allow_man_approv: 1
    max_active_certs: 1
    allow_expired_signer: 0
    auto_revoke_existing_certs: 1
    approval_points: 1

response:
    # The scep standard is a bit unclear if the root should be in the chain or not
    # We consider it a security risk (trust should be always set by hand) but
```

```
    # as most clients seem to expect it, we include the root by default
    # If you are sure your clients do not need the root, set this to 1
    getcacert_strip_root: 0

# Mapping of names to OpenXPKI profiles to be used with the
# Microsoft Certificate Template Name Ext. (1.3.6.1.4.1.311.20.2)
profile_map:
    pc-client: I18N_OPENXPKI_PROFILE_USER_AUTHENTICATION

subject_style: enroll

token: ca-one-special-scep

workflow_type: enrollment
```

*The renewal and replace period values are interpreted as OpenXPKI::DateTime relative date but given without sign.*

## Workflow Configuration

### technical validation

Configure the list of allowed key and hash algorithms.

**key_size**

A hash item list for allowed key sizes and algorithms. The name of the option must be the key algorithm as given by openssl, the required byte count is given as a range in bytes. There must not be any space between the dash and the numbers. Hint: Some implementations do not set the highest bit to 1 which will result in a nominal key size which is one bit smaller than the requested one. So stating a small offset here will reduce the propability to reject such a key.

**hash_type**

List (or single scalar) of accepted hash algorithms used to sign the request.

### Authentication

#### Signer on Behalf

The section *authorized_signer_on_behalf* is used to define the certificates which are accepted to do a "request on behalf". The list is given as a hash of hashes, were each entry is a combination of one or more matching rules.

Possible rules are subject, profile and identifier which can be used in any combination. The subject is evaluated as a regexp against the signer subject, therefore any characters with a special meaning in perl regexp need to be escaped! Identifier and profile are matched as is. The rules in one entry are ANDed together. If you want to provide alternatives, add multiple list items. The name of the rule is just used for logging purpose.

#### Challenge Password

The request must carry the password in the challengePassword attribute. The sample config above shows a static password example but it is also possible to use request parameters to lookup a password using connectors:

```
challenge:
   mode: bind
   value@: connector:scep.connectors.challenge
   args:
   - "[% context.cert_subject %]"
```

```
connectors:
    challenge:
        class: Connector::Builtin::Authentication::Password
        LOCATION: /home/pkiadm/ca-one/passwd.txt
```

This will use the cert_subject to validate the given password against a list found in the file /home/pkiadm/ca-one/passwd.txt. For more details, check the man page of OpenXPKI::Server::Workflow::Activity::SCEPv2::EvaluateChallenge

#### Renewal/Replace

A request is considered to be a renewal if the request is *not* self-signed but the signer subject matches the request subject. Renewal requests pass authentication if the signer certificate is valid in the current realm and neither revoked nor expired. You can allow expired certificates by setting the *allow_expired_signer* policy flag.

#### Manual Authentication

If you set the *allow_man_authen* policy flag, request that fail any of the above authentication methods can be manually authenticated via the UI.

#### No Authentication

To completly skip authentication, set *allow_anon_enroll* policy flag.

### Subject Checking

The policy setting *max_active_certs* gives the maximum allowed number of valid certificates sharing the same subject. If the certificate count after issuance of the current request will exceed this number, the workflow stops in the POLICY_PENDING state. There are several settings that influence this check, based on the operation mode:

#### Initial Enrollment

If you set the *auto_revoke_existing_certs* policy flag, all certificates with the same subject *will be revoked* prior to running this check. This does not make much sense with *max_active_certs* larger than 1 as all certificates will be revoked as soon as a new enrollment is started! The intended use is replacement of broken systems where the current certificate is no longer used anyway.

#### Renewal

If the certificate used to sign the renewal (see authentication) expires within the period specified by *renewal_period*, it is not counted against the limit.

#### Replace

Same as renewal based on the *replace_period* parameter. See below for an explanation of the *revoke_on_replace* feature.

### Eligibility

The default config has a static value of 1 for renewals and 0 for initial requests. If you set *approval_points* to 1, this will result in an immediate issue of certificate renewal requests but requires operator approval on initial enrollments.

Assume you want to use an ldap directory to auto approve initial requests based on the mac address of your client:

```
eligible:
    initial:
        value@: connector:your.connector
        args:
        - "[% context.cert_subject %]"
        - "[% context.url_mac %]"

connectors:
    devices:
        ## This connector just checks if the given mac
        ## exisits in the ldap
        class: Connector::Proxy::Net::LDAP::Simple
        LOCATION: ldap://localhost:389
        base: ou=devices,dc=mycompany,dc=com
        filter: (macaddress=[% ARGS.1 %])
        binddn: cn=admin,dc=mycompany,dc=com
        password: admin
        attrs: macaddress
```

To have the mac in the workflow, you need to pass it with the request as an url parameter to the wrapper: *http://host/scep/scep?mac=001122334455*.

For more options and samples, see the perldoc of OpenXPKI::Server::Workflow::Activity::SCEPv2::EvaluateEligibility

### Approval

A request is approved if it reaches the number of approvals defined by the *approval_points* policy setting. As written above, you can use a data source to get one approval point via the eligibility check. If a request has an insufficient number of approvals, the workflow will stop and an operator must give an approval using the WebUI. By raising the approval points value, you can also enforce a four-eyes approval. If you do not want manual approvals, set the policy flag *allow_man_approv* to zero - all requests that fail the eligibility check will be immediately terminated.

## Certificate Configuration

### SCEP Server Token

This is the cryptographic token used to sign and decrypt the SCEP communication itself. It is not related to the issuing process of the requested certificates!

The crypto configuration of a realm (crypto.yaml) defines a default token to be used for all scep services inside this realm. In case you want different servers to use different certificates, you can add additional token groups and reference them from the config using the *token* key.

The value must be the name of a token group, which needs to be registered as an anonymous alias:

```
openxpkiadm alias --realm ca-one --identifier <identifier> --group ca-one-special-scep --gen 1
```

Note that you need to care yourself about the generation index. The token will then be listed as anonymous group item:

```
openxpkiadm alias --realm ca-one

=== anonymous groups ===
ca-one-special-scep:
  Alias     : ca-one-special-scep-1
  Identifier: O9vtjge0wHpYhDpfko2O6xYtCWw
  NotBefore : 2014-03-25 15:26:18
  NotAfter  : 2015-03-25 15:26:18
```

### Profile Selection / Certificate Template Name Extension

This feature was originally introduced by Microsoft and uses a Microsoft specific OID (1.3.6.1.4.1.311.20.2). If your request contains this OID **and** the value of this oid is listed in the profile map, the workflow will use the given profile definition to issue the certificate. If no OID is present or the value is not in the map, the default profile from the server configuration is used.

The map is a hash list:

```
profile_map:
    tlsv2: I18N_OPENXPKI_PROFILE_TLS_SERVER_v2
    client: I18N_OPENXPKI_PROFILE_TLS_CLIENT
```

### Subject Rendering

By default the received csr is used to create the certificate "as is". To have some sort of control about the issued certificates, you can use the subject rendering engine which is also used with the frontend by setting a profile style to be used:

    subject_style: enroll

The subject will be created using Template Toolkit with the parsed subject hash as input vars. The vars hash will use the name of the attribute as key and pass all values as array in order of appearance (it is always an array, even if the attribute is found only once!). You can also add SAN items but there is no way to filter or remove san items that are passed with the request, yet.

Example: The default TLS Server profile contains an enrollment section:

```
enroll:
    subject:
        dn: CN=[% CN.0 %],DC=Test Deployment,DC=OpenXPKI,DC=org
```

The issued certificate will have the common name extracted from the incoming request but get the remaining path compontens as defined in the profile.

### Revoke on Replace

If you have a replace request (signed renewal with signer validity between replace_window and renew_window), you can trigger the automatic revocation of the signer certificate. Setting a reason code is mandatory, supported values can be taken from the openssl man page (mind the CamelCasing), the invalidity_time is optional and can be relative or absolute date as consumed by OpenXPKI::DateTime, any empty value becomes "now":

```
revoke_on_replace:
    reason_code: superseded
    invalidity_time: +000002
```

The above gives your friendly admins a 48h window to replace the certificates before they show up on the next CRL. It also works the other way round - assume you know a security breach happend on the seventh of april and you want to tell this to the people:

```
revoke_on_replace:
    reason_code: keyCompromise
    invalidity_time: 20140407
```

Note: Without any other measures, this will obviously enable an attacker who has access to a leaked key to obtain a new certificate. We used this to replace certificates after the Heartbleed bug with the scep systems seperated from the public network.

## Misc

**workflow_type**

The name of the workflow that is used by this server instance.

**response.getcacert_strip_root**

The scep standard is a bit unclear if the root should be in the chain or not. We consider it a security risk (trust should be always set by hand) but as most clients seem to expect it, we include the root by default. If you are sure your clients do not need the root and have it deployed, set this flag to 1 to strip the root certificate from the getcacert response.

## The workflow context

The workflow uses status flags in the context to take decissions. Flags are boolean if not stated otherwise. This is intended to be a debugging aid.

**csr_key_size_ok**

Weather the keysize of the csr matches the given array. If the key_size definition is missing, the flag is not set.

**have_all_approvals**

Result of the approval check done in CalcApproval.

**in_renew_window**

The request is within the configured renewal period.

**num_manual_authen**

The number of given manual authentications. Can override missing authentication on initial enrollment.

**scep_uniq_id_ok**

The internal request id is really unique across the whole system.

**signer_is_self_signed**

The signer and the csr have the same public key. Note: If you allow key renewal this might also be a renewal!

**signer_authorized**

The signer certificate is recognized as an authorized signer on behalf. See *authorized_signer_on_behalf* in the configuration section.

**signer_signature_valid**

The signature on the PKCS#7 container is valid.

**signer_sn_matches_csr**

The request subject matches the signer subject. This can be either a self-signed initial enrollment or a renewal!

**signer_status_revoked**

The signer certificate is marked revoked in the database.

**signer_trusted**

The PKI can build the complete chain from the signer certificate to a trusted root. It might be revoked or expired!

**signer_validity_ok**

The notbefore/notafter dates were valid at the time of validation. In case you have a grace_period set, a certificate is also valid if it has expired within the grace period.

**valid_chall_pass**

The provided challenge password has been approved.

**valid_kerb_authen**

Request was authenticated using kerberos (not implemented yet)

**csr_profile_oid**

The profile name as extracted from the Certificate Type Extension (Microsoft specific)

# WebUI Page API Reference

The web pages are created (mainly) on the client from a JSON control stucture delivered by the server. This document describes the structure expected by the rendering engine.

## Top-Level Structure

This is the root element of any json result:

```
%structure = (

    page => { TOP_LEVEL_INFO},

    right => [ PAGE_SECTION, PAGE_SECTION,...] , # optional, information which will be displayed

    main => [ PAGE_SECTION, PAGE_SECTION,...] , # information which will be displayed in the main

    reloadTree => BOOL (1/0), # optional, the browser will perform a complete reload. If an addit

    goto => STRING PAGE, # optional, will be evaluated as url-hashtag target

    status => { STATUS_INFO } # optional
);

Example { reloadTree => 1, goto => 'login/login'}
```

### Page Head (TOP_LEVEL_INFO):

This is rendered as the page main headline and intro text.

```
TOP_LEVEL_INFO:
{
    label => STRING, #Page Header
    description => STRING, # additional text (opt.)
}

Example: page => {label => 'OpenXPKI Login', description => 'Please log in!'}
```

### Status Notification (STATUS_INFO):

Show a status bar on top of the page, the level indicates the severity and results in different colors of the status bar.

```
STATUS_INFO:
{
    level => STRING, # allowed values: "info", "success","warn", "error"
    message => STRING # status message shown
}

Example:   status => { level => 'error', message => 'Login credentials are wrong!' }
```

## Page Level

The page sections (`main` and `right`) can hold multiple subpage definitions. The main section must always contain at least one section while right can be omitted or empty.

### Page Section (PAGE_SECTION)

This is the top level container of each page section.

```
PAGE_SECTION:
{
    type => STRING # determines type of section, can be one of: text|grid|form|keyvalue

    content => {
        label => STRING # optional, section headline

        description => STRING , # optional, additional text (html is allowed)

        buttons => [BUTTON_DEF, BUTTON_DEF, BUTTON_DEF] , #optional, defines the buttons/links fo

        # additional content-params depending on type (see below)
    },

    # additional section-params depending on type:
}
```

### SECTION-TYPE “text”

Print the label as subheadline (h2) and description as intro text, buttons are rendered after the text. Does not have any additional parameters. Note: If you omit label and description this can be used to render a plain button bar or even a single button.

### SECTION-TYPE “grid”

Grids are rendered using the jquery datatable plugin (http://datatables.net). The grid related parameters are just pushed to the dataTables engine and therefore have a different notation and syntax used as the remainder of the project.

```
content => {
    label => .., description => .., buttons => ..,
    columns => [ GRID_COL_DEF, GRID_COL_DEF , GRID_COL_DEF... ],
    data => [ GRID_ROW, GRID_ROW, GRID_ROW, ... ],
    actions => [ GRID_ACTION_DEF, GRID_ACTION_DEF, GRID_ACTION_DEF... ], # defines available acti
    processing_type => STRING, # only possible value (for now) is "all"
}

GRID_COL_DEF:
{
    sTitle => STRING, # displayed title of that columnd AND unique key
    format => STRING_FORMAT # optional, triggers a formatting helper (see below)
```

```
}

GRID_ROW:
    ['col1','col2','col3']


GRID_ACTION_DEF:
{
    path => STRING_PATH, # will be submitted to server as page. terms enclosed in {brackets} will
    label => STRING, # visible menu entry
    target => STRING_TARGET # optional, where to open the new page, one of main|right|modal|tab
    icon => STRING , # optional, file name of image icon, must be placed in htdocs/img/contextmenu
}
```

Columns, whose sTitle begin with an underscore will not be displayed but used as internal information (e.g. as path in GRID_ACTION_DEF). A column with the special title `_status` is used as css class for the row. Also a pulldown menu to filter by status will be displayed. The rows hold the data in form of a positional array.

Action target `modal` creates a modal popup, `tab` inits or extends a tabbed window view in the current section.

*Example*:

```
content => {
    columns => [
        { sTitle => "Serial" },
        { sTitle => "Subject" },
        { sTitle => "date_issued", format => 'timestamp'},
        { sTitle => "link", format => 'link'},
        { sTitle => "_id"}, # internal ID (will not be displayed)
        { sTitle => "_status"}, # row status
    ],
    data => [
        ['0123','CN=John M Miller,DC=My Company,DC=com',1379587708, {page => 'http://../', label =
        ['0456','CN=Bob Builder,DC=My Company,DC=com',1379587517,{...},'qqA2H','expired'],
    ],
    actions => [
        {
            path => 'cert!detail!{_id}',
            label => 'Details',
            icon => 'view',
            target => 'modal'
        },
        {
            path => 'cert!mail2issuer!{email}',
            label => 'Send an email to issuer'
        },
    ]
}
```

### SECTION-TYPE "form"

Render a form to submit data to the server

```
content => {
    label => .., description => ..,
    buttons => [ ... ], # a form must contain at least one button to be useful
    fields => [ FORM_FIELD_DEF,FORM_FIELD_DEF,FORM_FIELD_DEF ],
}

FORM_FIELD_DEF:
{
    name => STRING # internal key - will be transmitted to server
```

```
    value => MIXED, # value of the field, scalar or array (depending on type)
    label => STRING, # displayed label
    type => STRING_FIELD_TYPE, # see below for supported field types
    is_optional => BOOL, # if false (or not given at all) the field is required
    clonable => BOOL,  creates fields that can be added more than once
    visible => BOOL, #if set to "false" ("0" in perl) this field will be not displayed (initial)
    keys => ARRAY ,#optional, activates the special feature of "dynamic key value fields", see bel
    # + additional keys depending for some types
}
```

#### Field-Type “text”, “hidden”, “password”, “textarea”

No additional parameters, create a simple html form element without any extras.

#### Field-Type “checkbox/bool”

A html checkbox, `value` and `is_optional` are without effect, as always 0 or 1 is send to the server.

#### Field-Type “date”

A text field with a jquery datapicker attached. Additional (all optional) params are:

```
FORM_FIELD_DEF:
{
    notbefore => INTEGER, # optional, unixtime, earliest selectable date
    notafter => INTEGER, # optional, unixtime, earliest selectable date
    return_format => STRING # one of terse|printable|iso8601|epoch, see OpenXPKI::Datetime
}
```

#### Field-Type “select”

A html select element, the options parameter is required, others are optional:

```
FORM_FIELD_DEF:
{
    options => [{value=>'key 1',label=>'Label 1'},{value=>'key 2',label=>'Label 2'},...],
    prompt => STRING # first option shown in the box, no value (soemthing like "please choose")
    editable => BOOL # activates the ComboBox,
    actionOnChange => STRING_ACTION # if the pulldown is changed by the user (or an initial value
}
```

The `options` parameter can be fetched via an ajax call. If you set `options => 'fetch_cert_status_options'`, an ajax call to “server_url.cgi?action=fetch_cert_status_options” is made. The call must return the label/value list as defined given above.

Setting the editable flag to a true value enables the users to enter any value into the select box (created with Bootstrap Combobox).

#### Field-Type “radio”

The radio type is the little brother of the select field, but renders the items as a list of items using html radio-buttons. It shares the syntax of the `options` field with the select element:

```
FORM_FIELD_DEF:
{
    options => [{....}] or 'ajax_action_string'..
```

```
    multi => BOOL, # optional, if true, uses checkbox elements instead radio buttons
}
```

#### Field-Type "upload"

Renders a field to upload files with some additional benefits:

```
FORM_FIELD_DEF:
{
    mode => STRING, # one of hidden, visible, raw
    allowedFiles => ARRAY OF STRING, # ['txt', 'jpg'],
    textAreaSize => {width => '10', height => '15'},
}
```

By default, a file upload button is shown which loads the selected file into a hidden textarea. Binary content is encoded with base64 and prefixed with the word "binary:". With *mode = visible* the textarea is also shown so the user can either upload or paste the data (which is very handy for CSR uploads), the textAreaSize will affect the size of the area field. With `mode = raw` the element degrades to a html form upload button and the selected file is send with the form as raw data.

AllowedFiles can contain a list of allowed file extensions.

#### Dynamic key value fields

If a field is defined with the property "keys", a pulldown of options is displayed above the actual field. This allows the user to specify, which kind of information he wants to specify. The content of the actual field will be submitted to the server with the selected key in the key-pulldown.

**Example:** { name => '...', label => 'Dyn Key-Value', 'keys' => [{value=>"key_x",label=>"Typ X"},{value=>"key_y",label=>"Typ Y"}], type => 'text' },

This example definition will render a Textfield with label "Dyn Key-Value". Above the textfield a select is displayed with three options ("Typ x","Typ y" and "Typ z"). If the user chooses "Typ Z", the entered value in the textfield will be posted to server with key "key_z".

This feature makes often more sense in combination with "clonable" fields.

#### Dynamic form rendering

If a select field is defined with the property "actionOnChange", each change event of this pulldown will trigger an submit of all formvalues (without validity checks etc) to the server with key "action" set to the value of "actionOnChange".

The returned JSON must contain the key "_returnType" which should have the value "partial" or "full". This "_returnType" defines the mode of re-definition of the content of the form.

**Partial redefinition:** Beside the key "_returntype" the key "fields" is expected in the returned JSON-Structure. "fields" contains an array, which is semantically identic to the key "fields" in the definition of the form. This array "fields" must contain only only the fields (and properties), which should react to the change of the (master-)field (pulldown) . The property "name" is required (otherwise the client can not identify the field). The property "type" can not be subject to changes. With aid of the property "visible" one can dynamically show or hide some fields. Only known fields (which are already defined in the initial "fields"-property of the form-section) can be subject of the "partial" re-rendering. Its not possible to add new fields here.

You can define more than one (cascading) dependent select.

*Example*:

Initial definition of fields: fields => [

{ name => 'cert_typ', label => 'Typ',value=> 't2', prompt => 'please select a type', type => 'select', actionOnChange => 'test_dep_select!change_type', options=>[{value=>'t1',label=>'Typ 1'},{value=>'t2',label=>'Typ 2'},{value=>'t3',label=>'Typ 3'}] }, { name => 'cert_subtyp', label => 'Sub-Type',prompt => 'first select type!', type => 'select',options=>[] },

{ name => 'special', label => 'Spezial (nur Typ 2', type => 'checkbox',visible => 0 },

]

Action "test_dep_select!change_type" returns a (partially updated) definition of fields

{ _returnType => 'partial', fields => [

{ name => 'cert_subtyp', options=> [{value=>'x', label => 'Subtyp X'},...],value=>'x'} , { name => 'special',visible=> 1 } ]

};

**Full redefinition:** is not implemented yet.

## Item Level

### Buttons (BUTTON_DEF)

Defines a button.:

```
{
    page => STRING_PAGE,
    action => STRING_ACTION, # parameters "page" and "action" will be transmitted to server. if ar
    label => STRING, # The label of the button
    target => STRING_TARGET, # one of main|modal|right|tab (optional, default is main)
    css_class => STRING, # optional, css class for the button element
    do_submit => BOOL, # optional, if true, the button submits the contents of the form to the giv
}
```

### Formattet Strings (STRING_FORMAT)

Tells the ui to process the data before rendering with a special formatter. Available methods are:

#### timestamp

Expects a unix timestamp and outputs a full UTC timestamp.

#### datetime

Expects a parseable date, outputs a full UTC timestamp.

#### certstatus

Colorizes the given status word using css tags, e.g. `issued` becomes:

```
<span class="certstatus-issued">issued</span>
```

#### link

Create an internal framework link to a page or action, expects a hash with `label` and `page` and optional `target`, default is to open the link in a modal.

#### extlink

Similar to link but expects href to be an external target, default target is blank.

#### text

Readable text without html syntax (will be escaped)

#### raw

Displayed as is.

#### code

Rendered with fixed-with typo, unix linebreaks are converted to html linebreaks.

#### defhash/deflist

Outputs a key/value list (dl/dt/dd) - defhash expects a hash where keys are labels. deflist expects an array where each item is a hash with keys key and value.

#### ullist

Array of values, each item becomes a <li> in the list, values are html-escaped.

#### rawlist

Like ullist but displays the items "as is" (can contain HTML markup)

#### linklist

Array, where each item is a hash describing a link

## Customization

The framework allows to register additional components via an exposed api.

### Form-Field

Add a new FormField-Type:

```
OXI.FormFieldFactory.registerComponent('type','ComponentName',JS_CODE [,bOverwriteExisting]);
```

*Example*:

```
OXI.FormFieldFactory.registerComponent('select','MySpecialSelect', OXI.FormFieldContainer.extend(
    ....
}), true);
```

This will overwrite the handler for the select element. The ComponentName will be registered in the OXI Namespace and can be used to call the object from within userdefined code.

#### Formatter

Add a new Format-Handler:

```
OXI.FormatHelperFactory.registerComponent('format','ComponentName',JS_CODE [,bOverwriteExisting])
```

Operation

# Technical Operation Manual

# Crypto Token Configuration

## Overview

A cypto token is an entity used to do cryptographic operations. OpenXPKI organizes those tokens using groups and generations. A default system has four groups:

- certsign - represents the Issuing CA
- datasafe - used internally to encrypt sensitive data
- scep - the operational certificate of the SCEP server
- root - the root certificate of the Issuing CA chain

OpenXPKI expects that a token has only a limited lifetime and is substituted by a successor at a certain point in time. This relation is expressed by the generation counter.

## Initial Setup

All tokens consist of a private key and a certificate, the certificate must be present in the OpenXPKI internal database and is referenced by the certificate identifier. The private key lives outside the OpenXPKI systems. When using the default config, the system expects the private key as file where the name of the file is constructed from the complete alias name.

#### Root Certificate

openxpkiadm certificate import –file ca-root-1.crt

**openxpkiadm certificate import –file ca-one-signer-1.crt** –realm ca-one –token certsign

Architecture

# SCEP Server

The scep functionality is included as a special service with the core distribution. The scep service needs to be enabled in the global system configuration (`system.server.service`).

The communication with your scep clients requires the deployment of a cgi wrapper script with your webserver. The script will just parse the HTTP related parts and pass the data to the openxpki daemon and vice versa.

## Wrapper Configuration

The default wrapper looks for its config file at `/etc/openxpki/scep/default.conf`. The config uses plain ini format, a default is deployed by the package:

```
[global]
log_config = /etc/openxpki/scep/log.conf
log_facility = client.scep

socket=/var/openxpki/openxpki.socket
realm=ca-one
iprange=0.0.0.0/0
profile=I18N_OPENXPKI_PROFILE_TLS_SERVER
servername=scep-server-1
encryption_algorithm=3DES
```

### Parameters

**log_config**

Path to the log4perl config file.

**log_facility**

Facility to log with.

**socket**

Location of the OpenXPKI socket file, the webserver needs rw access.

**realm**

The realm of the ca to be used.

**iprange**

Implements a simple ip based access control, the clients ip adress is checked to be included in the given network. Only a single network definition is supported, the default of 0.0.0.0/0 allows all ips to connect.

**profile**

The default profile of the certificate to be requested, note that depending on the backing workflow this might be ignored or overridden by other paramters.

**servername**

Path to the server side config of this scep service. Equal to the key from the config section in the scep.yaml file.

**encryption**

Encrpytion to use, supported values are I<DES> and I<3DES>.

### Multiple Configs

The default location for config files is /etc/openxpki/scep, in combination with the default wrapper setup, the part after /scep/ in the url is used to probe for a filename holding a custom config, e.g.:

```
http://host/scep/mailgw  -> /etc/openxpki/scep/mailgw.conf
```

If no file is found, the default config is loaded from /etc/openxpki/scep/default.conf. The wrapper uses SCRIPT_URL or REQUEST_URI from the apache environment to extract the requests path, this should work in most environments with mod_rewrite, symlinks or alias definitions.

**custom base directory**

Set *OPENXPKI_SCEP_CLIENT_CONF_DIR* to a directory path. The autodetection will now use this path to find either the special or the default file. Note that there is no fallback to the default location!

**fixed file**

Set *OPENXPKI_SCEP_CLIENT_CONF_FILE* to an absolute file path. On apache, this can be combined with location to set a config for a special script:

```
<Location /cgi-bin/scep/mailgateway>
   SetEnv OPENXPKI_SCEP_CLIENT_CONF_FILE /home/mailadm/scep.conf
</Location>
```

*Note*: The scep standard is not exact about the use of HTTP/1.1 features. We saw a lot of clients which where sending plain HTTP/1.0 requests which is not compatible with name based virtual hosting!

# SOAP Server

The builtin SOAP Server provides methods to revoke certificates. The service is implemented using a cgi-wrapper script, so there is no need for the webserver to support SOAP, you just need to setup the wrapper script. For apache, just add a ScriptAlias:

```
ScriptAlias /soap  /usr/lib/cgi-bin/soap.fcgi
```

## Wrapper Configuration

The default wrapper looks for its config file at `/etc/openxpki/scep/default.conf`. The config uses plain ini format, a default is deployed by the package:

```
[global]
log_config = /etc/openxpki/soap/log.conf
log_facility = client.soap
socket = /var/openxpki/openxpki.socket
modules = OpenXPKI::SOAP::Revoke OpenXPKI::SOAP::Smartcard

[auth]
stack = _System
pki_realm = ca-one

[OpenXPKI::SOAP::Revoke]
workflow = certificate_revocation_request_v2
servername = signed-revoke

[OpenXPKI::SOAP::Smartcard]
workflow = sc_revoke
servername = smartcard-revoke
```

## Endpoint Configuration

Based on the given servername, a rules file is loaded for the server. You can define the rules for the signer authorization here:

```
authorized_signer:
    rule1:
        subject: CN=.+:soapclient,.*
    rule2:
        subject: CN=.+:pkiclient,.*
```

## SOAP Methods

The default interface exposes two methods. The reason code is optional in both calls and defaults to “unspecified”. Allowed values are the reason codes as used by openssl.

#### RevokeCertificateByIssuerSerial

This expects the full DN of the certificate issuer and the serial number of the certificate to revoke. The serial can be either in decimal or hexadecimal format prefixed with '0x':

```
RevokeCertificateByIssuerSerial(
    'CN=CA ONE,OU=Test CA,DC=OpenXPKI,DC=ORG',
    '0xdb7d5b06600bddcbecff',
    'keyCompromise'
)
```

#### RevokeCertificateByIdentifier

Expects the OpenXPKI identifier of the certificate:

```
RevokeCertificateByIdentifier(
    'TZNrDctI9RV8DT5TGvg81w7F-So',
    'keyCompromise'
)
```

Both calls return a hash with id and state of the started workflow:

```
{
  'id' => '145919',
  'state' => 'PENDING',
  'error' => ''
}
```

If anything goes wrong, you get a verbose error message in error:

```
{
  'error' => 'parameter missing'
}
```

### Multiple Endpoints

t.b.d

## Enrollment UI

This is a certificate enrollment interface for OpenXPKI. Basically, it runs on a bastion host and accepts CSRs from external users. These CSRs are passed to an internal OpenXPKI daemon via SCEP using the sscep to forward the request.

### Architecture

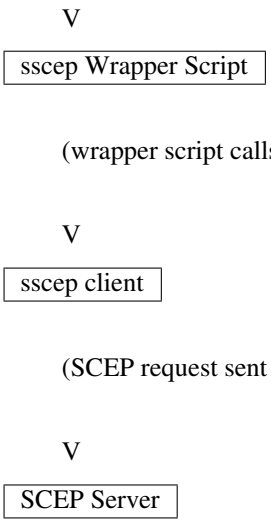
Apache HTTP Server

(CGI call of 'enroller' script)

V

Enroller Web UI

(enroller calls wrapper script)




## Configuration

The Mojolicious framework is designed to run nicely in a PSGI or CGI environment of a webserver. To run a test daemon that is reachable via your web browser, run the following:

script/enroller daemon

To run the test cases, use the following:

script/enroller test

### Apache HTTP Server

One method of serving the Enrollment UI is via Apache/CGI using the ScriptAlias directive:

**<Directory /srv/www/enroller>**

Options -FollowSymLinks AllowOverride None Order allow,deny Allow from all

</Directory> ScriptAlias / /srv/www/enroller/script/enroller/

### SCEP Client (e.g. sscep)

**TODO**

Developer

# Extending OpenXPKI

# Glossary

**CA Rollover** Using multiple CA Certificates with overlapping validity to issue certificates for the same purpose and namespace. CA Rollover allows for continuous and uninterrupted operation of a PKI.

**PKI Realm** A PKI realm is a "Logical Certificate Authority" which usually includes one or more [Issuing_CA] that are responsible for issuing certificates within the same name space.

CHAPTER 2

# Indices and tables

- genindex
- search

## C



## P