MapProxy Documentation

Release 1.12.0a

Oliver Tonnhofer

2018-03-20

CHAPTER

ONE

INSTALLATION

This tutorial guides you to the MapProxy installation process on Unix systems. For Windows refer to *Installation on Windows*.

This tutorial was created and tested with Debian 5.0/6.0 and Ubuntu 10.04 LTS, if you're installing MapProxy on a different system you might need to change some package names.

MapProxy is registered at the Python Package Index (PyPI). If you have installed Python setuptools (`python-setuptools` on Debian) you can install MapProxy with `sudo easy_install MapProxy`.

This is really easy *but* we recommend to install MapProxy into a virtual Python environment. A `virtualenv` is a self-contained Python installation where you can install arbitrary Python packages without affecting the system installation. You also don't need root permissions for the installation.

Read about virtualenv if you want to know more about the benefits.

1.1 Create a new virtual environment

`virtualenv` is available as `python-virtualenv` on most Linux systems. You can also download a self-contained version:

```
wget https://github.com/pypa/virtualenv/raw/master/virtualenv.py
```

To create a new environment with the name `mapproxy` call:

```
virtualenv --system-site-packages mapproxy
# or
python virtualenv.py --system-site-packages mapproxy
```

You should now have a Python installation under `mapproxy/bin/python`.

Note: Newer versions of virtualenv will use your Python system packages (like `python-imaging` or `python-yaml`) only when the virtualenv was created with the `--system-site-packages` option. If your (older) version of virtualenv does not have this option, then it will behave that way by default.

You need to either prefix all commands with `mapproxy/bin`, set your `PATH` variable to include the bin directory or *activate* the virtualenv with:

```
source mapproxy/bin/activate
```

This will change the `PATH` for you and will last for that terminal session.

1.2 Install Dependencies

MapProxy is written in Python, thus you will need a working Python installation. MapProxy works with Python 2.7, 3.3 and 3.4 which should already be installed with most Linux distributions. Python 2.6 should still work, but it is no longer officially supported.

MapProxy has some dependencies, other libraries that are required to run. There are different ways to install each dependency. Read *Dependency details* for a list of all required and optional dependencies.

1.2.1 Installation

On a Debian or Ubuntu system, you need to install the following packages:

```
sudo aptitude install python-imaging python-yaml libproj0
```

To get all optional packages:

```
sudo aptitude install libgeos-dev python-lxml libgdal-dev python-shapely
```

Note: Check that the `python-shapely` package is `>=1.2`, if it is not you need to install it with `pip install Shapely`.

1.2.2 Dependency details

libproj

MapProxy uses the Proj4 C Library for all coordinate transformation tasks. It is included in most distributions as `libproj0`.

Pillow

Pillow, the successor of the Python Image Library (PIL), is used for the image processing and it is included in most distributions as `python-imaging`. Please make sure that you have Pillow installed as MapProxy is no longer compatible with the original PIL. The version of `python-imaging` should be >=2.

You can install a new version of Pillow from source with:

```
sudo aptitude install build-essential python-dev libjpeg-dev \
  zlib1g-dev libfreetype6-dev
pip install Pillow
```

YAML

MapProxy uses YAML for the configuration parsing. It is available as `python-yaml`, but you can also install it as a Python package with `pip install PyYAML`.

Shapely and GEOS *(optional)*

You will need Shapely to use the *coverage feature* of MapProxy. Shapely offers Python bindings for the GEOS library. You need Shapely (`python-shapely`) and GEOS (`libgeos-dev`). You can install Shapely as a Python package with `pip install Shapely` if you system does not provide a recent (>= 1.2.0) version of Shapely.

GDAL *(optional)*

The *coverage feature* allows you to read geometries from OGR datasources (Shapefiles, PostGIS, etc.). This package is optional and only required for OGR datasource support (BBOX, WKT and GeoJSON coverages are supported natively). OGR is part of GDAL (`libgdal-dev`).

lxml *(optional)*

lxml is used for more advanced WMS FeatureInformation operations like XSL transformation or the concatenation of multiple XML/HTML documents. It is available as `python-lxml`.

1.3 Install MapProxy

Your virtual environment should already contain pip, a tool to install Python packages. If not, `easy_install pip` is enough to get it.

To install you need to call:

```
pip install MapProxy
```

You specify the release version of MapProxy. E.g.:

```
pip install MapProxy==1.8.0
```

or to get the latest 1.8.0 version:

```
pip install "MapProxy>=1.8.0,<=1.8.99"
```

To check if the MapProxy was successfully installed, you can call the *mapproxy-util* command.

```
mapproxy-util --version
```

Note: `pip` and `easy_install` will download packages from the Python Package Index and therefore they require full internet access. You need to set the `http_proxy` environment variable if you only have access to the internet via an HTTP proxy. See *Access sources through HTTP proxy* for more information.

1.4 Create a configuration

To create a new set of configuration files for MapProxy call:

```
mapproxy-util create -t base-config mymapproxy
```

This will create a `mymapproxy` directory with a minimal example configuration (`mapproxy.yaml` and `seed.yaml`) and two full example configuration files (`full_example.yaml` and `full_seed_example.yaml`).

Refer to the *configuration documentation* for more information. With the default configuration the cached data will be placed in the `cache_data` subdirectory.

1.5 Start the test server

To start a test server:

```
cd mymapproxy
mapproxy-util serve-develop mapproxy.yaml
```

There is already a test layer configured that obtains data from the Omniscale OpenStreetMap WMS. Feel free to use this service for testing.

MapProxy comes with a demo service that lists all configured WMS and TMS layers. You can access that service at http://localhost:8080/demo/

1.6 Upgrade

You can upgrade MapProxy with pip in combination with a version number or with the `--upgrade` option. Use the `--no-deps` option to avoid upgrading the dependencies.

To upgrade to version 1.x.y:

```
pip install 'MapProxy==1.x.y'
```

To upgrade to the latest release:

```
pip install --upgrade --no-deps MapProxy
```

To upgrade to the current development version:

```
pip install --upgrade --no-deps https://github.com/mapproxy/mapproxy/tarball/master
```

1.6.1 Changes

New releases of MapProxy are backwards compatible with older configuration files. MapProxy will issue warnings on startup if a behavior will change in the next releases. You are advised to upgrade in single release steps (e.g. 1.2.0 to 1.3.0 to 1.4.0) and to check the output of `mapproxy-util serve-develop` for any warnings. You should also refer to the Changes Log of each release to see if there is anything to pay attention for.

If you upgrade from 0.8, please read the old mirgation documentation <http://mapproxy.org/docs/1.5.0/migrate.html>‘_.

CHAPTER

TWO

INSTALLATION ON WINDOWS

At frist you need a working Python installation. You can download Python from: https://www.python.org/download/. MapProxy requires Python 2.7, 3.3, 3.4, 3.5 or 3.6. Python 2.6 should still work, but it is no longer officially supported. We would recommend the latest 2.7 version available.

2.1 Virtualenv

If you are using your Python installation for other applications as well, then we advise you to install MapProxy into a virtual Python environment to avoid any conflicts with different dependencies. *You can skip this if you only use the Python installation for MapProxy.* Read about virtualenv if you want to now more about the benefits.

To create a new virtual environment for your MapProxy installation and to activate it go to the command line and call:

```
C:\Python27\python path\to\virtualenv.py c:\mapproxy_venv
C:\mapproxy_venv\Scripts\activate.bat
```

Note: The last step is required every time you start working with your MapProxy installation. Alternatively you can always explicitly call `\mapproxy_venv\Scripts\<command>`.

Note: Apache mod_wsgi does not work well with virtualenv on Windows. If you want to use mod_wsgi for deployment, then you should skip the creation the virtualenv.

After you activated the new environment, you have access to `python` and `pip`. To install MapProxy with most dependencies call:

```
pip install MapProxy
```

This might take a minute. You can skip the next step.

2.2 PIP

MapProxy and most dependencies can be installed with the `pip` command. `pip` is already installed if you are using Python >=2.7.9, or Python >=3.4. Read the pip documentation for more information.

After that you can install MapProxy with:

```
c:\Python27\Scripts\pip install MapProxy
```

This might take a minute.

2.3 Dependencies

Read *Dependency details* for more information about all dependencies.

2.3.1 Pillow and YAML

Pillow and PyYAML are installed automatically by `pip`.

2.3.2 PyProj

Since libproj4 is generally not available on a Windows system, you will also need to install the Python package `pyproj`. You need to manually download the `pyproj` package for your system. See below for *Platform dependent packages*.

```
pip install path\to\pyproj-xxx.whl
```

2.3.3 Shapely and GEOS *(optional)*

Shapely can be installed with `pip install Shapely`. This will already include the required `geos.dll`.

2.3.4 GDAL *(optional)*

MapProxy requires GDAL/OGR for coverage support. MapProxy can either load the `gdal.dll` directly or use the `osgeo.ogr` Python package. You can download and install inofficial Windows binaries of GDAL and the Python package (e.g. *gdal-19-xxxx-code.msi*).

You need to add the installation path to the Windows `PATH` environment variable in both cases. You can set the variable temporary on the command line (spaces in the filename need no quotes or escaping):

```
set PATH=%PATH%;C:\Program Files (x86)\GDAL
```

Or you can add it to your systems environment variables.

You also need to set `GDAL_DRIVER_PATH` or `OGR_DRIVER_PATH` to the `gdalplugins` directory when you want to use the Oracle plugin (extra download from URL above):

```
set GDAL_DRIVER_PATH=C:\Program Files (x86)\GDAL\gdalplugins
```

2.4 Platform dependent packages

`pip` downloads all packages from https://pypi.python.org/, but not all platform combinations might be available as a binary package, especially if you run a 64bit version of Python.

If you run into trouble during installation, because it is trying to compile something (e.g. complaining about `vcvarsall.bat`), you should look at Christoph Gohlke's Unofficial Windows Binaries for Python Extension Packages. This is a reliable site for binary packages for Python. You need to download the right package: The `cpxx` code refers to the Python version (e.g. `cp27` for Python 2.7); `win32` for 32bit Python installations and `amd64` for 64bit.

You can install the `.whl`, `.zip` or `.exe` packages with `pip`:

```
pip install path\to\package-xxx.whl
```

2.5 Check installation

To check if the MapProxy was successfully installed you can call `mapproxy-util`. You should see the installed version number.

```
mapproxy-util --version
```

Now continue with *Create a configuration* from the installation documentation.

CHAPTER

THREE

INSTALLATION ON OSGEO4W

OSGeo4W is a popular package of open-source geospatial tools for Windows systems. Besides packing a lot of GIS tools and a nice installer, it also features a full Python installation, along with some of the packages that MapProxy needs to run.

In order to install MapProxy within an OSGeo4W environment, the first step is to ensure that the needed Python packages are installed. In order to do so:

- Download and run the *OSGeo4W installer*
- Select advanced installation
- When shown a list of available packages, check (at least) `python` and `python-pil` for installation.

Please refer to the OSGeo4W installer FAQ if you've got trouble running it.

At this point, you should see an OSGeo4W shell icon on your desktop and/or start menu. Right-click that, and *run as administrator*.

In the OSGeo4W window, run:

```
C:\OSGeo4W> pip install mapproxy
```

and

```
C:\OSGeo4W> pip install pyproj
```

If these last two commands didn't print out any errors, your installation of MapProxy is successful. You can now close the OSGeo4W shell with administrator privileges, as it is no longer needed.

In older versions of OSGeo4W `pip` may not recognized. In such a case, please follow the instructions for installing pip with get-pip.py and rerty the above `pip install` commands.

3.1 Check installation

To check if the MapProxy was successfully installed, you can launch a regular OSGeo4W shell, and call `mapproxy-util`. You should see the installed version number:

```
C:\OSGeo4W> mapproxy-util --version
```

Note: You need to run *all* MapProxy-related commands from an OSGeo4W shell, and not from a standard command shell.

Now continue with *Create a configuration* from the installation documentation.

3.2 Unattended OSGeo4W environment

If you need to run unattended commands (like scheduled runs of *mapproxy-seed*), make a copy of `C:\OSGeo4W\OSGeo4W.bat` and modify the last line, to call `cmd` so it runs the MapProxy script you need, e.g.:

```
cmd /c mapproxy-seed -s C:\path\to\seed.yaml -f C:\path\to\mapproxy.yaml
```

CHAPTER

FOUR

TUTORIAL

This tutorial should give you a quick introduction to the MapProxy configuration.

You should have a *working MapProxy installation*, if you want to follow this tutorial.

4.1 Configuration format

The configuration of MapProxy uses the YAML format. YAML is a superset of JSON. That means every valid JSON is also valid YAML. MapProxy uses no advanced features of YAML, so you could even use JSON. YAML uses a more readable and user-friendly syntax. We encourage you to use it.

If you are familiar with YAML you can skip to the next section.

The YAML configuration consist of comments, dictionaries, lists, strings, numbers and booleans.

4.1.1 Comments

Everything after a hash character (`#`) is a comment and will be ignored.

4.1.2 Numbers

Any numerical value like `12`, `-4`, `0`, and `3.1415`.

4.1.3 Strings

Any string within single or double quotes. You can omit the quotes if the string has no other meaning in YAML syntax. For example:

```
'foo'
foo
'43' # with quotes, otherwise it would be numeric
'[string, not a list]'
A string with spaces and punctuation.
```

4.1.4 Booleans

True or false values:

```
yes
true
True
no
false
False
```

4.1.5 List

A list is a collection of other valid objects. There are two formats. The condensed form uses square brackets:

```
[1, 2, 3]
[42, string, [another list with a string]]
```

The block form requires every list item on a separate line, starting with - (dash and a blank):

```
- 1
- 2
- 3

- 42
- string
- [another list]
```

4.1.6 Dictionaries

A dictionary maps keys to values. Values itself can be any valid object.

There are two formats. The condensed form uses braces:

```
{foo: 3, bar: baz}
```

The block form requires every key value pair on a seperate line:

```
foo: 3
bar: baz
```

You can also nest dictionaries. Each nested dictionary needs to be indented by one or more whitespaces. Tabs are *not* permitted and all keys to the same dictionary need to be indented by the same amount of spaces.

```
baz:
  ham: 2
  spam:
    bam: True
  inside_baz: 'yepp'
```

4.2 Configuration Layout

The MapProxy configuration is a dictionary, each key configures a different aspect of MapProxy. There are the following keys:

- **`services`: This is the place to activate and configure MapProxy's services** like WMS and TMS.
- **`layers`: Configure the layers that MapProxy offers. Each layer can consist** of multiple sources and caches.
- **`sources`:** Define where MapProxy can retrieve new data.
- **`caches`:** Here you can configure the internal caches.
- **`grids`: MapProxy aligns all cached images (tiles) to a grid. Here you can define** that grid.
- **`globals`: Here you can define some internals of MapProxy and default values** that are used in the other configuration directives.

The order of the directives is not important, so you can organize it your way.

4.3 Example Configuration

4.3.1 Configuring a Service

At first we need to *configure at least one service*. To enable a service, you have to include its name as a key in the *services* dictionary. For example:

```
services:
  tms:
```

Each service is a YAML dictionary, with the service type as the key. The dictionary can be empty, but you need to add the colon so that the configuration parser knows it's a dictionary.

A service might accept more configuration options. The WMS service, for example, takes a dictionary with metadata. This data is used in the capabilities documents.

Here is an example with some contact information:

```
services:
  wms:
    md:
        title: MapProxy WMS Proxy
        abstract: This is the fantastic MapProxy.
        online_resource: http://mapproxy.org/
        contact:
            person: Your Name Here
            position: Technical Director
            organization:
            address: Fakestreet 123
            city: Somewhere
            postcode: 12345
            country: Germany
            phone: +49(0)000-000000-0
            fax: +49(0)000-000000-0
            email: info@omniscale.de
        access_constraints:
            This service is intended for private and
            evaluation use only. The data is licensed
            as Creative Commons Attribution-Share Alike 2.0
            (http://creativecommons.org/licenses/by-sa/2.0/)
        fees: 'None'
```

access_constraints demonstrates how you can write a string over multiple lines, just indent every line the same way as the first. And remember, YAML does not accept tab characters, you must use space.

For this tutorial we add another service called *demo*. This is a demo service that lists all configured WMS and TMS layers. You can test each layer with a simple OpenLayers client. So our configuration file should look like:

```
services:
  demo:
  wms:
    [rest of WMS configuration]
```

4.3.2 Adding a Source

Next you need to *define the source* of your data. Every source has a name and a type. Let's add a WMS source:

```
sources:
  test_wms:
    type: wms
    req:
```

```
    url: http://osm.omniscale.net/proxy/service?
    layers: osm
```

In this example *test_wms* is the name of the source, you need this name later to reference it. Most sources take more parameters – some are optional, some are required. The type *wms* requires the *req* parameter that describes the WMS request. You need to define at least a URL and the layer names, but you can add more options like *transparent* or *format*.

4.3.3 Adding a Layer

After defining a source we can use it to *create a layer* for the MapProxy WMS.

A layer requires a title, which will be used in the capabilities documents and a source. For this layer we want to use our *test_wms* data source:

```
layers:
  - name: cascaded_test
    title: Cascaded Test Layer
    sources: [test_wms]
```

Now we have setuped MapProxy as cascading WMS. That means MapProxy only redirect requests to the WMS defined in *test_wms* data source.

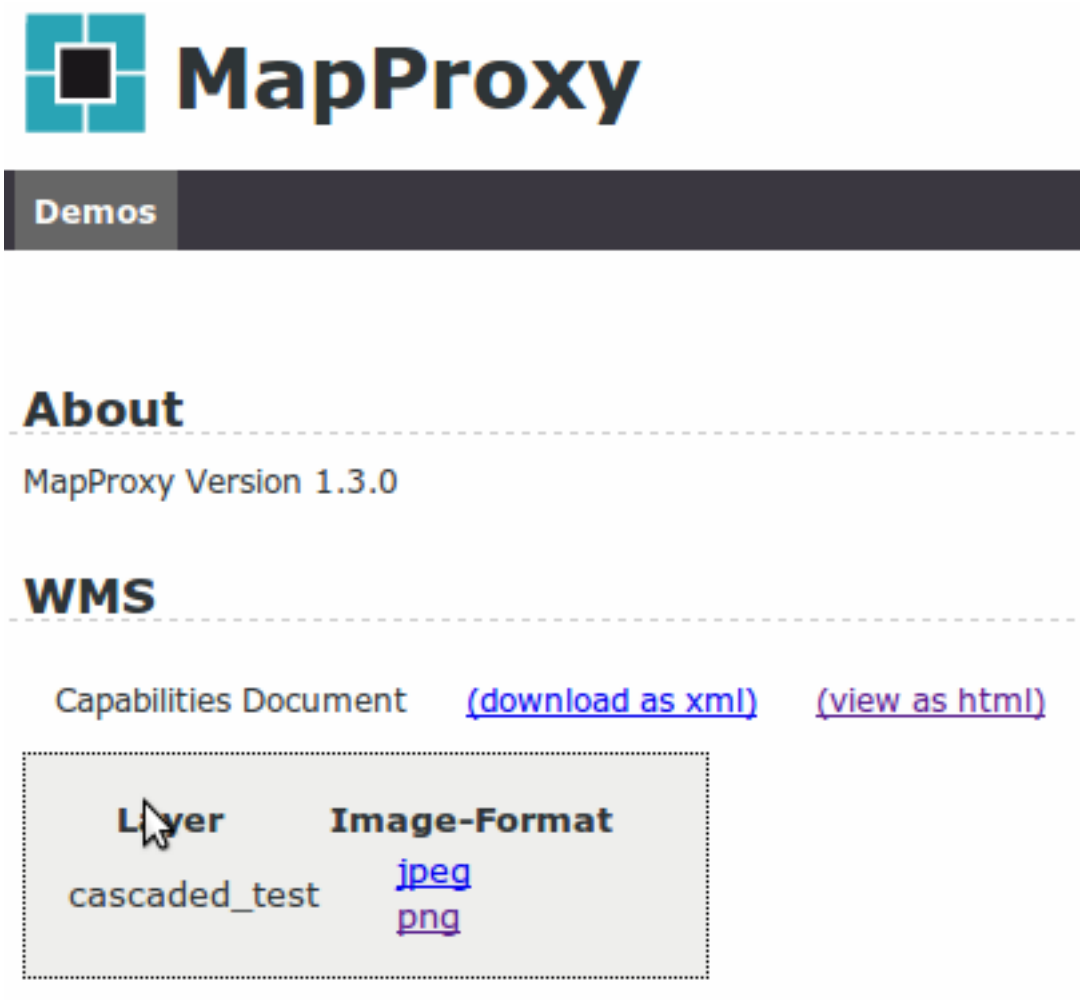
4.3.4 Starting the development server

That's it for the first configuration, you can now *start MapProxy*:

```
mapproxy-util serve-develop mapproxy.yaml
```

You can `download the configuration`.

When you type *localhost:8080/demo/* in the URL of your webbrowser you should see a demo site like shown below.



Here you can see the capabilities of your configured service and watch it in action.

4.3.5 Adding a Cache

To speed up the source with MapProxy we *create a cache* for this source.

Each cache needs to know where it can get new data and how it should be cached. We define our *test_wms* as source for the cache. MapProxy splits images in small tiles and these tiles will be aligned to a grid. It also caches images in different resolutions, like an image pyramid. You can define this image pyramid in detail but we start with one of the default grid definitions of MapProxy. *GLOBAL_GEODETIC* defines a grid that covers the whole world. It uses EPSG:4326 as the spatial reference system and aligns with the default grid and resolutions that OpenLayers uses.

Our cache configuration should now look like:

```
caches:
  test_wms_cache:
    sources: [test_wms]
    grids: [GLOBAL_GEODETIC]
```

4.3.6 Adding a cached Layer

We can now use our defined cache as source for a layer. When the layer is requested by a client, MapProxy looks in the cache for the requested data and only if it hasn't cached the data yet, it requests the *test_wms* data source.

The layer configuration should now look like:

```
layers:
  - name: test_wms_cache
    title: Cached Test Layer
    sources: [test_wms_cache]
```

You can download the configuration.

4.3.7 Defining Resolutions

By default MapProxy caches traditional power-of-two image pyramids with a default number of cached resolutions of 20. The resolutions between each pyramid level doubles. If you want to change this, you can do so by *defining your own grid*. Fortunately MapProxy grids provied the ability to inherit from an other grid. We let our grid inherit from the previously used *GLOBAL_GEODETIC* grid and add five fixed resolutions to it.

The grid configuration should look like:

```
grids:
  res_grid:
    base: GLOBAL_GEODETIC
    res: [1, 0.5, 0.25, 0.125, 0.0625]
```

As you see, we used *base* to inherit from *GLOBAL_GEODETIC* and *res* to define our preferred resolutions. The resolutions are always in the unit of the SRS, in this case in degree per pixel. You can use the *MapProxy scales util* to convert between scales and resolutions.

Instead of defining fixed resolutions, we can also define a factor that is used to calculate the resolutions. The default value of this factor is 2, but you can set it to each value you want. Just change *res* with *res_factor* and add your preferred factor after it.

A magical value of *res_factor* is **sqrt2**, the square root of two. It doubles the number of cached resolutions, so you have 40 instead of 20 available resolutions. Every second resolution is identical to the power-of-two resolutions, so you can use this layer not only in classic WMS clients with free zomming, but also in tile-based clients like OpenLayers which only request in these resolutions. Look at the *configuration examples for vector data for more information*.

4.3.8 Defining a Grid

In the previous section we saw how to extend a grid to provide self defined resolutions, but sometimes *GLOBAL_GEODETIC* grid is not useful because it covers the whole world and we want only a part of it. So let's see how to *define our own grid*.

For this example we define a grid for Germany. We need a spatial reference system (*srs*) that match the region of Germany and a bounding box (*bbox*) around Germany to limit the requestable aera. To make the specification of the *bbox* a little bit easier, we put the *bbox_srs* parameter to the grid configuration. So we can define the *bbox* in EPSG:4326.

The *grids* configuration is a dictionary and each grid configuration is identified by its name. We call our grid *germany* and its configuration should look like:

```
grids:
  germany:
    srs: 'EPSG:25832'
    bbox: [6, 47.3, 15.1, 55]
    bbox_srs: 'EPSG:4326'
```

We have to replace *GLOBAL_GEODETIC* in the cache configuration with our *germany* grid. After that MapProxy caches all data in UTM32.

MapProxy request the source in the projection of the grid. You can configure *the supported SRS for each WMS source* and MapProxy takes care of any transformations if the *srs* of our grid is different from the data source.

You can download the configuration.

4.3.9 Merging Multiple Layers

If you have two WMS and want to offer a single layer with data from both server, you can combine these in one cache. MapProxy will combine the images before it stores the tiles on disk. The sources should be defined from bottom to top and all sources except the bottom need to be transparent.

The code below is an example for configure MapProxy to combine two WMS in one cache and one layer:

```
services:
  wms:
  demo:

sources:
  test_wms:
    type: wms
    req:
      url: http://osm.omniscale.net/proxy/service?
      layers: osm
  roads_wms:
    type: wms
    req:
      url: http://osm.omniscale.net/proxy/service?
      layers: osm_roads
      transparent: true

caches:
  combined_cache:
    sources: [test_wms, roads_wms]
    grids: [GLOBAL_GEODETIC]

layers:
  - name: cached_test_wms_with_roads
    title: Cached Test WMS with Roads
    sources: [combined_cache]
```

You can `download the configuration`.

4.3.10 Coverages

Sometimes you don't want to provide the full data of a WMS in a layer. With MapProxy you can define areas where data is available or where data you are interested in is. MapProxy provides three ways to restrict the area of available data: Bounding boxes, polygons and OGR datasource. To keep it simple, we only discuss bounding boxes. For more informations about the other methods take a look at *the coverages documentation*. To restrict the area with a bounding box, we have to define it in the coverage option of the data source. The listing below restricts the requestable area to Germany:

```
sources:
  test_wms:
    type: wms
    req:
      url: http://osm.omniscale.net/proxy/service?
      layers: osm
    coverage:
      bbox: [5.5, 47.4, 15.2, 54.8]
      bbox_srs: 'EPSG:4326'
```

As you see notation of a coverage bounding box is similar to the notation in the grid option.

4.3.11 Meta Tiles and Meta Buffer

When you have experience with WMS in tiled clients you should know the problem of labeling issues. MapProxy can help to resolve these issues with two methods called *Meta Tiling* and *Meta Buffering*.

There is a *chapter on WMS labeling issues* that discusses these options.

4.3.12 Seeding

Configuration

MapProxy creates all tiles on demand. That means, only tiles requested once are cached. Fortunately MapProxy comes with a command line script for pre-generating all required tiles called `mapproxy-seed`. It has its own configuration file called `seed.yaml` and a couple of options. We now create a config file for `mapproxy-seed`.

As all MapProxy configuration files it's notated in YAML. The mandatory option is `seeds`. Here you can create multiple seeding tasks that define what should be seeded. You can specify a list of caches for seeding with `caches` . The cache names should match the names in your MapProxy configuration. If you have specified multiple grids for one cache in your MapProxy configuration, you can select these caches to seed. They must also comply with the caches in your MapProxy configuration. Furthermore you can limit the levels that should be seeded. If you want to seed only a limited area, you can use the `coverages` option.

In the example below, we configure `mapproxy-seed` to seed our previously created cache `test_wms_cache` from level 6 to level 16. To show a different possibility to define a coverage, we use a polygon file to determine the area we want to seed.

```
seeds:
  test_cache_seed:
    caches: [test_wms_cache]
    levels:
      from: 6
      to: 16
    coverages: [germany]

coverages:
  germany:
    polygons: ./GM.txt
    polygons_srs: EPSG:900913
```

As you see in the `coverages` section the `polygons` option point to a text file. This text file contains polygons in Well-Known-Text (WKT) form. The third option tells `mapproxy-seed` the `srs` of the WKT polygons.

You can `download the configuration` and the `polygon file`.

Start Seeding

Now it's time to start seeding. `mapproxy-seed` has a couple of options. We have to use options `-s` to define our `seed.yaml` and `-f` for our MapProxy configuration file. We also use the `--dry-run` option to see what MapProxy would do, without making any actual requests to our sources. A mis-configured seeding can take days or weeks, so you should keep an eye on the tile numbers the dry-run prints out.

Run `mapproxy-seed` like:

```
mapproxy-seed -f mapproxy.yaml -s seed.yaml --dry-run
```

If you sure, that seeding works right, remove `--dry-run`.

4.3.13 What's next?

You should read the *configuration examples* to get a few more ideas what MapProxy can do.

MapProxy has lots of small features that might be useful for your projects, so it is a good idea to read the other chapters of the documentation after that.

If you have any questions? We have a mailing list and IRC channel where you can get support.

CHAPTER

FIVE

CONFIGURATION

There are two configuration files used by MapProxy.

`mappproxy.yaml` This is the main configuration of MapProxy. It configures all aspects of the server: Which servers should be started, where comes the data from, what should be cached, etc..

`seed.yaml` This file is the configuration for the `mapproxy-seed` tool. See *seeding documentation* for more information.

5.1 mapproxy.yaml

The configuration uses the YAML format. The Wikipedia contains a good introduction to YAML.

The MapProxy configuration is grouped into sections, each configures a different aspect of MapProxy. These are the following sections:

- `globals`: Internals of MapProxy and default values that are used in the other configuration sections.
- `services`: The services MapProxy offers, e.g. WMS or TMS.
- `sources`: Define where MapProxy can retrieve new data.
- `caches`: Configure the internal caches.
- `layers`: Configure the layers that MapProxy offers. Each layer can consist of multiple sources and caches.
- `grids`: Define the grids that MapProxy uses to aligns cached images.

The order of the sections is not important, so you can organize it your way.

Note: The indentation is significant and shall only contain space characters. Tabulators are **not** permitted for indentation.

There is another optional section:

New in version 1.6.0.

- `parts`: YAML supports references and with that you can define configuration parts and use them in other configuration sections. For example, you can define all you coverages in one place and reference them from the sources. However, MapProxy will log a warning if you put the referent in a place where it is not a valid option. To prevent these warnings you are advised to put these configuration snippets inside the `parts` section.

For example:

```
parts:
  coverages:
      mycoverage: &mycoverage
        bbox: [0, 0, 10, 10]
        srs: 'EPSG:4326'

sources:
```

```
  mysource1:
    coverage: *mycoverage
    ...
  mysource2:
    coverage: *mycoverage
    ...
```

5.1.1 base

You can split a configuration into multiple files with the `base` option. The `base` option loads the other files and merges the loaded configuration dictionaries together – it is not a literal include of the other files.

For example:

```
base: [mygrids.yaml, mycaches_sources.yaml]
service: ...
layers: ...
```

Changed in version 1.4.0: Support for recursive imports and for multiple files.

5.2 services

Here you can configure which services should be started. The configuration for all services is described in the *Services* documentation.

Here is an example:

```
services:
  tms:
  wms:
    md:
      title: MapProxy Example WMS
      contact:
      # [...]
```

5.3 layers

Here you can define all layers MapProxy should offer. The layer definition is similar to WMS: each layer can have a name and title and you can nest layers to build a layer tree.

Layers should be configured as a list (– in YAML), where each layer configuration is a dictionary (`key: value` in YAML)

```
layers:
  - name: layer1
    title: Title of Layer 1
    sources: [cache1, source2]
  - name: layer2
    title: Title of Layer 2
    sources: [cache3]
```

Each layer contains information about the layer and where the data comes from.

Changed in version 1.4.0.

The old syntax to configure each layer as a dictionary with the key as the name is deprecated.

```
layers:
  mylayer:
    title: My Layer
    source: [mysource]
```

should become

```
layers:
  - name: mylayer
    title: My Layer
    source: [mysource]
```

The mixed format where the layers are a list (-) but each layer is still a dictionary is no longer supported (e.g. `- mylayer:` becomes `- name: mylayer`).

5.3.1 name

The name of the layer. You can omit the name for group layers (e.g. layers with `layers`), in this case the layer is not addressable in WMS and used only for grouping.

5.3.2 title

Readable name of the layer, e.g WMS layer title.

5.3.3 layers

Each layer can contain another `layers` configuration. You can use this to build group layers and to build a nested layer tree.

For example:

```
layers:
  - name: root
    title: Root Layer
    layers:
      - name: layer1
        title: Title of Layer 1
        layers:
          - name: layer1a
            title: Title of Layer 1a
            sources: [source1a]
          - name: layer1b
            title: Title of Layer 1b
            sources: [source1b]
      - name: layer2
        title: Title of Layer 2
        sources: [cache2]
```

`root` and `layer1` is a group layer in this case. The WMS service will render `layer1a` and `layer1b` if you request `layer1`. Note that `sources` is optional if you supply `layers`. You can still configure `sources` for group layers. In this case the group `sources` will replace the `sources` of the child layers.

MapProxy will wrap all layers into an unnamed root layer, if you define multiple layers on the first level.

Note: The old syntax (see `name` *above*) is not supported if you use the nested layer configuration format.

5.3.4 sources

A list of data sources for this layer. You can use sources defined in the `sources` and `caches` section. MapProxy will merge multiple sources from left (bottom) to right (top).

WMS and Mapserver sources also support tagged names (`wms:lyr1,lyr2`). See *Tagged source names*.

5.3.5 tile_sources

New in version 1.8.2.

A list of caches for this layer. This list overrides `sources` for WMTS and TMS. `tile_sources` are not merged like `sources`, instead all the caches are added as additional tile (matrix) sets.

5.3.6 min_res, max_res or min_scale, max_scale

Limit the layer to the given min and max resolution or scale. MapProxy will return a blank image for requests outside of these boundaries (`min_res` is inclusive, `max_res` exclusive). You can use either the resolution or the scale values, missing values will be interpreted as *unlimited*. Resolutions should be in meters per pixel.

The values will also apear in the capabilities documents (i.e. WMS ScaleHint and Min/MaxScaleDenominator).

Pleas read *scale vs. resolution* for some notes on *scale*.

5.3.7 legendurl

Configure a URL to an image that should be returned as the legend for this layer. Local URLs (`file://`) are also supported. MapProxy ignores the legends from the sources of this layer if you configure a `legendurl` here.

5.3.8 md

New in version 1.4.0.

Add additional metadata for this layer. This metadata appears in the WMS 1.3.0 capabilities documents. Refer to the OGC 1.3.0 specification for a description of each option.

See also *INSPIRE View Service* for configuring additional INSPIRE metadata.

Here is an example layer with extended layer capabilities:

```
layers:
  - name: md_layer
    title: WMS layer with extended capabilities
    sources: [wms_source]
    md:
      abstract: Some abstract
      keyword_list:
        - vocabulary: Name of the vocabulary
          keywords:   [keyword1, keyword2]
        - vocabulary: Name of another vocabulary
          keywords:   [keyword1, keyword2]
        - keywords:   ["keywords without vocabulary"]
      attribution:
        title: My attribution title
        url:   http://example.org/
      logo:
         url:    http://example.org/logo.jpg
         width:  100
         height: 100
         format: image/jpeg
```

```
    identifier:
      - url:    http://example.org/
        name:   HKU1234
        value:  Some value
    metadata:
      - url:    http://example.org/metadata2.xml
        type:   INSPIRE
        format: application/xml
      - url:    http://example.org/metadata2.xml
        type:   ISO19115:2003
        format: application/xml
    data:
      - url:    http://example.org/datasets/test.shp
        format: application/octet-stream
      - url:    http://example.org/datasets/test.gml
        format: text/xml; subtype=gml/3.2.1
    feature_list:
      - url:    http://example.org/datasets/test.pdf
        format: application/pdf
```

5.3.9 dimensions

New in version 1.6.0.

Note: Dimensions are only supported for uncached WMTS services for now. See *WMTS service with dimensions* for a working use-case.

Configure the dimensions that this layer supports. Dimensions should be a dictionary with one entry for each dimension. Each dimension is another dictionary with a list of `values` and an optional `default` value. When the `default` value is omitted, the last value will be used.

```
layers:
  - name: dimension_layer
    title: layer with dimensions
    sources: [cache]
    dimensions:
      time:
        values:
          - "2012-11-12T00:00:00"
          - "2012-11-13T00:00:00"
          - "2012-11-14T00:00:00"
          - "2012-11-15T00:00:00"
        default: "2012-11-15T00:00:00"
      elevation:
        values:
          - 0
          - 1000
          - 3000
```

5.4 caches

Here you can configure which sources should be cached. Available options are:

5.4.1 sources

A list of data sources for this cache. You can use sources defined in the `sources` and `caches` section. This parameter is *required*. MapProxy will merge multiple sources from left (bottom) to right (top) before they are

stored on disk.

```
caches:
  my_cache:
    sources: [background_wms, overlay_wms]
    ...
```

WMS and Mapserver sources also support tagged names (`wms:lyr1,lyr2`). See *Tagged source names*.

Band merging

New in version 1.9.0.

You can also define a list of sources for each color band. The target color bands are specified as `r`, `g`, `b` for RGB images, optionally with `a` for the alpha band. You can also use `l` (luminance) to create tiles with a single color band (e.g. grayscale images).

You need to define the `source` and the `band` index for each source band. The indices of the source bands are numeric and start from 0.

The following example creates a colored infra-red (false-color) image by using near infra-red for red, red (band 0) for green, and green (band 1) for blue:

```
caches:
  cir_cache:
     sources:
         r: [{source: nir_cache, band: 0}]
         g: [{source: dop_cache, band: 0}]
         b: [{source: dop_cache, band: 1}]
```

You can define multiple sources for each target band. The values are summed and clipped at 255. An optional `factor` allows you to reduce the values. You can use this to mix multiple bands into a single grayscale image:

```
caches:
 grayscale_cache:
     sources:
         l: [
             {source: dop_cache, band: 0, factor: 0.21},
             {source: dop_cache, band: 1, factor: 0.72},
             {source: dop_cache, band: 2, factor: 0.07},
         ]
```

Cache sources

New in version 1.5.0.

You can also use other caches as a source. MapProxy loads tiles directly from that cache if the grid of the target cache is identical or *compatible* with the grid of the source cache. You have a compatible grid when all tiles in the cache grid are also available in source grid, even if the tile coordinates (X/Y/Z) are different.

When the grids are not compatible, e.g. when they use different projections, then MapProxy will access the source cache as if it is a WMS source and it will use meta-requests and do image reprojection as necessary.

See *Using existing caches* for more information.

5.4.2 `format`

The internal image format for the cache. Available options are `image/png`, `image/jpeg` etc. and `mixed`. The default is `image/png`.

New in version 1.5.0.

With the `mixed` format, MapProxy stores tiles as either PNG or JPEG, depending on the transparency of each tile. Images with transparency will be stored as PNG, fully opaque images as JPEG. You need to set the `request_format` to `image/png` when using `mixed`-mode:

```
caches:
  mixed_mode_cache:
    format: mixed
    request_format: image/png
    ...
```

5.4.3 request_format

MapProxy will try to use this format to request new tiles, if it is not set `format` is used. This option has no effect if the source does not support that format or the format of the source is set explicitly (see `suported_format` or `format` for sources).

5.4.4 link_single_color_images

If set to `true`, MapProxy will not store tiles that only contain a single color as a separate file. MapProxy stores these tiles only once and uses symbolic links to this file for every occurrence. This can reduce the size of your tile cache if you have larger areas with no data (e.g. water areas, areas with no roads, etc.).

Note: This feature is only available on Unix, since Windows has no support for symbolic links.

5.4.5 minimize_meta_requests

If set to `true`, MapProxy will only issue a single request to the source. This option can reduce the request latency for uncached areas (on demand caching).

By default MapProxy requests all uncached meta-tiles that intersect the requested bbox. With a typical configuration it is not uncommon that a requests will trigger four requests each larger than 2000x2000 pixel. With the `minimize_meta_requests` option enabled, each request will trigger only one request to the source. That request will be aligned to the next tile boundaries and the tiles will be cached.

5.4.6 watermark

Add a watermark right into the cached tiles. The watermark is thus also present in TMS or KML requests.

text The watermark text. Should be short.

opacity The opacity of the watermark (from 0 transparent to 255 full opaque). Use a value between 30 and 100 for unobtrusive watermarks.

font_size Font size of the watermark text.

color Color of the watermark text. Default is grey which works good for vector images. Can be either a list of color values (`[255, 255, 255]`) or a hex string (`#ffffff`).

spacing Configure the spacing between repeated watermarks. By default the watermark will be placed on every tile, with `wide` the watermark will be placed on every second tile.

5.4.7 grids

You can configure one or more grids for each cache. MapProxy will create one cache for each grid.

```
grids: ['my_utm_grid', 'GLOBAL_MERCATOR']
```

MapProxy supports on-the-fly transformation of requests with different SRSs. So it is not required to add an extra cache for each supported SRS. For best performance only the SRS most requests are in should be used.

There is some special handling for layers that need geographical and projected coordinate systems. For example, if you set one grid with `EPSG:4326` and one with `EPSG:3857` then all requests for projected SRS will access the `EPSG:3857` cache and requests for geographical SRS will use `EPSG:4326`.

5.4.8 `meta_size` and `meta_buffer`

Change the `meta_size` and `meta_buffer` of this cache. See *global cache options* for more details.

5.4.9 `bulk_meta_tiles`

Enables meta-tile handling for tiled sources. See *global cache options* for more details.

5.4.10 `image`

See below for all image options.

5.4.11 `use_direct_from_level` and `use_direct_from_res`

You can limit until which resolution MapProxy should cache data with these two options. Requests below the configured resolution or level will be passed to the underlying source and the results will not be stored. The resolution of `use_direct_from_res` should use the units of the first configured grid of this cache. This takes only effect when used in WMS services.

5.4.12 `disable_storage`

If set to `true`, MapProxy will not store any tiles for this cache. MapProxy will re-request all required tiles for each incoming request, even if the there are matching tiles in the cache. See *seed_only* if you need an *offline* mode.

Note: Be careful when using a cache with disabled storage in tile services when the cache uses WMS sources with metatiling.

5.4.13 `cache_dir`

Directory where MapProxy should store tiles for this cache. Uses the value of `globals.cache.base_dir` by default. MapProxy will store each cache in a subdirectory named after the cache and the grid SRS (e.g. `cachename_EPSG1234`). See *directory option* on how configure a complete path.

5.4.14 `cache`

New in version 1.2.0.

Configure the type of the background tile cache. You configure the type with the `type` option. The default type is `file` and you can leave out the `cache` option if you want to use the file cache. Read *Caches* for a detailed list of all available cache backends.

5.4.15 Example `caches` configuration

```
caches:
 simple:
   source: [mysource]
   grids: [mygrid]
 fullexample:
   source: [mysource, mysecondsource]
   grids: [mygrid, mygrid2]
   meta_size: [8, 8]
   meta_buffer: 256
   watermark:
     text: MapProxy
   request_format: image/tiff
   format: image/jpeg
   cache:
     type: file
     directory_layout: tms
```

5.5 grids

Here you can define the tile grids that MapProxy uses for the internal caching. There are multiple options to define the grid, but beware, not all are required at the same time and some combinations will result in ambiguous results.

There are three pre-defined grids all with global coverage:

- `GLOBAL_GEODETIC`: EPSG:4326, origin south-west, compatible with OpenLayers map in EPSG:4326
- `GLOBAL_MERCATOR`: EPSG:900913, origin south-west, compatible with OpenLayers map in EPSG:900913
- `GLOBAL_WEBMERCATOR`: similar to `GLOBAL_MERCATOR` but uses EPSG:3857 and origin north-west, compatible with OpenStreetMap/etc.

New in version 1.6.0: `GLOBAL_WEBMERCATOR`

5.5.1 `name`

Overwrite the name of the grid used in WMTS URLs. The name is also used in TMS and KML URLs when the `use_grid_names` option of the services is set to `true`.

5.5.2 `srs`

The spatial reference system used for the internal cache, written as `EPSG:xxxx`.

5.5.3 `tile_size`

The size of each tile. Defaults to 256x256 pixel.

```
tile_size: [512, 512]
```

5.5.4 `res`

A list with all resolutions that MapProxy should cache.

```
res: [1000, 500, 200, 100]
```

5.5.5 res_factor

Here you can define a factor between each resolution. It should be either a number or the term `sqrt2`. `sqrt2` is a shorthand for a resolution factor of 1.4142, the square root of two. With this factor the resolution doubles every second level. Compared to the default factor 2 you will get another cached level between all standard levels. This is suited for free zooming in vector-based layers where the results might look to blurry/pixelated in some resolutions.

For requests with no matching cached resolution the next best resolution is used and MapProxy will transform the result.

5.5.6 threshold_res

A list with resolutions at which MapProxy should switch from one level to another. MapProxy automatically tries to determine the optimal cache level for each request. You can tweak the behavior with the `stretch_factor` option (see below).

If you need explicit transitions from one level to another at fixed resolutions, then you can use the `threshold_res` option to define these resolutions. You only need to define the explicit transitions.

Example: You are caching at 1000, 500 and 200m/px resolutions and you are required to display the 1000m/px level for requests with lower than 700m/px resolutions and the 500m/px level for requests with higher resolutions. You can define that transition as follows:

```
res: [1000, 500, 200]
threshold_res: [700]
```

Requests with 1500, 1000 or 701m/px resolution will use the first level, requests with 700 or 500m/px will use the second level. All other transitions (between 500 an 200m/px in this case) will be calculated automatically with the `stretch_factor` (about 416m/px in this case with a default configuration).

5.5.7 bbox

The extent of your grid. You can use either a list or a string with the lower left and upper right coordinates. You can set the SRS of the coordinates with the `bbox_srs` option. If that option is not set the `srs` of the grid will be used.

MapProxy always expects your BBOX coordinates order to be west, south, east, north regardless of your SRS *axis order*.

```
bbox: [0, 40, 15, 55]
  or
bbox: "0,40,15,55"
```

5.5.8 bbox_srs

The SRS of the grid bbox. See above.

5.5.9 origin

New in version 1.3.0.

The default origin (x=0, y=0) of the tile grid is the lower left corner, similar to TMS. WMTS defines the tile origin in the upper left corner. MapProxy can translate between services and caches with different tile origins, but there are some limitations for grids with custom BBOX and resolutions that are not of factor 2. You can only use one service in these cases and need to use the matching `origin` for that service.

The following values are supported:

ll or sw: If the x=0, y=0 tile is in the lower-left/south-west corner of the tile grid. This is the default.

ul or nw: If the x=0, y=0 tile is in the upper-left/north-west corner of the tile grid.

5.5.10 num_levels

The total number of cached resolution levels. Defaults to 20, except for grids with `sqrt2` resolutions. This option has no effect when you set an explicit list of cache resolutions.

5.5.11 min_res and max_res

The the resolutions of the first and the last level.

5.5.12 stretch_factor

MapProxy chooses the *optimal* cached level for requests that do not exactly match any cached resolution. MapProxy will stretch or shrink images to the requested resolution. The *stretch_factor* defines the maximum factor MapProxy is allowed to stretch images. Stretched images result in better performance but will look blurry when the value is to large (> 1.2).

Example: Your MapProxy caches 10m and 5m resolutions. Requests with 9m resolution will be generated from the 10m level, requests for 8m from the 5m level.

5.5.13 max_shrink_factor

This factor only applies for the first level and defines the maximum factor that MapProxy will shrink images.

Example: Your MapProxy layer starts with 1km resolution. Requests with 3km resolution will get a result, requests with 5km will get a blank response.

5.5.14 base

With this option, you can base the grid on the options of another grid you already defined.

5.5.15 Defining Resolutions

There are multiple options that influence the resolutions MapProxy will use for caching: `res`, `res_factor`, `min_res`, `max_res`, `num_levels` and also `bbox` and `tile_size`. We describe the process MapProxy uses to build the list of all cache resolutions.

If you supply a list with resolution values in `res` then MapProxy will use this list and will ignore all other options.

If `min_res` is set then this value will be used for the first level, otherwise MapProxy will use the resolution that is needed for a single tile (`tile_size`) that contains the whole `bbox`.

If you have `max_res` and `num_levels`: The resolutions will be distributed between `min_res` and `max_res`, both resolutions included. The resolutions will be logarithmical, so you will get a constant factor between each resolution. With resolutions from 1000 to 10 and 6 levels you would get 1000, 398, 158, 63, 25, 10 (rounded here for readability).

If you have `max_res` and `res_factor`: The resolutions will be multiplied by `res_factor` until larger then `max_res`.

If you have `num_levels` and `res_factor`: The resolutions will be multiplied by `res_factor` for up to `num_levels` levels.

5.5.16 Example `grids` configuration

```
grids:
  localgrid:
    srs: EPSG:31467
    bbox: [5,50,10,55]
    bbox_srs: EPSG:4326
    min_res: 10000
    res_factor: sqrt2
  localgrid2:
    base: localgrid
    srs: EPSG:25832
    tile_size: [512, 512]
```

5.6 sources

A sources defines where MapProxy can request new data. Each source has a `type` and all other options are dependent to this type.

See *Sources* for the documentation of all available sources.

An example:

```
sources:
  sourcename:
    type: wms
    req:
      url: http://localhost:8080/service?
      layers: base
  anothersource:
    type: wms
    # ...
```

5.7 globals

Here you can define some internals of MapProxy and default values that are used in the other configuration directives.

5.7.1 `image`

Here you can define some options that affect the way MapProxy generates image results.

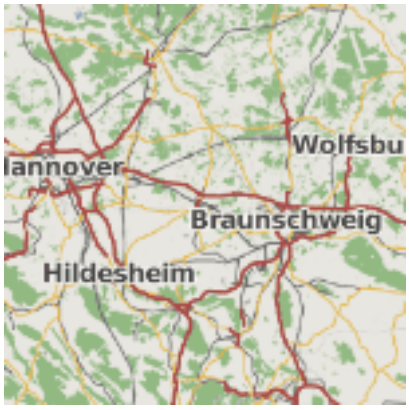
`resampling_method` The resampling method used when results need to be rescaled or transformed. You can use one of nearest, bilinear or bicubic. Nearest is the fastest and bicubic the slowest. The results will look best with bilinear or bicubic. Bicubic enhances the contrast at edges and should be used for vector images.

With *bilinear* you should get about 2/3 of the *nearest* performance, with *bicubic* 1/3.

See the examples below:

```
nearest:
```

`bilinear`:



`bicubic`:

`paletted` Enable paletted (8bit) PNG images. It defaults to `true` for backwards compatibility. You should set this to `false` if you need 24bit PNG files. You can enable 8bit PNGs for single caches with a custom format (`colors: 256`).

`formats` Modify existing or define new image formats. *See below* for all image format options.

5.7.2 `cache`

The following options define how tiles are created and stored. Most options can be set individually for each cache as well.

New in version 1.6.0: `tile_lock_dir`

New in version 1.10.0: `bulk_meta_tiles`

`meta_size` MapProxy does not make a single request for every tile it needs, but it will request a large meta-tile that consist of multiple tiles. `meta_size` defines how large a meta-tile is. A `meta_size` of `[4, 4]`

will request 16 tiles in one pass. With a tile size of 256x256 this will result in 1024x1024 requests to the source. Tiled sources are still requested tile by tile, but you can configure MapProxy to load multiple tiles in bulk with *bulk_meta_tiles*.

bulk_meta_tiles Enables meta-tile handling for caches with tile sources. If set to *true*, MapProxy will request neighboring tiles from the source even if only one tile is requested from the cache. `meta_size` defines how many tiles should be requested in one step and `concurrent_tile_creators` defines how many requests are made in parallel. This option improves the performance for caches that allow to store multiple tiles with one request, like SQLite/MBTiles but not the `file` cache.

meta_buffer MapProxy will increase the size of each meta-tile request by this number of pixels in each direction. This can solve cases where labels are cut-off at the edge of tiles.

base_dir The base directory where all cached tiles will be stored. The path can either be absolute (e.g. `/var/mapproxy/cache`) or relative to the mapproxy.yaml file. Defaults to `./cache_data`.

lock_dir MapProxy uses locking to limit multiple request to the same service. See `concurrent_requests`. This option defines where the temporary lock files will be stored. The path can either be absolute (e.g. `/tmp/lock/mapproxy`) or relative to the mapproxy.yaml file. Defaults to `./cache_data/tile_locks`.

tile_lock_dir MapProxy uses locking to prevent that the same tile gets created multiple times. This option defines where the temporary lock files will be stored. The path can either be absolute (e.g. `/tmp/lock/mapproxy`) or relative to the mapproxy.yaml file. Defaults to `./cache_data/dir_of_the_cache/tile_locks`.

concurrent_tile_creators This limits the number of parallel requests MapProxy will make to a source. This limit is per request for this cache and not for all MapProxy requests. To limit the requests MapProxy makes to a single server use the `concurrent_requests` option.

Example: A request in an uncached region requires MapProxy to fetch four meta-tiles. A `concurrent_tile_creators` value of two allows MapProxy to make two requests to the source WMS request in parallel. The splitting of the meta-tile and the encoding of the new tiles will happen in parallel to.

link_single_color_images Enables the `link_single_color_images` option for all caches if set to `true`. See *link_single_color_images*.

max_tile_limit Maximum number of tiles MapProxy will merge together for a WMS request. This limit is for each layer and defaults to 500 tiles.

5.7.3 srs

proj_data_dir MapProxy uses Proj4 for all coordinate transformations. If you need custom projections or need to tweak existing definitions (e.g. add towgs parameter set) you can point MapProxy to your own set of proj4 init files. The path should contain an `epsg` file with the EPSG definitions.

The configured path can be absolute or relative to the mapproxy.yaml.

axis_order_ne and axis_order_en The axis ordering defines in which order coordinates are given, i.e. lon/lat or lat/lon. The ordering is dependent to the SRS. Most clients and servers did not respected the ordering and everyone used lon/lat ordering. With the WMS 1.3.0 specification the OGC emphasized that the axis ordering of the SRS should be used.

Here you can define the axis ordering of your SRS. This might be required for proper WMS 1.3.0 support if you use any SRS that is not in the default configuration.

By default MapProxy assumes lat/long (north/east) order for all geographic and x/y (east/north) order for all projected SRS.

You need to add the SRS name to the appropriate parameter, if that is not the case for your SRS.:

```
srs:
  # for North/East ordering
  axis_order_ne: ['EPSG:9999', 'EPSG:9998']
```

```
  # for East/North ordering
  axis_order_en: ['EPSG:0000', 'EPSG:0001']
```

If you need to override one of the default values, then you need to define both axis order options, even if one is empty.

5.7.4 http

HTTP related options.

Secure HTTP Connections (HTTPS)

MapProxy supports access to HTTPS servers. Just use `https` instead of `http` when defining the URL of a source.

MapProxy verifies the SSL/TLS connections against your systems "certification authority" (CA) certificates. You can provide your own set of root certificates with the `ssl_ca_certs` option. See the Python SSL documentation for more information about the format.

```
http:
  ssl_ca_certs: /etc/ssl/certs/ca-certificates.crt
```

New in version 1.11.0: MapProxy uses the systems CA files by default, if you use Python >=2.7.9 or >=3.4.

Note: You need to supply a CA file that includes the root certificates if you use older MapProxy or older Python versions. Otherwise MapProxy will fail to establish the connection. You can set the `http.ssl_no_cert_checks` options to `true` to disable this verification.

`ssl_no_cert_checks`

If you want to use SSL/TLS but do not need certificate verification, then you can disable it with the `ssl_no_cert_checks` option. You can also disable this check on a source level.

```
http:
  ssl_no_cert_checks: true
```

client_timeout

This defines how long MapProxy should wait for data from source servers. Increase this value if your source servers are slower.

method

Configure which HTTP method should be used for HTTP requests. By default (*AUTO*) MapProxy will use GET for most requests, except for requests with a long query string (e.g. WMS requests with *sld_body*) where POST is used instead. You can disable this behavior with either *GET* or *POST*.

```
http:
  method: GET
```

headers

Add additional HTTP headers to all requests to your sources.

```
http:
  headers:
    My-Header: header value
```

access_control_allow_origin

New in version 1.8.0.

Sets the `Access-control-allow-origin` header to HTTP responses for Cross-origin resource sharing. This header is required for WebGL or Canvas web clients. Defaults to *. Leave empty to disable the header. This option is only available in *globals*.

5.7.5 tiles

Configuration options for the TMS/Tile service.

expires_hours The number of hours a Tile is valid. TMS clients like web browsers will cache the tile for this time. Clients will try to refresh the tiles after that time. MapProxy supports the ETag and Last-Modified headers and will respond with the appropriate HTTP *'304 Not modified'* response if the tile was not changed.

5.7.6 mapserver

Options for the *Mapserver source*.

binary

The complete path to the `mapserv` executable. Required if you use the `mapserver` source.

working_dir

Path where the Mapserver should be executed from. It should be the directory where any relative paths in your mapfile are based on. Defaults to the directory of `binary`.

5.8 Image Format Options

New in version 1.1.0.

There are a few options that affect how MapProxy encodes and transforms images. You can set these options in the `globals` section or individually for each source or cache.

5.8.1 Options

Available options are:

format The mime-type of this image format. The format defaults to the name of the image configuration.

mode One of `RGB` for 24bit images, `RGBA` 32bit images with alpha, `P` for paletted images or `I` for integer images.

colors The number of colors to reduce the image before encoding. Use `0` to disable color reduction (quantizing) for this format and `256` for paletted images. See also *globals.image.paletted*.

transparent `true` if the image should have an alpha channel.

resampling_method The resampling method used for scaling or reprojection. One of `nearest`, `bilinear` or `bicubic`.

encoding_options Options that modify the way MapProxy encodes (saves) images. These options are format dependent. See below.

opacity Configures the opacity of a layer or cache. This value is used when the source or cache is placed on other layers and it can be used to overlay non-transparent images. It does not alter the image itself, and only effects when multiple layers are merged to one image. The value should be between 0.0 (full transparent) and 1.0 (opaque, i.e. the layers below will not be rendered).

encoding_options

The following encoding options are available:

jpeg_quality An integer value from 0 to 100 that defines the image quality of JPEG images. Larger values result in slower performance, larger file sizes but better image quality. You should try values between 75 and 90 for good compromise between performance and quality.

quantizer The algorithm used to quantize (reduce) the image colors. Quantizing is used for GIF and paletted PNG images. Available quantizers are `mediancut` and `fastoctree`. `fastoctree` is much faster and also supports 8bit PNG with full alpha support, but the image quality can be better with `mediancut` in some cases. The quantizing is done by the Python Image Library (PIL). `fastoctree` is a new quantizer that is only available in Pillow >=2.0. See *installation of PIL*.

5.8.2 Global

You can configure image formats globally with the `image.formats` option. Each format has a name and one or more options from the list above. You can choose any name, but you need to specify a `format` if the name is not a valid mime-type (e.g. `myformat` instead of `image/png`).

Here is an example that defines a custom format:

```
globals:
  image:
    formats:
      my_format:
        format: image/png
        mode: P
        transparent: true
```

You can also modify existing image formats:

```
globals:
  image:
    formats:
      image/png:
        encoding_options:
          quantizer: fastoctree
```

MapProxy will use your image formats when you are using the format name as the `format` of any source or cache.

For example:

```
caches:
  mycache:
    format: my_format
    sources: [source1, source2]
    grids: [my_grid]
```

5.8.3 Local

You can change all options individually for each cache or source. You can do this by choosing a base format and changing some options:

```
caches:
  mycache:
    format: image/jpeg
    image:
      encoding_options:
        jpeg_quality: 80
    sources: [source1, source2]
    grids: [my_grid]
```

You can also configure the format from scratch:

```
caches:
  mycache:
    image:
      format: image/jpeg
      resampling_method: nearest
    sources: [source1, source2]
    grids: [my_grid]
```

5.9 Notes

5.9.1 Scale vs. resolution

Scale is the ratio of a distance on a map and the corresponding distance on the ground. This implies that the map distance and the ground distance are measured in the same unit. For MapProxy a *map* is just a collection of pixels and the pixels do not have any size/dimension. They do correspond to a ground size but the size on the *map* is depended of the physical output format. MapProxy can thus only work with resolutions (pixel per ground unit) and not scales.

This applies to all servers and the OGC WMS standard as well. Some neglect this fact and assume a fixed pixel dimension (like 72dpi), the OCG WMS 1.3.0 standard uses a pixel size of 0.28 mm/px (around 91dpi). But you need to understand that a *scale* will differ if you print a map (200, 300 or more dpi) or if you show it on a computer display (typical 90-120 dpi, but there are mobile devices with more than 300 dpi).

You can convert between scales and resolutions with the *mapproxy-util scales tool*.

MapProxy will use the OCG value (0.28mm/px) if it's necessary to use a scale value (e.g. MinScaleDenominator in WMS 1.3.0 capabilities), but you should always use resolutions within MapProxy.

WMS ScaleHint

The WMS ScaleHint is a bit misleading. The parameter is not a scale but the diagonal pixel resolution. It also defines the `min` as the minimum value not the minimum resolution (e.g. 10m/px is a lower resolution than 5m/px, but 5m/px is the minimum value). MapProxy always uses the term resolutions as the side length in ground units per pixel and minimum resolution is always the higher number (100m/px < 10m/px). Keep that in mind when you use these values.

CHAPTER

SIX

SERVICES

The following services are available:

- *Web Map Service (OGC WMS)* and *WMS-C*
- *Tiled Map Services (TMS)*
- *Keyhole Markup Language (OGC KML)*
- *Web Map Tile Services (WMTS)*
- *MapProxy Demo Service*

You need to add the service to the `services` section of your MapProxy configuration to enable it. Some services take additional options.

```
services:
  tms:
  kml:
  wms:
    wmsoption1: xxx
    wmsoption2: xxx
```

6.1 Web Map Service (OGC WMS)

The WMS server is accessible at `/service`, `/ows` and `/wms` and it supports the WMS versions 1.0.0, 1.1.1 and 1.3.0.

See *INSPIRE View Service* for configuring INSPIRE metadata.

The WMS service will use all configured *layers*.

The service takes the following additional option.

6.1.1 `attribution`

Adds an attribution (copyright) line to all WMS requests.

`text` The text line of the attribution (e.g. some copyright notice, etc).

6.1.2 `md`

`md` is for metadata. These fields are used for the WMS `GetCapabilities` responses. See the example below for all supported keys.

New in version 1.8.1: `keyword_list`

6.1.3 srs

The `srs` option defines which SRS the WMS service supports.:

```
srs: ['EPSG:4326', 'CRS:84', 'EPSG:900913']
```

See *axis order* for further configuration that might be needed for WMS 1.3.0.

6.1.4 bbox_srs

New in version 1.3.0.

The `bbox_srs` option controls in which SRS the BBOX is advertised in the capabilities document. It should only contain SRS that are configured in the `srs` option.

You need to make sure that all layer extents are valid for these SRS. E.g. you can't choose a local SRS like UTM if you're using a global grid without limiting all sources with a `coverage`.

For example, a config with:

```
services:
  wms:
    srs: ['EPSG:4326', 'EPSG:3857', 'EPSG:31467']
    bbox_srs: ['EPSG:4326', 'EPSG:3857', 'EPSG:31467']
```

will show the bbox in the capabilities in EPSG:4326, EPSG:3857 and EPSG:31467.

New in version 1.7.0: You can also define an explicit bbox for specific SRS. This bbox will overwrite all layer extents for that SRS.

The following example will show the actual bbox of each layer in EPSG:4326 and EPSG:3857, but always the specified bbox for EPSG:31467:

```
services:
  wms:
    srs: ['EPSG:4326', 'EPSG:3857', 'EPSG:31467']
    bbox_srs:
      - 'EPSG:4326'
      - 'EPSG:3857'
      - srs: 'EPSG:31467'
        bbox: [2750000, 5000000, 4250000, 6500000]
```

You can use this to offer global datasets with SRS that are only valid in a local region, like UTM zones.

6.1.5 image_formats

A list of image mime types the server should offer.

6.1.6 featureinfo_types

A list of feature info types the server should offer. Available types are `text`, `html`, `xml` and `json`. The types are advertised in the capabilities with the correct mime type. Defaults to `[text, html, xml]`.

6.1.7 featureinfo_xslt

You can define XSLT scripts to transform outgoing feature information. You can define scripts for different feature info types:

`html` Define a script for `INFO_FORMAT=text/html` requests.

xml Define a script for `INFO_FORMAT=application/vnd.ogc.gml` and `INFO_FORMAT=text/xml` requests.

See *FeatureInformation for more informaiton*.

6.1.8 strict

Some WMS clients do not send all required parameters in feature info requests, MapProxy ignores these errors unless you set `strict` to `true`.

6.1.9 on_source_errors

Configure what MapProxy should do when one or more sources return errors or no response at all (e.g. timeout). The default is `notify`, which adds a text line in the image response for each erroneous source, but only if a least one source was successful. When `on_source_errors` is set to `raise`, MapProxy will return an OGC service exception in any error case.

6.1.10 max_output_pixels

New in version 1.3.0.

The maximum output size for a WMS requests in pixel. MapProxy returns an WMS exception in XML format for requests that are larger. Defaults to `[4000, 4000]` which will limit the maximum output size to 16 million pixels (i.e. 5000x3000 is still allowed).

See also *globals.cache.max_tile_limit* for the maximum number of tiles MapProxy will merge together for each layer.

6.1.11 versions

New in version 1.7.0.

A list of WMS version numbers that MapProxy should support. Defaults to `['1.0.0', '1.1.0', '1.1.1', '1.3.0']`.

6.1.12 Full example

```
services:
  wms:
    srs: ['EPSG:4326', 'CRS:83', 'EPSG:900913']
    versions: ['1.1.1']
    image_formats: ['image/png', 'image/jpeg']
    attribution:
      text: "© MyCompany"
    md:
      title: MapProxy WMS Proxy
      abstract: This is the fantastic MapProxy.
      online_resource: http://mapproxy.org/
      contact:
        person: Your Name Here
        position: Technical Director
        organization:
        address: Fakestreet 123
        city: Somewhere
        postcode: 12345
        state: XYZ
        country: Germany
```

```
    phone: +49(0)000-000000-0
    fax: +49(0)000-000000-0
    email: you@example.org
  access_constraints: This service is intended for private and evaluation use only.
  fees: 'None'
  keyword_list:
   - vocabulary: GEMET
     keywords:   [Orthoimagery]
   - keywords:   ["View Service", MapProxy]
```

6.1.13 WMS-C

The MapProxy WMS service also supports the WMS Tiling Client Recommendation from OSGeo.

If you add `tiled=true` to the GetCapabilities request, MapProxy will add metadata about the internal tile structure to the WMS capabilities document. Clients that support WMS-C can use this information to request tiles at the exact tile boundaries. MapProxy can return the tile as-it-is for these requests, the performace is on par with the TMS service.

MapProxy will limit the WMS support when `tiled=true` is added to the *GetMap* requests and it will return WMS service exceptions for requests that do not match the exact tile boundaries or if the requested image size or format differs.

6.2 Tiled Map Services (TMS)

MapProxy supports the Tile Map Service Specification from the OSGeo. The TMS is available at `/tms/1.0.0`.

The TMS service will use all configured *layers* that have a name and single cached source. Any layer grouping will be flattened.

Here is an example TMS request: `/tms/1.0.0/base/EPSG900913/3/1/0.png`. `png` is the internal format of the cached tiles. `base` is the name of the layer and `EPSG900913` is the SRS of the layer. The tiles are also available under the layer name `base_EPSG900913` when `use_grid_names` is false or unset.

A request to `/tms/1.0.0` will return the TMS metadata as XML. `/tms/1.0.0/layername` will return information about the bounding box, resolutions and tile size of this specific layer.

6.2.1 use_grid_names

New in version 1.5.0.

When set to *true*, MapProxy uses the actual name of the grid as the grid identifier instead of the SRS code. Tiles will then be available under `/tms/1.0.0/mylayer/mygrid/` instead of `/tms/1.0.0/mylayer/EPSG1234/` or `/tms/1.0.0/mylayer_EPSG1234/`.

6.2.2 Example

```
services:
  tms:
    use_grid_names: true
```

6.2.3 OpenLayers

When you create a map in OpenLayers with an explicit `mapExtent`, it will request only a single tile for the first (z=0) level. TMS begins with two or four tiles by default, depending on the SRS. MapProxy supports a different

TMS mode to support this use-case. MapProxy will start with a single-tile level if you request `/tiles` instead of `/tms`.

Alternatively, you can use the OpenLayers TMS option `zoomOffset` to compensate the difference. The option is available since OpenLayers 2.10.

There is an example available at *the configuration-examples section*, which shows the use of OpenLayers in combination with an overlay of tiles on top of OpenStreetMap tiles.

6.2.4 Google Maps

The TMS standard counts tiles starting from the lower left corner of the tile grid, while Google Maps and compatible services start at the upper left corner. The `/tiles` service accepts an `origin` parameter that flips the y-axis accordingly. You can set it to either `sw` (south-west), the default, or to `nw` (north-west), required for Google Maps.

Example:

```
http://localhost:8080/tiles/osm_EPSG900913/1/0/1.png?origin=nw
```

New in version 1.5.0: You can use the `origin` option of the TMS service to change the default origin of the tiles service. If you set it to `nw` then you can leave the `?origin=nw` parameter from the URL. This only works for the tiles service at `/tiles`, not for the TMS at `/tms/1.0.0/`.

Example:

```
services:
  tms:
    origin: 'nw'
```

6.3 Keyhole Markup Language (OGC KML)

MapProxy supports KML version 2.2 for integration into Google Earth. Each layer is available as a Super Overlay – image tiles are loaded on demand when the user zooms to a specific region. The initial KML file is available at `/kml/layername/EPSG1234/0/0/0.kml`. The tiles are also available under the layer name `layername_EPSG1234` when `use_grid_names` is false or unset.

New in version 1.5.0: The initial KML is also available at `/kml/layername_EPSG1234` and `/kml/layername/EPSG1234`.

6.3.1 use_grid_names

New in version 1.5.0.

When set to *true*, MapProxy uses the actual name of the grid as the grid identifier instead of the SRS code. Tiles will then be available under `/kml/mylayer/mygrid/` instead of `/kml/mylayer/EPSG1234/`.

6.3.2 Example

```
services:
  kml:
    use_grid_names: true
```

6.4 Web Map Tile Services (WMTS)

New in version 1.1.0.

MapProxy supports the OGC WMTS 1.0.0 specification.

The WMTS service is similar to the TMS service and will use all configured *layers* that have a name and single cached source. Any layer grouping will be flattened.

There are some limitations depending on the grid configuration you use. Please refer to *grid.origin* for more information.

The metadata (ServiceContact, etc.) of this service is taken from the WMS configuration. You can add `md` to the `wmts` configuration to replace the WMS metadata. See *WMS metadata*.

WMTS defines different access methods and MapProxy supports KVP and RESTful access. Both are enabled by default.

6.4.1 KVP

MapProxy supports `GetCapabilities` and `GetTile` KVP requests. The KVP service is available at `/service` and `/ows`.

You can enable or disable the KVP service with the `kvp` option. It is enabled by default and you need to enable `restful` if you disable this one.

```
services:
  wmts:
    kvp: false
    restful: true
```

6.4.2 RESTful

New in version 1.3.0.

MapProxy supports RESTful WMTS requests with custom URL templates. The RESTful service capabilities are available at `/wmts/1.0.0/WMTSCapabilities.xml`.

You can enable or disable the RESTful service with the `restful` option. It is enabled by default and you need to enable `kvp` if you disable this one.

```
services:
  wmts:
    restful: false
    kvp: true
```

URL Template

WMTS RESTful services supports custom tile URLs. You can configure your own URL template with the `restful_template` option.

The default template is `/{Layer}/{TileMatrixSet}/{TileMatrix}/{TileCol}/{TileRow}.{Format}`

The template variables are identical with the WMTS specification. `TileMatrixSet` is the grid name, `TileMatrix` is the zoom level, `TileCol` and `TileRow` are the x and y of the tile.

You can access the tile x=3, y=9, z=4 at `http://example.org//1.0.0/mylayer-mygrid/4-3-9/tile` with the following configuration:

```
services:
  wmts:
    restful: true
    restful_template:
        '/1.0.0/{Layer}-{TileMatrixSet}/{TileMatrix}-{TileCol}-{TileRow}/tile'
```

6.5 MapProxy Demo Service

MapProxy comes with a demo service that lists all configured WMS and TMS layers. You can test each layer with a simple OpenLayers client.

The service is available at `/demo/`.

This service takes no further options:

```
services:
    demo:
```

CHAPTER

SEVEN

SOURCES

MapProxy supports the following sources:

- *WMS*
- *ArcGIS REST API*
- *Tiles*
- *Mapserver*
- *Mapnik*
- *Debug*

You need to choose a unique name for each configured source. This name will be used to reference the source in the `caches` and `layers` configuration.

The sources section looks like:

```
sources:
  mysource1:
    type: xxx
    type_dependend_option1: a
    type_dependend_option2: b
  mysource2:
    type: yyy
    type_dependend_option3: c
```

See below for a detailed description of each service.

7.1 WMS

Use the type `wms` to for WMS servers.

7.1.1 req

This describes the WMS source. The only required options are `url` and `layers`. You need to set `transparent` to `true` if you want to use this source as an overlay.

```
req:
  url: http://example.org/service?
  layers: base,roads
  transparent: true
```

All other options are added to the query string of the request.

```
req:
  url: http://example.org/service?
  layers: roads
```

```
styles: simple
map: /path/to/mapfile
```

You can also configure `sld` or `sld_body` parameters, in this case you can omit `layers`. `sld` can also point to a `file://`-URL. MapProxy will read this file and use the content as the `sld_body`. See *sources with SLD* for more information.

You can omit layers if you use *Tagged source names*.

7.1.2 wms_opts

This option affects what request MapProxy sends to the source WMS server.

version The WMS version number used for requests (supported: 1.0.0, 1.1.0, 1.1.1, 1.3.0). Defaults to 1.1.1.

legendgraphic If this is set to `true`, MapProxy will request legend graphics from this source. Each MapProxy WMS layer that contains one or more sources with legend graphics will then have a LegendURL.

legendurl Configure a URL to an image that should be returned as the legend for this source. Local URLs (`file://`) are also supported.

map If this is set to `false`, MapProxy will not request images from this source. You can use this option in combination with `featureinfo: true` to create a source that is only used for feature info requests.

featureinfo If this is set to `true`, MapProxy will mark the layer as queryable and incoming *GetFeatureInfo* requests will be forwarded to the source server.

featureinfo_xslt Path to an XSLT script that should be used to transform incoming feature information.

featureinfo_format The `INFO_FORMAT` for FeatureInfo requests. By default MapProxy will use the same format as requested by the client.

`featureinfo_xslt` and `featureinfo_format`

See *FeatureInformation for more information*.

7.1.3 coverage

Define the covered area of the source. The source will only be requested if there is an intersection between the requested data and the coverage. See *coverages* for more information about the configuration. The intersection is calculated for meta-tiles and not the actual client request, so you should expect more visible data at the coverage boundaries.

7.1.4 seed_only

Disable this source in regular mode. If set to `true`, this source will always return a blank/transparent image. The source will only be requested during the seeding process. You can use this option to run MapProxy in an offline mode.

7.1.5 min_res, max_res or min_scale, max_scale

Limit the source to the given min and max resolution or scale. MapProxy will return a blank image for requests outside of these boundaries (`min_res` is inclusive, `max_res` exclusive). You can use either the resolution or the scale values, missing values will be interpreted as *unlimited*. Resolutions should be in meters per pixel.

The values will also apear in the capabilities documents (i.e. WMS ScaleHint and Min/MaxScaleDenominator). The boundaries will be regarded for each source, but the values in the capabilities might differ if you combine multiple sources or if the MapProxy layer already has a `min/max_res` configuration.

Please read *scale vs. resolution* for some notes on *scale*.

7.1.6 supported_srs

A list with SRSs that the WMS source supports. MapProxy will only query the source in these SRSs. It will reproject data if it needs to get data from this layer in any other SRS.

You don't need to configure this if you only use this WMS as a cache source and the WMS supports all SRS of the cache.

If MapProxy needs to reproject and the source has multiple `supported_srs`, then it will use the first projected SRS for requests in a projected SRS, or the first geographic SRS for requests in a geographic SRS. E.g when *supported_srs* is `['EPSG:4326', 'EPSG:31467']` caches with EPSG:3857 (projected, meter) will use EPSG:31467 (projected, meter) and not EPSG:4326 (geographic, lat/long).

7.1.7 forward_req_params

New in version 1.5.0.

A list with request parameters that will be forwarded to the source server (if available in the original request). A typical use case of this feature would be to forward the *TIME* parameter when working with a WMS-T server.

This feature only works with *uncached sources*.

7.1.8 supported_formats

Use this option to specify which image formats your source WMS supports. MapProxy only requests images in one of these formats, and will convert any image if it needs another format. If you do not supply this options, MapProxy assumes that the source supports all formats.

7.1.9 image

See *Image Format Options* for other options.

`transparent_color`

> Specify a color that should be converted to full transparency. Can be either a list of color values (`[255, 255, 255]`) or a hex string (`#ffffff`).

`transparent_color_tolerance`

> Tolerance for the `transparent_color` substitution. The value defines the tolerance in each direction. E.g. a tolerance of 5 and a color value of 100 will convert colors in the range of 95 to 105.
>
> ```
> image:
>   transparent_color: '#ffffff'
>   transparent_color_tolerance: 20
> ```

7.1.10 concurrent_requests

This limits the number of parallel requests MapProxy will issue to the source server. It even works across multiple WMS sources as long as all have the same `concurrent_requests` value and all `req.url` parameters point to the same host. Defaults to 0, which means no limitation.

7.1.11 http

You can configure the following HTTP related options for this source:

- method
- headers
- client_timeout
- ssl_ca_certs
- ssl_no_cert_checks

See *HTTP Options* for detailed documentation.

7.1.12 Tagged source names

New in version 1.1.0.

MapProxy supports tagged source names for most sources. This allows you to define the layers of a source in the caches or (WMS)-layers configuration.

Instead of referring to a source by the name alone, you can add a list of comma delimited layers: `sourcename:lyr1,lyr2`. You need to use quotes for tagged source names.

This works for layers and caches:

```
layers:
  - name: test
    title: Test Layer
    sources: ['wms1:lyr1,lyr2']

caches:
  cache1:
    sources: ['wms1:lyrA,lyrB']
    [...]

sources:
  wms1:
    type: wms
    req:
      url: http://example.org/service?
```

You can either omit the `layers` in the `req` parameter, or you can use them to limit the tagged layers. In this case MapProxy will raise an error if you configure `layers: lyr1,lyr2` and then try to access `wms:lyr2,lyr3` for example.

7.1.13 Example configuration

Minimal example:

```
my_minimal_wmssource:
  type: wms
  req:
    url: http://localhost:8080/service?
    layers: base
```

Full example:

```
my_wmssource:
  type: wms
  wms_opts:
    version: 1.0.0
```

```
    featureinfo: True
  supported_srs: ['EPSG:4326', 'EPSG:31467']
  image:
    transparent_color: '#ffffff'
    transparent_color_tolerance: 0
  coverage:
     polygons: GM.txt
     polygons_srs: EPSG:900913
  forward_req_params: ['TIME', 'CUSTOM']
  req:
    url: http://localhost:8080/service?mycustomparam=foo
    layers: roads
    another_param: bar
    transparent: true
```

7.2 ArcGIS REST API

Use the type `arcgis` for ArcGIS MapServer and ImageServer REST server endpoints. This source is based on *the WMS source* and most WMS options apply to the ArcGIS source too.

7.2.1 req

This describes the ArcGIS source. The only required option is `url`. You need to set `transparent` to `true` if you want to use this source as an overlay. You can also add ArcGIS specific parameters to `req`, for example to set the interpolation method for ImageServers.

7.2.2 opts

New in version 1.10.0.

New in version 1.11.0: `map` option

This option affects what request MapProxy sends to the source ArcGIS server.

featureinfo If this is set to `true`, MapProxy will mark the layer as queryable and incoming *GetFeatureInfo* requests will be forwarded as `identify` requests to the source server. ArcGIS REST server support only HTML and JSON format. You need to enable support for JSON *featureinfo_types*.

featureinfo_return_geometries Whether the source should include the feature geometries.

featureinfo_tolerance Tolerance in pixel within the ArcGIS server should identify features.

map If this is set to `false`, MapProxy will not request images from this source. You can use this option in combination with `featureinfo: true` to create a source that is only used for feature info requests.

7.2.3 seed_only

New in version 1.11.0.

See *seed_only*

7.2.4 Example configuration

MapServer example:

```
my_minimal_arcgissource:
  type: arcgis
  req:
    layers: show: 0,1
    url: http://example.org/ArcGIS/rest/services/Imagery/MapService
    transparent: true
```

ImageServer example:

```
my_arcgissource:
  type: arcgis
  coverage:
     polygons: GM.txt
     srs: EPSG:3857
  req:
    url: http://example.org/ArcGIS/rest/services/World/MODIS/ImageServer
    interpolation: RSP_CubicConvolution
    bandIds: 2,0,1
```

7.3 Tiles

Use the type `tile` to request data from from existing tile servers like TileCache and GeoWebCache. You can also use this source cascade MapProxy installations.

7.3.1 `url`

This source takes a `url` option that contains a URL template. The template format is `%(key_name)s`. MapProxy supports the following named variables in the URL:

x, y, z The tile coordinate.

format The format of the tile.

quadkey Quadkey for the tile as described in http://msdn.microsoft.com/en-us/library/bb259689.aspx

tc_path TileCache path like `09/000/000/264/000/000/345`. Note that it does not contain any format extension.

tms_path TMS path like `5/12/9`. Note that it does not contain the version, the layername or the format extension.

arcgiscache_path ArcGIS cache path like `L05/R00000123/C00000abc`. Note that it does not contain any format extension.

bbox Bounding box of the tile. For WMS-C servers that expect a fixed parameter order.

New in version 1.1.0: `arcgiscache_path` and `bbox` parameter.

7.3.2 `origin`

Deprecated since version 1.3.0: Use grid with the `origin` option.

The origin of the tile grid (i.e. the location of the 0,0 tile). Supported values are `sw` for south-west (lower-left) origin or `nw` for north-west (upper-left) origin. `sw` is the default.

7.3.3 `grid`

The grid of the tile source. Defaults to `GLOBAL_MERCATOR`, a grid that is compatible with popular web mapping applications.

7.3.4 coverage

Define the covered area of the source. The source will only be requested if there is an intersection between the incoming request and the coverage. See *coverages* for more information.

7.3.5 transparent

You need to set this to `true` if you want to use this source as an overlay.

7.3.6 http

You can configure the following HTTP related options for this source:

- `headers`
- `client_timeout`
- `ssl_ca_certs`
- `ssl_no_cert_checks`

See *HTTP Options* for detailed documentation.

7.3.7 seed_only

See *seed_only*

7.3.8 min_res, max_res or min_scale, max_scale

New in version 1.5.0.

See *min_res, max_res or min_scale, max_scale*.

7.3.9 on_error

New in version 1.4.0.

You can configure what MapProxy should do when the tile service returns an error. Instead of raising an error, MapProxy can generate a single color tile. You can configure if MapProxy should cache this tile, or if it should use it only to generate a tile or WMS response.

You can configure multiple status codes within the `on_error` option. You can also use the catch-all value `other`. This will not only catch all other HTTP status codes, but also source errors like HTTP timeouts or non-image responses.

Each status code takes the following options:

`response`

> Specify the color of the tile that should be returned in case of this error. Can be either a list of color values (`[255, 255, 255]`, `[255, 255, 255, 0]`)) or a hex string (`'#ffffff'`, `'#fa1fbb00'`) with RGBA values, or the string `transparent`.

`cache`

> Set this to `True` if MapProxy should cache the single color tile. Otherwise (`False`) MapProxy will use this generated tile only for this request. This is the default.

You need to enable `transparent` for your source, if you use `on_error` responses with transparency.

```
my_tile_source:
  type: tile
  url: http://localhost:8080/tiles/%(tms_path)s.png
  transparent: true
  on_error:
    204:
      response: transparent
      cache: True
    502:
      response: '#ede9e3'
      cache: False
    other:
      response: '#ff0000'
      cache: False
```

7.3.10 Example configuration

```
my_tile_source:
  type: tile
  grid: mygrid
  url: http://localhost:8080/tile?x=%(x)s&y=%(y)s&z=%(z)s&format=%(format)s
```

7.4 Mapserver

New in version 1.1.0.

Use the type `mapserver` to directly call the Mapserver CGI executable. This source is based on *the WMS source* and most options apply to the Mapserver source too.

The only differences are that it does not support the `http` option and the `req.url` parameter is ignored. The `req.map` should point to your Mapserver mapfile.

The mapfile used must have a WMS server enabled, e.g. with `wms_enable_request` or `ows_enable_request` in the mapfile.

7.4.1 `mapserver`

You can also set these options in the *globals* section.

`binary`

The complete path to the `mapserv` executable.

`working_dir`

Path where the Mapserver should be executed from. It should be the directory where any relative paths in your mapfile are based on.

New in version 1.11.0: The `mapserv` binary is searched in all directories of the `PATH` environment, if `binary` is not set.

7.4.2 Example configuration

```
my_ms_source:
  type: mapserver
  req:
    layers: base
    map: /path/to/my.map
```

```
  mapserver:
    binary: /usr/cgi-bin/mapserv
    working_dir: /path/to
```

7.5 Mapnik

New in version 1.1.0.

Changed in version 1.2.0: New `layers` option and support for *tagged sources*.

Use the type `mapnik` to directly call Mapnik without any WMS service. It uses the Mapnik Python API and you need to have a working Mapnik installation that is accessible by the Python installation that runs MapProxy. A call of `python -c 'import mapnik'` should return no error.

7.5.1 mapfile

The filename of you Mapnik XML mapfile.

7.5.2 layers

A list of layer names you want to render. MapProxy disables each layer that is not included in this list. It does not reorder the layers and unnamed layers (*Unknown*) are always rendered.

7.5.3 use_mapnik2

New in version 1.3.0.

Use Mapnik 2 if set to `true`. This option is now deprecated and only required for Mapnik 2.0.0. Mapnik 2.0.1 and newer are available as `mapnik` package.

7.5.4 transparent

Set to `true` to render from mapnik sources with background-color="transparent", `false` (default) will force a black background color.

7.5.5 scale_factor

New in version 1.8.0.

Set the Mapnik scale_factor option. Mapnik scales most style options like the width of lines and font sizes by this factor. See also *HQ/Retina tiles*.

7.5.6 Other options

The Mapnik source also supports the `min_res`/`max_res`/`min_scale`/`max_scale`, `concurrent_requests`, `seed_only` and `coverage` options. See *WMS*.

7.5.7 Example configuration

```
my_mapnik_source:
  type: mapnik
  mapfile: /path/to/mapnik.xml
```

7.6 Debug

Adds information like resolution and BBOX to the response image. This is useful to determine a fixed set of resolutions for the `res`-parameter. It takes no options.

Example:

```
debug_source:
  type: debug
```

CHAPTER

EIGHT

CACHES

New in version 1.2.0.

MapProxy supports multiple backends to store the internal tiles. The default backend is file based and does not require any further configuration.

8.1 Configuration

You can configure a backend for each cache with the `cache` option. Each backend has a `type` and one or more options.

```
caches:
  mycache:
    sources: [...]
    grids: [...]
    cache:
      type: backendtype
      backendoption1: value
      backendoption2: value
```

The following backend types are available.

- *file*
- *mbtiles*
- *sqlite*
- *geopackage*
- *couchdb*
- *riak*
- *redis*
- *s3*
- *compact*

8.2 `file`

This is the default cache type and it uses a single file for each tile. Available options are:

`directory_layout`: The directory layout MapProxy uses to store tiles on disk. Defaults to `tc` which uses a TileCache compatible directory layout (`zz/xxx/xxx/xxx/yyy/yyy/yyy.format`). `mp` uses a directory layout with less nesting (`zz/xxxx/xxxx/yyyy/yyyy.format`'). `tms` uses TMS compatible directories (`zz/xxxx/yyyy.format`). `quadkey` uses Microsoft Virtual Earth or quadkey compatible directories (see http://msdn.microsoft.com/en-us/library/bb259689.aspx). `arcgis` uses a direc-

tory layout with hexadecimal row and column numbers that is compatible to ArcGIS exploded caches (`Lzz/Rxxxxxxxx/Cyyyyyyyy.format`).

Note: `tms`, `quadkey` and `arcgis` layout are not suited for large caches, since it will create directories with thousands of files, which most file systems do not handle well.

---

use_grid_names: When `true` MapProxy will use the actual grid name in the path instead of the SRS code. E.g. tiles will be stored in `./cache_data/mylayer/mygrid/` instead of `./cache_data/mylayer/EPSG1234/`.

New in version 1.5.0.

directory: Directory where MapProxy should directly store the tiles. This will not add the cache name or grid name (`use_grid_name`) to the path. You can use this option to point MapProxy to an existing tile collection (created with `gdal2tiles` for example).

New in version 1.5.0.

tile_lock_dir: Directory where MapProxy should write lock files when it creates new tiles for this cache. Defaults to `cache_data/tile_locks`.

New in version 1.6.0.

8.3 mbtiles

Use a single SQLite file for this cache. It uses the MBTile specification.

Available options:

filename: The path to the MBTiles file. Defaults to `cachename.mbtiles`.

tile_lock_dir: Directory where MapProxy should write lock files when it creates new tiles for this cache. Defaults to `cache_data/tile_locks`.

New in version 1.6.0.

You can set the `sources` to an empty list, if you use an existing MBTiles file and do not have a source.

```
caches:
  mbtiles_cache:
    sources: []
    grids: [GLOBAL_MERCATOR]
    cache:
      type: mbtiles
      filename: /path/to/bluemarble.mbtiles
```

Note: The MBTiles format specification does not include any timestamps for each tile and the seeding function is limited therefore. If you include any `refresh_before` time in a seed task, all tiles will be recreated regardless of the value. The cleanup process does not support any `remove_before` times for MBTiles and it always removes all tiles. Use the `--summary` option of the `mapproxy-seed` tool.

The note about `bulk_meta_tiles` for SQLite below applies to MBtiles as well.

8.4 sqlite

New in version 1.6.0.

Use SQLite databases to store the tiles, similar to `mbtiles` cache. The difference to `mbtiles` cache is that the `sqlite` cache stores each level into a separate database. This makes it easy to remove complete levels during mapproxy-seed cleanup processes. The `sqlite` cache also stores the timestamp of each tile.

Available options:

dirname: The directory where the level databases will be stored.

tile_lock_dir: Directory where MapProxy should write lock files when it creates new tiles for this cache. Defaults to `cache_data/tile_locks`.

New in version 1.6.0.

```
caches:
  sqlite_cache:
    sources: [mywms]
    grids: [GLOBAL_MERCATOR]
    cache:
      type: sqlite
      directory: /path/to/cache
```

Note: New in version 1.10.0.

All tiles from a meta tile request are stored in one transaction into the SQLite file to increase performance. You need to activate the *bulk_meta_tiles* option to get the same benefit when you are using tiled sources.

```
caches:
  sqlite_cache:
    sources: [mytilesource]
    bulk_meta_tiles: true
    grids: [GLOBAL_MERCATOR]
    cache:
      type: sqlite
      directory: /path/to/cache
```

8.5 couchdb

New in version 1.3.0.

Store tiles inside a CouchDB. MapProxy creates a JSON document for each tile. This document contains metadata, like timestamps, and the tile image itself as a attachment.

8.5.1 Requirements

Besides a running CouchDB you will need the Python requests package. You can install it the usual way, for example `pip install requests`.

8.5.2 Configuration

You can configure the database and database name and the tile ID and additional metadata.

Available options:

url: The URL of the CouchDB server. Defaults to `http://localhost:5984`.

db_name: The name of the database MapProxy uses for this cache. Defaults to the name of the cache.

tile_lock_dir: Directory where MapProxy should write lock files when it creates new tiles for this cache. Defaults to `cache_data/tile_locks`.

New in version 1.6.0.

tile_id: Each tile document needs a unique ID. You can change the format with a Python format string that expects the following keys:

x, y, z: The tile coordinate.

grid_name: The name of the grid.

The default ID uses the following format:

```
%(grid_name)s-%(z)d-%(x)d-%(y)d
```

Note: You can't use slashes (/) in CouchDB IDs.

tile_metadata: MapProxy stores a JSON document for each tile in CouchDB and you can add additional key-value pairs with metadata to each document. There are a few predefined values that MapProxy will replace with tile-depended values, all other values will be added as they are.

Predefined values:

{{x}}, {{y}}, {{z}}: The tile coordinate.

{{timestamp}}: The creation time of the tile as seconds since epoch. MapProxy will add a `timestamp` key for you, if you don't provide a custom timestamp key.

{{utc_iso}}: The creation time of the tile in UTC in ISO format. For example: `2011-12-31T23:59:59Z`.

{{tile_centroid}}: The center coordinate of the tile in the cache's coordinate system as a list of long/lat or x/y values.

{{wgs_tile_centroid}}: The center coordinate of the tile in WGS 84 as a list of long/lat values.

8.5.3 Example

```
caches:
  mycouchdbcache:
    sources: [mywms]
    grids: [mygrid]
    cache:
      type: couchdb
      url: http://localhost:9999
      db_name: mywms_tiles
      tile_metadata:
        mydata: myvalue
        tile_col: '{{x}}'
        tile_row: '{{y}}'
        tile_level: '{{z}}'
        created_ts: '{{timestamp}}'
        created: '{{utc_iso}}'
        center: '{{wgs_tile_centroid}}'
```

MapProxy will place the JSON document for tile z=3, x=1, y=2 at `http://localhost:9999/mywms_tiles/mygrid-3-1-2`. The document will look like:

```
{
    "_attachments": {
        "tile": {
            "content_type": "image/png",
            "digest": "md5-ch4j5Piov6a5FlAZtwPVhQ==",
            "length": 921,
            "revpos": 2,
            "stub": true
        }
    },
    "_id": "mygrid-3-1-2",
    "_rev": "2-9932acafd060e10bc0db23231574f933",
    "center": [
```

```
        -112.5,
        -55.7765730186677
    ],
    "created": "2011-12-15T12:56:21Z",
    "created_ts": 1323953781.531889,
    "mydata": "myvalue",
    "tile_col": 1,
    "tile_level": 3,
    "tile_row": 2
}
```

The `_attachments`-part is the internal structure of CouchDB where the tile itself is stored. You can access the tile directly at: `http://localhost:9999/mywms_tiles/mygrid-3-1-2/tile`.

8.6 riak

New in version 1.6.0.

Store tiles in a Riak cluster. MapProxy creates keys with binary data as value and timestamps as user defined metadata. This backend is good for very large caches which can be distributed over many nodes. Data can be distributed over multiple nodes providing a fault-tolernt and high-available storage. A Riak cluster is masterless and each node can handle read and write requests.

8.6.1 Requirements

You will need the Python Riak client version 2.4.2 or older. You can install it in the usual way, for example with `pip install riak==2.4.2`. Environments with older version must be upgraded with `pip install -U riak==2.4.2`. Python library depends on packages *python-dev*, *libffi-dev* and *libssl-dev*.

8.6.2 Configuration

Available options:

nodes: A list of riak nodes. Each node needs a `host` and optionally a `pb_port` and an `http_port` if the ports differ from the default. Defaults to single localhost node.

protocol: Communication protocol. Allowed options is `http`, `https` and `pbc`. Defaults to `pbc`.

bucket: The name of the bucket MapProxy uses for this cache. The bucket is the namespace for the tiles and must be unique for each cache. Defaults to cache name suffixed with grid name (e.g. `mycache_webmercator`).

default_ports: Default `pb` and `http` ports for `pbc` and `http` protocols. Will be used as the default for each defined node.

secondary_index: If `true` enables secondary index for tiles. This improves seed cleanup performance but requires that Riak uses LevelDB as the backend. Refer to the Riak documentation. Defaults to `false`.

8.6.3 Example

```
myriakcache:
  sources: [mywms]
  grids: [mygrid]
  cache:
    type: riak
    nodes:
      - host: 1.example.org
        pb_port: 9999
```

```
    - host: 1.example.org
    - host: 1.example.org
  protocol: pbc
  bucket: myriakcachetiles
  default_ports:
    pb: 8087
    http: 8098
```

8.7 redis

New in version 1.10.0.

Store tiles in a Redis in-memory database. This backend is useful for short-term caching. Typical use-case is a small Redis cache that allows you to benefit from meta-tiling.

Your Redis database should be configured with `maxmemory` and `maxmemory-policy` options to limit the memory usage. For example:

```
maxmemory 256mb
maxmemory-policy volatile-ttl
```

8.7.1 Requirements

You will need the Python Redis client. You can install it in the usual way, for example with `pip install redis`.

8.7.2 Configuration

Available options:

host: Host name of the Redis server. Defaults to `127.0.0.1`.

port: Port of the Redis server. Defaults to `6379`.

db: Number of the Redis database. Please refer to the Redis documentation. Defaults to *0*.

prefix: The prefix added to each tile-key in the Redis cache. Used to distinguish tiles from different caches and grids. Defaults to `cache-name_grid-name`.

default_ttl: The default Time-To-Live of each tile in the Redis cache in seconds. Defaults to 3600 seconds (1 hour).

8.7.3 Example

```
redis_cache:
    sources: [mywms]
    grids: [mygrid]
    cache:
      type: redis
      default_ttl: 600
```

8.8 geopackage

New in version 1.10.0.

Store tiles in a geopackage database. MapProxy creates a tile table if one isn't defined and populates the required meta data fields. This backend is good for datasets that require portability. Available options:

filename: The path to the geopackage file. Defaults to `cachename.gpkg`.

table_name: The name of the table where the tiles should be stored (or retrieved if using an existing cache). Defaults to the `cachename_gridname`.

levels: Set this to true to cache to a directory where each level is stored in a separate geopackage. Defaults to `false`. If set to true, `filename` is ignored.

directory: If levels is true use this to specify the directory to store geopackage files.

You can set the `sources` to an empty list, if you use an existing geopackage file and do not have a source.

```
caches:
  geopackage_cache:
    sources: []
    grids: [GLOBAL_MERCATOR]
    cache:
      type: geopackage
      filename: /path/to/bluemarble.gpkg
      table_name: bluemarble_tiles
```

Note: The geopackage format specification does not include any timestamps for each tile and the seeding function is limited therefore. If you include any `refresh_before` time in a seed task, all tiles will be recreated regardless of the value. The cleanup process does not support any `remove_before` times for geopackage and it always removes all tiles. Use the `--summary` option of the `mapproxy-seed` tool.

8.9 s3

New in version 1.10.0.

Store tiles in a Amazon Simple Storage Service (S3).

8.9.1 Requirements

You will need the Python boto3 package. You can install it in the usual way, for example with `pip install boto3`.

8.9.2 Configuration

Available options:

bucket_name: The bucket used for this cache. You can set the default bucket with `globals.cache.s3.bucket_name`.

profile_name: Optional profile name for shared credentials for this cache. Alternative methods of authentification are using the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` environmental variables, or by using an IAM role when using an Amazon EC2 instance. You can set the default profile with `globals.cache.s3.profile_name`.

directory: Base directory (path) where all tiles are stored.

directory_layout: Defines the directory layout for the tiles (`12/12345/67890.png`, `L12/R00010932/C00003039.png`, etc.). See *file* for available options. Defaults to `tms` (e.g. `12/12345/67890.png`). This cache cache also supports `reverse_tms` where tiles are stored as `y/x/z.format`. See *note* below.

Note: The hierarchical `directory_layouts` can hit limitations of S3 *"if you are routinely processing 100 or more requests per second"*. `directory_layout: reverse_tms` can work around this limitation. Please read S3 Request Rate and Performance Considerations for more information on this issue.

8.9.3 Example

```
cache:
  my_layer_20110501_epsg_4326_cache_out:
    sources: [my_layer_20110501_cache]
    cache:
      type: s3
      directory: /1.0.0/my_layer/default/20110501/4326/
      bucket_name: my-s3-tiles-cache

globals:
  cache:
    s3:
      profile_name: default
```

8.10 compact

New in version 1.10.0: Support for format version 1

New in version 1.11.0: Support for format version 2

Store tiles in ArcGIS compatible compact cache files. A single compact cache `.bundle` file stores up to about 16,000 tiles.

Version 1 of the compact cache format is compatible with ArcGIS 10.0 and the default version of ArcGIS 10.0-10.2. Version 2 is supported by ArcGIS 10.3 or higher. Version 1 stores is one additional `.bundlx` index file for each `.bundle` data file.

Available options:

directory: Directory where MapProxy should store the level directories. This will not add the cache name or grid name to the path. You can use this option to point MapProxy to an existing compact cache.

version: The version of the ArcGIS compact cache format. This option is required. Either `1` or `2`.

You can set the `sources` to an empty list, if you use an existing compact cache files and do not have a source.

The following configuration will load tiles from `/path/to/cache/L00/R0000C0000.bundle`, etc.

```
caches:
  compact_cache:
    sources: []
    grids: [webmercator]
    cache:
      type: compact
      version: 2
      directory: /path/to/cache
```

Note: MapProxy does not support reading and writiting of the `conf.cdi` and `conf.xml` files. You need to configure a compatible MapProxy grid when you want to reuse exsting ArcGIS compact caches in MapProxy. You need to create or modify existing `conf.cdi` and `conf.xml` files when you want to use compact caches created with MapProxy in ArcGIS.

Note: The compact cache format does not include any timestamps for each tile and the seeding function is limited therefore. If you include any `refresh_before` time in a seed task, all tiles will be recreated regardless of the value. The cleanup process does not support any `remove_before` times for compact caches and it always removes all tiles. Use the `--summary` option of the `mapproxy-seed` tool.

Note: The compact cache format is append-only to allow parallel read and write operations. Removing or refreshing tiles with `mapproxy-seed` does not reduce the size of the cache files. You can use the *defrag-compact-cache* util to reduce the file size of existing bundle files.

CHAPTER

NINE

SEEDING

9.1 Introduction

The MapProxy creates all tiles on demand. To improve the performance for commonly requested areas it is possible to pre-generate these tiles. The `mapproxy-seed` script does this task.

The tool can seed one or more polygon or BBOX areas for each cache. It can seed missing tiles and refresh old tiles. A *cleanup* can be used to remove old tiles.

9.1.1 Method

MapProxy does not seed the tile pyramid level by level, but traverses the tile pyramid depth-first. It starts in the first zoom level and decides if the tiles in the next zoom level need to be seeded by checking each subtile for intersection with the coverage. It recursively repeats this step for all tiles below till it reaches the last zoom level to seed. Only then, before getting back to the parent tile, the tile is actually seeded.

The following shows in which order tiles are seeded for a simple cache with three levels:

```
Level 0 with 1 tile:

          21

Level 1 with 4 tiles:

         5  10
        15  20

Level 2 with 16 tiles:

      1  2  6  7
      3  4  8  9
     11 12 16 17
     13 14 18 19
```

This method is optimized to work *with* the caches of your operating system and geospatial database, as the same area is requested for multiple scales in direct succession.

It also makes checks against complex coverages efficient as subtiles can be rejected early on.

9.2 mapproxy-seed

The command line script expects a seed configuration that describes which tiles from which layer should be generated. See configuration for the format of the file.

9.2.1 Options

-s <seed.yaml>, **--seed-conf**==<seed.yaml>
The seed configuration. You can also pass the configuration as the last argument to `mapproxy-seed`

-f <mapproxy.yaml>, **--proxy-conf**=<mapproxy.yaml>
The MapProxy configuration to use. This file should describe all caches and grids that the seed configuration references.

-c N, **--concurrency** N
The number of concurrent seed worker. Some parts of the seed tool are CPU intensive (image splitting and encoding), use this option to distribute that load across multiple CPUs. To limit the concurrent requests to the source WMS see *concurrent_requests*

-n, --dry-run
This will simulate the seed/cleanup process without requesting, creating or removing any tiles.

--summary
Print a summary of all seeding and cleanup tasks and exit.

--quiet
Reduce the output of the progress logger.

-i, --interactive
Print a summary of each seeding and cleanup task and ask if `mapproxy-seed` should seed/cleanup that task. It will query for each task before it starts.

--seed=<task1,task2,..>
Only seed the named seeding tasks. You can select multiple tasks with a list of comma seperated names, or you can use the `--seed` option multiple times. You can use `ALL` to select all tasks. This disables all cleanup tasks unless you also use the `--cleanup` option.

--cleanup=<task1,task2,..>
Only cleanup the named tasks. You can select multiple tasks with a list of comma seperated names, or you can use the `--cleanup` option multiple times. You can use `ALL` to select all tasks. This disables all seeding tasks unless you also use the `--seed` option.

--continue
Continue an interrupted seed progress. MapProxy will start the seeding progress at the begining if the progress file (`--progress-file`) was not found. MapProxy can only continue if the previous seed was started with the `--progress-file` or `--continue` option.

--progress-file
Filename where MapProxy stores the seeding progress for the `--continue` option. Defaults to `.mapproxy_seed_progress` in the current working directory. MapProxy will remove that file after a successful seed.

--duration
Stop seeding process after this duration. This option accepts duration in the following format: 120s, 15m, 4h, 0.5d Use this option in combination with `--continue` to be able to resume the seeding. By default,

--reseed-file
File created by `mapproxy-seed` at the start of a new seeding.

--reseed-interval
Only start seeding if `--reseed-file` is older then this duration. This option accepts duration in the following format: 120s, 15m, 4h, 0.5d Use this option in combination with `--continue` to be able to resume the seeding. By default,

--use-cache-lock
Lock each cache to prevent multiple parallel *mapproxy-seed* calls to work on the same cache. It does not lock normal operation of MapProxy.

--log-config
The logging configuration file to use.

New in version 1.5.0: `--continue` and `--progress-file` option

New in version 1.7.0: `--log-config` option

New in version 1.10.0: `--duration`, `--reseed-file` and `--reseed-interval` option

9.2.2 Examples

Seed with concurrency of 4:

```
mapproxy-seed -f mapproxy.yaml -c 4 seed.yaml
```

Print summary of all seed tasks and exit:

```
mapproxy-seed -f mapproxy.yaml -s seed.yaml --summary --seed ALL
```

Interactively select which tasks should be seeded:

```
mapproxy-seed -f mapproxy.yaml -s seed.yaml -i
```

Seed task1 and task2 and cleanup task3 with concurrency of 2:

```
mapproxy-seed -f mapproxy.yaml -s seed.yaml -c 2 --seed task1,task2 \
 --cleanup task3
```

9.3 Configuration

Note: The configuration changed with MapProxy 1.0.0, the old format with `seeds` and `views` is still supported but will be deprecated in the future. See *below* for information about the old format.

The configuration is a YAML file with three sections:

`seeds` Configure seeding tasks.

`cleanups` Configure cleanup tasks.

`coverages` Configure coverages for seeding and cleanup tasks.

9.3.1 Example

```
seeds:
  myseed1:
    [...]
  myseed2
    [...]

cleanups:
  mycleanup1:
    [...]
  mycleanup2:
    [...]

coverages:
  mycoverage1:
    [...]
  mycoverage2:
    [...]
```

9.4 seeds

Here you can define multiple seeding tasks. A task defines *what* should be seeded. Each task is configured as a dictionary with the name of the task as the key. You can use the names to select single tasks on the command line of `mapproxy-seed`.

`mapproxy-seed` will always process one tile pyramid after the other. Each tile pyramid is defined by a cache and a corresponding grid. A cache with multiple grids consists of multiple tile pyramids. You can configure which tile pyramid you want to seed with the `caches` and `grids` options.

You can further limit the part of the tile pyramid with the `levels` and `coverages` options.

Each seed tasks takes the following options:

9.4.1 caches

A list with the caches that should be seeded for this task. The names should match the cache names in your MapProxy configuration.

9.4.2 grids

A list with the grid names that should be seeded for the `caches`. The names should match the grid names in your MapProxy configuration. All caches of this tasks need to support the grids you specify here. By default, the grids that are common to all configured caches will be seeded.

9.4.3 levels

Either a list of levels that should be seeded, or a dictionary with `from` and `to` that define a range of levels. You can omit `from` to start at level 0, or you can omit `to` to seed till the last level. By default, all levels will be seeded.

Examples:

```
# seed multiple levels
levels: [2, 3, 4, 8, 9]

# seed a single level
levels: [3]

# seed from level 0 to 10 (including level 10)
levels:
  to: 10

# seed from level 3 to 6 (including level 3 and 6)
levels:
  from: 3
  to: 6
```

9.4.4 coverages

A list with coverage names. Limits the seed area to the coverages. By default, the whole coverage of the grids will be seeded.

9.4.5 refresh_before

Regenerate all tiles that are older than the given date. The date can either be absolute or relative. By default, existing tiles will not be refreshed.

MapProxy can also use the last modification time of a file. File paths should be relative to the proxy configuration or absolute.

Examples:

```
# absolute as ISO time
refresh_before:
  time: 2010-10-21T12:35:00

# relative from the start time of the seed process
refresh_before:
  weeks: 1
  days: 7
  hours: 4
  minutes: 15

# modification time of a given file
refresh_before:
  mtime: path/to/file
```

9.4.6 Example

```
seeds:
  myseed1:
    caches: [osm_cache]
    coverages: [germany]
    grids: [GLOBAL_MERCATOR]
    levels:
      to: 10

  myseed2
    caches: [osm_cache]
    coverages: [niedersachsen, bremen, hamburg]
    grids: [GLOBAL_MERCATOR]
    refresh_before:
      weeks: 3
    levels:
      from: 11
      to: 15
```

9.5 cleanups

Here you can define multiple cleanup tasks. Each task is configured as a dictionary with the name of the task as the key. You can use the names to select single tasks on the command line of `mapproxy-seed`.

9.5.1 caches

A list with the caches where you want to cleanup old tiles. The names should match the cache names in your MapProxy configuration.

9.5.2 grids

A list with the grid names for the `caches` where you want to cleanup. The names should match the grid names in your MapProxy configuration. All caches of this tasks need to support the grids you specify here. By default, the grids that are common to all configured caches will be used.

9.5.3 levels

Either a list of levels that should be cleaned up, or a dictionary with `from` and `to` that define a range of levels. You can omit `from` to start at level 0, or you can omit `to` to cleanup till the last level. By default, all levels will be cleaned up.

Examples:

```
# cleanup multiple levels
levels: [2, 3, 4, 8, 9]

# cleanup a single level
levels: [3]

# cleanup from level 0 to 10 (including level 10)
levels:
  to: 10

# cleanup from level 3 to 6 (including level 3 and 6)
levels:
  from: 3
  to: 6
```

9.5.4 coverages

A list with coverage names. Limits the cleanup area to the coverages. By default, the whole coverage of the grids will be cleaned up.

Note: Be careful when cleaning up caches with large coverages and levels with lots of tiles (>14). Without `coverages`, the seed tool works on the file system level and it only needs to check for existing tiles if they should be removed. With `coverages`, the seed tool traverses the whole tile pyramid and needs to check every posible tile if it exists and if it should be removed. This is much slower.

9.5.5 remove_all

When set to true, remove all tiles regardless of the time they were created. You still limit the tiles with the `levels` and `coverage` options. MapProxy will try to remove tiles in a more efficient way with this option. For example: It will remove complete level directories for `file` caches instead of comparing each tile with a timestamp.

9.5.6 remove_before

Remove all tiles that are older than the given date. The date can either be absolute or relative. `remove_before` defaults to the start time of the seed process, so that newly created tile will not be removed.

MapProxy can also use the last modification time of a file. File paths should be relative to the proxy configuration or absolute.

Examples:

```
# absolute as ISO time
remove_before:
  time: 2010-10-21T12:35:00

# relative from the start time of the seed process
remove_before:
  weeks: 1
  days: 7
  hours: 4
  minutes: 15
```

```
# modification time of a given file
remove_before:
  mtime: path/to/file
```

9.5.7 Example

```
cleanups:
  highres:
    caches: [osm_cache]
    grids: [GLOBAL_MERCATOR, GLOBAL_SPERICAL]
    remove_before:
      days: 14
    levels:
      from: 16
  old_project:
    caches: [osm_cache]
    grids: [GLOBAL_MERCATOR]
    coverages: [mypolygon]
    levels:
      from: 14
      to: 18
```

9.6 coverages

There are three different ways to describe the extent of a seeding or cleanup task.

- a simple rectangular bounding box,
- a text file with one or more polygons in WKT format,
- polygons from any data source readable with OGR (e.g. Shapefile, GeoJSON, PostGIS)

Read the *coverage documentation* for more information.

Note: You will need to install additional dependencies, if you want to use polygons to define your geographical extent of the seeding area, instead of simple bounding boxes. See *coverage documentation*.

Each coverage has a name that is used in the seed and cleanup task configuration. If you don't specify a coverage for a task, then the BBOX of the grid will be used.

9.6.1 Example

```
coverages:
  germany:
    datasource: 'shps/world_boundaries_m.shp'
    where: 'CNTRY_NAME = "Germany"'
    srs: 'EPSG:900913'
  switzerland:
    datasource: 'polygons/SZ.txt'
    srs: 'EPSG:900913'
  austria:
    bbox: [9.36, 46.33, 17.28, 49.09]
    srs: 'EPSG:4326'
```

9.7 Output

`mapproxy-seed` prints out the progress of the current seeding task on the console.

Example progress log:

```
[16:48:26]  4  41.00% 582388, 4734701, 586740, 4737666 (5812 tiles)
```

The output starts with the current time and ends with the number of tiles it has seeded or removed so far. The third value is the current progress in percent. The progress can make large jumps, if the seeding detects that a tile and all its subtiles are outside of the seeding coverage. The second and fourth value show the level and bounding box of where the seeding tool is in this moment. Keep in mind, that it does not seed level by level. This is described in *seeding method*.

9.8 Example: Background seeding

New in version 1.10.0.

The `--duration` option allows you run MapProxy seeding for a limited time. In combination with the `--continue` option, you can resume the seeding process at a later time. You can use this to call `mapproxy-seed` with `cron` to seed in the off-hours.

However, this will restart the seeding process from the beginning every time the is seeding completed. You can prevent this with the `--reeseed-interval` and `--reseed-file` option. The following example starts seeding for six hours. It will seed for another six hours, every time you call this command again. Once all seed and cleanup tasks were processed the command will exit immediately every time you call it within 14 days after the first call. After 14 days, the modification time of the `reseed.time` file will be updated and the re-seeding process starts again.

```
mapproxy-seed -f mapproxy.yaml -s seed.yaml  \
  --reseed-interval 14d --duration 6h --reseed-file reseed.time \
  --continue --progress-file .mapproxy_seed_progress
```

You can use the `--reseed-file` as a `refresh_before` and `remove_before` `mtime`-file.

9.9 Old Configuration

Note: The following description is for the old seed configuration.

The configuration contains two keys: `views` and `seeds`. `views` describes the geographical extents that should be seeded. `seeds` links actual layers with those `views`.

9.9.1 Seeds

Contains a dictionary with layer/view mapping.:

```
seeds:
    cache1:
        views: ['world', 'germany', 'oldb']
    cache2:
        views: ['world', 'germany']
        remove_before:
            time: '2009-04-01T14:45:00'
            # or
            minutes: 15
            hours: 4
```

```
days: 9
weeks: 8
```

***remove_before*:** If present, recreate tiles if they are older than the date or time delta. At the end of the seeding process all tiles that are older will be removed.

You can either define a fixed time or a time delta. The *time* is a ISO-like date string (no time-zones, no abbreviations). To define time delta use one or more *seconds*, *minutes*, *hours*, *days* or *weeks* entries.

9.9.2 Views

Contains a dictionary with all views. Each view describes a coverage/geographical extent and the levels that should be seeded.

Coverages

Note: You will need to install additional dependencies, if you want to use polygons to define your geographical extent of the seeding area, instead of simple bounding boxes. See *coverage documentation*.

There are three different ways to describe the extent of the seed view.

- a simple rectangular bounding box,
- a text file with one or more polygons in WKT format,
- polygons from any data source readable with OGR (e.g. Shapefile, PostGIS)

Read the *coverage documentation* for more information.

9.9.3 Other options

`srs`: A list with SRSs. If the layer contains caches for multiple SRS, only the caches that match one of the SRS in this list will be seeded.

`res`: Seed until this resolution is cached.

or

`level`: A number until which this layer is cached, or a tuple with a range of levels that should be cached.

Example configuration

```
views:
  germany:
    datasource: 'shps/world_boundaries_m.shp'
    where: 'CNTRY_NAME = "Germany"'
    srs: 'EPSG:900913'
    level: [0, 14]
    srs: ['EPSG:900913', 'EPSG:4326']
  switzerland:
    datasource: 'polygons/SZ.txt'
    srs: EPSG:900913
    level: [0, 14]
    srs: ['EPSG:900913']
  austria:
    bbox: [9.36, 46.33, 17.28, 49.09]
    srs: EPSG:4326
    level: [0, 14]
    srs: ['EPSG:900913']
```

```
seeds:
  osm:
    views: ['germany', 'switzerland', 'austria']
    remove_before:
      time: '2010-02-20T16:00:00'
  osm_roads:
    views: ['germany']
    remove_before:
      days: 30
```

CHAPTER

TEN

COVERAGES

With coverages you can define areas where data is available or where data you are interested in is. MapProxy supports coverages for *sources* and in the *mapproxy-seed tool*. Refer to the corresponding section in the documentation.

There are five different ways to describe a coverage:

- a simple rectangular bounding box,
- a text file with one or more (multi)polygons in WKT format,
- a GeoJSON file with (multi)polygons features,
- (multi)polygons from any data source readable with OGR (e.g. Shapefile, GeoJSON, PostGIS),
- a file with webmercator tile coordinates.

New in version 1.10.

You can also build intersections, unions and differences between multiple coverages.

10.1 Requirements

If you want to use polygons to define a coverage, instead of simple bounding boxes, you will also need Shapely and GEOS. For loading polygons from shapefiles you'll also need GDAL/OGR.

MapProxy requires Shapely 1.2.0 or later and GEOS 3.1.0 or later.

On Debian:

```
sudo aptitude install libgeos-dev libgdal-dev
pip install Shapely
```

10.2 Configuration

All coverages are configured by defining the source of the coverage and the SRS. The configuration of the coverage depends on the type. The SRS can allways be configured with the `srs` option.

New in version 1.5.0: MapProxy can autodetect the type of the coverage. You can now use `coverage` instead of the `bbox`, `polygons` or `ogr_datasource` option. The old options are still supported.

10.3 Coverage Types

10.3.1 Bounding box

For simple box coverages.

bbox or datasource: A simple BBOX as a list of minx, miny, maxx, maxy, e.g: *[4, -30, 10, -28]* or as a string *4,-30,10,-28*.

10.3.2 Polygon file

Text files with one WKT polygon or multi-polygon per line. You can create your own files or use one of the files we provide for every country. Read the index to find your country.

datasource: The path to the polygon file. Should be relative to the proxy configuration or absolute.

10.3.3 GeoJSON

New in version 1.10: Previous versions required OGR/GDAL for reading GeoJSON.

You can use GeoJSON files with Polygon and MultiPolygons geometries. FeatureCollections and Features of these geometries are suppored as well. MapProxy uses OGR to read GeoJSON files if you define a `where` filter.

datasource: The path to the GeoJSON file. Should be relative to the proxy configuration or absolute.

10.3.4 OGR datasource

Any polygon datasource that is supported by OGR (e.g. Shapefile, GeoJSON, PostGIS).

datasource: The name of the datasource. Refer to the OGR format page for a list of all supported datasources. File paths should be relative to the proxy configuration or absolute.

where: Restrict which polygons should be loaded from the datasource. Either a simple where statement (e.g. `'CNTRY_NAME="Germany"'`) or a full select statement. Refer to the OGR SQL support documentation. If this option is unset, the first layer from the datasource will be used.

10.3.5 Expire tiles

New in version 1.10.

Text file with webmercator tile coordinates. The tiles should be in `z/x/y` format (e.g. `14/1283/6201`), with one tile coordinate per line. Only tiles in the webmercator grid are supported (origin is always *nw*).

expire_tiles: File or directory with expire tile files. Directories are loaded recursive.

10.3.6 Union

New in version 1.10.

A union coverage contains the combined coverage of one or more sub-coverages. This can be used to combine multiple coverages a single source. Each sub-coverage can be of any supported type and SRS.

union: A list of multiple coverages.

10.3.7 Difference

New in version 1.10.

A difference coverage subtracts the coverage of other sub-coverages from the first coverage. This can be used to exclude parts from a coverage. Each sub-coverage can be of any supported type and SRS.

difference: A list of multiple coverages.

10.3.8 Intersection

New in version 1.10.

An intersection coverage contains only areas that are covered by all sub-coverages. This can be used to limit a larger coverage to a smaller area. Each sub-coverage can be of any supported type and SRS.

`difference`: A list of multiple coverages.

10.4 Clipping

New in version 1.10.0.

By default MapProxy tries to get and serve full source image even if a coverage only touches it. Clipping by coverage can be enabled by setting `clip: true`. If enabled, all areas outside the coverage will be converted to transparent pixels.

The `clip` option is only active for source coverages and not for seeding coverages.

10.5 Examples

10.5.1 sources

Use the `coverage` option to define a coverage for a WMS or tile source.

```
sources:
  mywms:
    type: wms
    req:
      url: http://example.com/service?
      layers: base
    coverage:
      bbox: [5, 50, 10, 55]
      srs: 'EPSG:4326'
```

Example of an intersection coverage with clipping:

```
sources:
  mywms:
    type: wms
    req:
      url: http://example.com/service?
      layers: base
    coverage:
      clip: true
      intersection:
        - bbox: [5, 50, 10, 55]
          srs: 'EPSG:4326'
        - datasource: coverage.geojson
          srs: 'EPSG:4326'
```

10.5.2 mapproxy-seed

To define a seed-area in the `seed.yaml`, add the coverage directly to the view.

```
coverages:
  germany:
    datasource: 'shps/world_boundaries_m.shp'
```

```
    where: 'CNTRY_NAME = "Germany"'
    srs: 'EPSG:900913'
```

Here is the same example with a PostGIS source:

```
coverages:
  germany:
    datasource: "PG: dbname='db' host='host' user='user'
  password='password'"
    where: "select * from coverages where country='germany'"
    srs: 'EPSG:900913'
```

And here is an example with a GeoJSON source:

```
coverages:
  germany:
    datasource: 'boundary.geojson'
    srs: 'EPSG:4326'
```

See the OGR driver list for all supported formats.

CHAPTER

ELEVEN

MAPPROXY-UTIL

The commandline tool `mapproxy-util` provides sub-commands that are helpful when working with MapProxy.

To get a list of all sub-commands call:

```
mapproxy-util
```

To call a sub-command:

```
mapproxy-util subcommand
```

Each sub-command provides additional information:

```
mapproxy-util subcommand --help
```

The current sub-commands are:

- *create*
- *serve-develop*
- *serve-multiapp-develop*
- *scales*
- *wms-capabilities*
- *grids*
- *export*
- *defrag-compact-cache*
- `autoconfig` (see *mapproxy-util autoconfig*)

11.1 `create`

This sub-command creates example configurations for you. There are templates for each configuration file.

`-l`, `--list-templates`
: List names of all available configuration templates.

`-t` `<name>`**`, --template`** `<name>`
: Create a configuration with the named template.

`-f` `<mapproxy.yaml>`**`, --mapproxy-conf`** `<mapproxy.yaml>`
: The path of the MapProxy configuration. Required for some templates.

`--force`
: Overwrite any existing configuration with the same output filename.

11.1.1 Configuration templates

Available templates are:

base-config: Creates an example `mapproxy.yaml` and `seed.yaml` file. You need to pass the destination directory to the command.

log-ini: Creates an example logging configuration. You need to pass the target filename to the command.

wsgi-app: Creates an example server script for the given MapProxy configuration (*`--f/--mapproxy-conf`*) . You need to pass the target filename to the command.

11.1.2 Example

```
mapproxy-util create -t base-config ./
```

11.2 serve-develop

This sub-command starts a MapProxy instance of your configuration as a stand-alone server.

You need to pass the MapProxy configuration as an argument. The server will automatically reload if you change the configuration or any of the MapProxy source code.

-b <address>, **--bind** <address>
The server address where the HTTP server should listen for incomming connections. Can be a port (`:8080`), a host (`localhost`) or both (`localhost:8081`). The default is `localhost:8080`. You need to use `0.0.0.0` to be able to connect to the server from external clients.

11.2.1 Example

```
mapproxy-util serve-develop ./mapproxy.yaml
```

11.3 serve-multiapp-develop

New in version 1.3.0.

This sub-command is similar to `serve-develop` but it starts a *MultiMapProxy* instance.

You need to pass a directory of your MapProxy configurations as an argument. The server will automatically reload if you change any configuration or any of the MapProxy source code.

-b <address>, **--bind** <address>
The server address where the HTTP server should listen for incomming connections. Can be a port (`:8080`), a host (`localhost`) or both (`localhost:8081`). The default is `localhost:8080`. You need to use `0.0.0.0` to be able to connect to the server from external clients.

11.3.1 Example

```
mapproxy-util serve-multiapp-develop my_projects/
```

11.4 scales

New in version 1.2.0.

This sub-command helps to convert between scales and resolutions.

Scales are ambiguous when the resolution of the output device (LCD, printer, mobile, etc) is unknown and therefore MapProxy only uses resolutions for configuration (see *Scale vs. resolution*). You can use the `scales` sub-command to calculate between known scale values and resolutions.

The command takes a list with one or more scale values and returns the corresponding resolution value.

--unit <m|d>
: Return resolutions in this unit per pixel (default meter per pixel).

-l <n>, **--levels** <n>
: Calculate resolutions for `n` levels. This will double the resolution of the last scale value if `n` is larger than the number of the provided scales.

-d <dpi>, **--dpi** <dpi>
: The resolution of the output display to use for the calculation. You need to set this to the same value of the client/server software you are using. Common values are 72 and 96. The default value is the equivalent of a pixel size of .28mm, which is around 91 DPI. This is the value the OGC uses since the WMS 1.3.0 specification.

--as-res-config
: Format the output so that it can be pasted into a MapProxy grid configuration.

--res-to-scale
: Calculate from resolutions to scale.

11.4.1 Example

For multiple levels as MapProxy configuration snippet:

```
mapproxy-util scales -l 4 --as-res-config 100000

res: [
     #  res            level        scale
       28.0000000000, #  0      100000.00000000
       14.0000000000, #  1       50000.00000000
        7.0000000000, #  2       25000.00000000
        3.5000000000, #  3       12500.00000000
]
```

With multiple scale values and custom DPI:

```
mapproxy-util scales --dpi 96 --as-res-config \
    100000 50000 25000 10000

res: [
     #  res            level        scale
       26.4583333333, #  0      100000.00000000
       13.2291666667, #  1       50000.00000000
        6.6145833333, #  2       25000.00000000
        2.6458333333, #  3       10000.00000000
]
```

11.5 wms-capabilities

New in version 1.5.0.

This sub-command parses a valid capabilites document from a URL and displays all available layers.

This tool does not create a MapProxy configuration, but the output should help you to set up or modify your MapProxy configuration.

The command takes a valid URL GetCapabilities URL.

--host <URL>
Display all available Layers for this service. Each new layer will be marked with a hyphen and all sublayers are indented.

--version <versionnumber>
Parse the Capabilities-document for the given version. Only version 1.1.1 and 1.3.0 are supported. The default value is 1.1.1

11.5.1 Example

With the following MapProxy layer configuration:

```
layers:
  - name: osm
    title: Omniscale OSM WMS - osm.omniscale.net
    sources: [osm_cache]
  - name: foo
    title: Group Layer
    layers:
      - name: layer1a
        title: Title of Layer 1a
        sources: [osm_cache]
      - name: layer1b
        title: Title of Layer 1b
        sources: [osm_cache]
```

Parsed capabilities document:

```
mapproxy-util wms-capabilities http://127.0.0.1:8080/service?REQUEST=GetCapabilities

Capabilities Document Version 1.1.1
Root-Layer:
  - title: MapProxy WMS Proxy
    url: http://127.0.0.1:8080/service?
    opaque: False
    srs: ['EPSG:31467', 'EPSG:31466', 'EPSG:4326', 'EPSG:25831', 'EPSG:25833',
          'EPSG:25832', 'EPSG:31468', 'EPSG:900913', 'CRS:84', 'EPSG:4258']
    bbox:
        EPSG:900913: [-20037508.3428, -20037508.3428, 20037508.3428, 20037508.3428]
        EPSG:4326: [-180.0, -85.0511287798, 180.0, 85.0511287798]
    queryable: False
    llbbox: [-180.0, -85.0511287798, 180.0, 85.0511287798]
    layers:
      - name: osm
        title: Omniscale OSM WMS - osm.omniscale.net
        url: http://127.0.0.1:8080/service?
        opaque: False
        srs: ['EPSG:31467', 'EPSG:31466', 'EPSG:25832', 'EPSG:25831', 'EPSG:25833',
              'EPSG:4326', 'EPSG:31468', 'EPSG:900913', 'CRS:84', 'EPSG:4258']
        bbox:
            EPSG:900913: [-20037508.3428, -20037508.3428, 20037508.3428, 20037508.3428]
            EPSG:4326: [-180.0, -85.0511287798, 180.0, 85.0511287798]
        queryable: False
        llbbox: [-180.0, -85.0511287798, 180.0, 85.0511287798]
      - name: foobar
        title: Group Layer
```

```
    url: http://127.0.0.1:8080/service?
    opaque: False
    srs: ['EPSG:31467', 'EPSG:31466', 'EPSG:25832', 'EPSG:25831', 'EPSG:25833',
          'EPSG:4326', 'EPSG:31468', 'EPSG:900913', 'CRS:84', 'EPSG:4258']
    bbox:
        EPSG:900913: [-20037508.3428, -20037508.3428, 20037508.3428, 20037508.3428]
        EPSG:4326: [-180.0, -85.0511287798, 180.0, 85.0511287798]
    queryable: False
    llbbox: [-180.0, -85.0511287798, 180.0, 85.0511287798]
    layers:
      - name: layer1a
        title: Title of Layer 1a
        url: http://127.0.0.1:8080/service?
        opaque: False
        srs: ['EPSG:31467', 'EPSG:31466', 'EPSG:25832', 'EPSG:25831', 'EPSG:25833',
              'EPSG:4326', 'EPSG:31468', 'EPSG:900913', 'CRS:84', 'EPSG:4258']
        bbox:
            EPSG:900913: [-20037508.3428, -20037508.3428, 20037508.3428, 20037508.3428]
            EPSG:4326: [-180.0, -85.0511287798, 180.0, 85.0511287798]
        queryable: False
        llbbox: [-180.0, -85.0511287798, 180.0, 85.0511287798]
      - name: layer1b
        title: Title of Layer 1b
        url: http://127.0.0.1:8080/service?
        opaque: False
        srs: ['EPSG:31467', 'EPSG:31466', 'EPSG:25832', 'EPSG:25831', 'EPSG:25833',
              'EPSG:4326', 'EPSG:31468', 'EPSG:900913', 'CRS:84', 'EPSG:4258']
        bbox:
            EPSG:900913: [-20037508.3428, -20037508.3428, 20037508.3428, 20037508.3428]
            EPSG:4326: [-180.0, -85.0511287798, 180.0, 85.0511287798]
        queryable: False
        llbbox: [-180.0, -85.0511287798, 180.0, 85.0511287798]
```

11.6 grids

New in version 1.5.0.

This sub-command displays information about configured grids.

The command takes a MapProxy configuration file and returns all configured grids.

Furthermore, default values for each grid will be displayed if they are not defined explicitly. All default values are marked with an asterisk in the output.

-f <path/to/config>, **--mapproxy-config** <path/to/config>
: Display all configured grids for this MapProxy configuration with detailed information. If this option is not set, the sub-command will try to use the last argument as the mapproxy config.

-l, **--list**
: Display only the names of the grids for the given configuration, which are used by any grid.

--all
: Show also grids that are not referenced by any cache.

-g <grid_name>, **--grid** <grid_name>
: Display information only for a single grid. The tool will exit, if the grid name is not found.

-c <coverage name>, **--coverage** <coverage name>
: Display an approximation of the number of tiles for each level that which are within this coverage. The coverage must be defined in Seed configuration.

-s <seed.yaml>, **--seed-conf** <seed.yaml>
: This option loads the seed configuration and is needed if you use the `--coverage` option.

11.6.1 Example

With the following MapProxy grid configuration:

```
grids:
  localgrid:
    srs: EPSG:31467
    bbox: [5,50,10,55]
    bbox_srs: EPSG:4326
    min_res: 10000
  localgrid2:
    base: localgrid
    srs: EPSG:25832
    res_factor: sqrt2
    tile_size: [512, 512]
```

List all configured grids:

```
mapproxy-util grids --list --mapproxy-config /path/to/mapproxy.yaml

GLOBAL_GEODETIC
GLOBAL_MERCATOR
localgrid
localgrid2
```

Display detailed information for one specific grid:

```
mapproxy-util grids --grid localgrid --mapproxy-conf /path/to/mapproxy.yaml

localgrid:
    Configuration:
        bbox: [5, 50, 10, 55]
        bbox_srs: 'EPSG:4326'
        min_res: 10000
        origin*: 'sw'
        srs: 'EPSG:31467'
        tile_size*: [256, 256]
    Levels: Resolutions, # x * y = total tiles
        00:  10000,             #      1 * 1      =        1
        01:  5000.0,            #      1 * 1      =        1
        02:  2500.0,            #      1 * 1      =        1
        03:  1250.0,            #      2 * 2      =        4
        04:  625.0,             #      3 * 4      =       12
        05:  312.5,             #      5 * 8      =       40
        06:  156.25,            #      9 * 15     =      135
        07:  78.125,            #     18 * 29     =      522
        08:  39.0625,           #     36 * 57     =   2.052K
        09:  19.53125,          #     72 * 113    =   8.136K
        10:  9.765625,          #    144 * 226    =  32.544K
        11:  4.8828125,         #    287 * 451    = 129.437K
        12:  2.44140625,        #    574 * 902    = 517.748K
        13:  1.220703125,       #   1148 * 1804   =   2.071M
        14:  0.6103515625,      #   2295 * 3607   =   8.278M
        15:  0.30517578125,     #   4589 * 7213   =  33.100M
        16:  0.152587890625,    #   9178 * 14426  = 132.402M
        17:  0.0762939453125,   #  18355 * 28851  = 529.560M
        18:  0.03814697265625,  #  36709 * 57701  =   2.118G
        19:  0.019073486328125, #  73417 * 115402 =   8.472G
```

11.7 export

This sub-command exports tiles from one cache to another. This is similar to the seed tool, but you don't need to edit the configuration. The destination cache, grid and the coverage can be defined on the command line.

Required arguments:

-f, --mapproxy-conf
The path of the MapProxy configuration of the source cache.

--source
Name of the source or cache to export.

--levels
Comma separated list of levels to export. You can also define a range of levels. For example `'1,2,3,4,5','1..10'` or `'1,3,4,6..8'`.

--grid
The tile grid for the export. The option can either be the name of the grid as defined in the in the MapProxy configuration, or it can be the grid definition itself. You can define a grid as a single string of the key-value pairs. The grid definition *supports all grid parameters*. See below for examples.

--dest
Destination of the export. Can be a filename, directory or URL, depending on the export `--type`.

--type
Choose the export type. See below for a list of all options.

Other options:

--fetch-missing-tiles
If MapProxy should request missing tiles from the source. By default, the export tool will only existing tiles.

--coverage, --srs, --where
Limit the export to this coverage. You can use a BBOX, WKT files or OGR datasources. See *Coverages*.

-c `N`**, --concurrency** `N`
The number of concurrent export processes.

11.7.1 Export types

tms: Export tiles in a TMS like directory structure.

mapproxy or tc: Export tiles like the internal cache directory structure. This is compatible with TileCache.

mbtile: Export tiles into a MBTile file.

sqlite: Export tiles into SQLite level files.

geopackage: Export tiles into a GeoPackage file.

arcgis: Export tiles in a ArcGIS exploded cache directory structure.

compact-v1: Export tiles as ArcGIS compact cache bundle files (version 1).

11.7.2 Examples

Export tiles into a TMS directory structure under `./cache/`. Limit export to the BBOX and levels 0 to 6.

```
mapproxy-util export -f mapproxy.yaml --grid osm_grid \
    --source osm_cache --dest ./cache/ \
    --levels 1..6 --coverage 5,50,10,60 --srs 4326
```

Export tiles into an MBTiles file. Limit export to a shape coverage.

```
mapproxy-util export -f mapproxy.yaml --grid osm_grid \
    --source osm_cache --dest osm.mbtiles --type mbtile \
    --levels 1..6 --coverage boundaries.shp \
    --where 'CNTRY_NAME = "Germany"' --srs 3857
```

Export tiles into an MBTiles file using a custom grid definition.

```
mapproxy-util export -f mapproxy.yaml --levels 1..6 \
    --grid "srs='EPSG:4326' bbox=[5,50,10,60] tile_size=[512,512]" \
    --source osm_cache --dest osm.mbtiles --type mbtile \
```

11.8 defrag-compact-cache

The ArcGIS compact cache format version 1 and 2 are append only. Updating existing tiles will increase the file size. Bundle files become larger and fragmented with time. The `defrag-compact-cache` sub-command compacts existing bundle files by rewriting and reorganizing each bundle file.

Required arguments:

-f, --mapproxy-conf
The path of the MapProxy configuration with the configured compact caches.

Optional arguments:

--caches
Comma separated list of caches to defragment. By default all configured compact caches will be defragmented.

--min-percent, --min-mb
Bundle files with only a minmal fragmentation are skipped. You can define this threshold with `--min-percent` as the required minimal percentage of unused space and `--min-mb` as the minimal required unused space in megabytes. Both thresholds must be exceeded. Defaults to 10% and 1MB.

-n, --dry-run
This will simulate the defragmentation process.

11.8.1 Examples

Defragment bundle files from `map1_cache` and `map2_cache` when they have more than 20% and 5MB of unused space. E.g. a 20 MB bundle file only gets rewritten if it becomes smaller then 15MB after defragmentation; a 500MB bundle file only gets rewritten if it becomes smaller then 400MB after defragmentation.

```
mapproxy-util defrag-compact-cache -f mapproxy.yaml \
  --min-percent 20 \
  --min-mb 5 \
  --caches map1_cache,map2_cache
```

CHAPTER

TWELVE

MAPPROXY-UTIL AUTOCONFIG

The `autoconfig` sub-command of `mapproxy-util` creates MapProxy and MapProxy-seeding configurations based on existing WMS capabilities documents.

It creates a `source` for each available layer. The source will include a BBOX coverage from the layer extent, `legendurl` for legend graphics, `featureinfo` for querlyable layers, scale hints and all detected `supported_srs`. It will duplicate the layer tree to the `layers` section of the MapProxy configuration, including the name, title and abstract.

The tool will create a cache for each source layer and `supported_srs` _if_ there is a grid configured in your `--base` configuration for that SRS.

The MapProxy layers will use the caches when available, otherwise they will use the source directly (cascaded WMS).

Note: The tool can help you to create new configations, but it can't predict how you will use the MapProxy services. The generated configuration can be highly inefficient, especially when multiple layers with separate caches are requested at once. Please make sure you understand the configuration and check the documentation for more options that are useful for your use-cases.

12.1 Options

`--capabilities` `<url|filename>`
: URL or filename of the WMS capabilities document. The tool will add *REQUEST* and *SERVICE* parameters to the URL as necessary.

`--output` `<filename>`
: Filename for the created MapProxy configuration.

`--output-seed` `<filename>`
: Filename for the created MapProxy-seeding configuration.

`--force`
: Overwrite any existing configuration with the same output filename.

`--base` `<filename>`
: Base configuration that should be included in the `--output` file with the `base` option.

`--overwrite` `<filename>`

`--overwrite-seed` `<filename>`
: YAML configuration that overwrites configuration optoins before the generated configuration is written to `--output`/`--output-seed`.

12.1.1 Example

Print configuration on console:

```
mapproxy-util autoconfig \
    --capabilities http://osm.omniscale.net/proxy/service
```

Write MapProxy and MapProxy-seeding configuration to files:

```
mapproxy-util autoconfig \
    --capabilities http://osm.omniscale.net/proxy/service \
    --output mapproxy.yaml \
    --output-seed seed.yaml
```

Write MapProxy configuration with caches for grids from `base.yaml`:

```
mapproxy-util autoconfig \
    --capabilities http://osm.omniscale.net/proxy/service \
    --output mapproxy.yaml \
    --base base.yaml
```

12.2 Overwrites

It's likely that you need to tweak the created configuration – e.g. to define another coverage, disable featureinfo, etc. You can do this by editing the output file of course, or you can modify the output by defining all changes to an overwrite file. Overwrite files are applied everytime you call `mapproxy-util autoconfig`.

Overwrites are YAML files that will be merged with the created configuration file.

The overwrites are applied independently for each `services`, `sources`, `caches` and `layers` section. That means, for example, that you can modify the `supported_srs` of a source and the tool will use the updated SRS list to decide which caches will be configured for that source.

12.2.1 Example

Created configuration:

```
sources:
  mysource_wms:
    type: wms
    req:
        url: http://example.org
        layers: a
```

Overwrite file:

```
sources:
  mysource_wms:
    supported_srs: ['EPSG:4326'] # add new value for mysource_wms
    req:
        layers: a,b  # overwrite existing value
        custom_param: 42  #  new value
```

Actual configuration written to `--output`:

```
sources:
  mysource_wms:
    type: wms
    supported_srs: ['EPSG:4326']
    req:
        url: http://example.org
        layers: a,b
        custom_param: 42
```

12.2.2 Special keys

There are a few special keys that you can use in your overwrite file.

All

The value of the `__all__` key will be merged into all dictionaries. The following overwrite will add `sessionid` to the `req` options of all `sources`:

```
sources:
  __all__:
    req:
      sessionid: 123456789
```

Extend

The values of keys ending with `__extend__` will be added to existing lists.

To add another SRS for one source:

```
sources:
    my_wms:
      supported_srs__extend__: ['EPSG:31467']
```

Wildcard

The values of keys starting or ending with three underscores (`___`) will be merged with values where the key matches the suffix or prefix.

For example, to set `levels` for `osm_webmercator` and `aerial_webmercator` and to set `refresh_before` for `osm_webmercator` and `osm_utm32`:

```
seeds:
    ____webmercator:
        levels:
          from: 0
          to: 12

    osm____:
        refresh_before:
            days: 5
```

CHAPTER

THIRTEEN

DEPLOYMENT

MapProxy implements the Web Server Gateway Interface (WSGI) which is for Python what the Servlet API is for Java. There are different ways to deploy WSGI web applications.

MapProxy comes with a simple HTTP server that is easy to start and sufficient for local testing, see *Testing*. For production and load testing it is recommended to choose one of the *production setups*.

13.1 Testing

The `serve-develop` subcommand of `mapproxy-util` starts an HTTP server for local testing. It takes an existing MapProxy configuration file as an argument:

```
mapproxy-util serve-develop mapproxy.yaml
```

The server automatically reloads if the configuration or any code of MapProxy changes.

`--bind, -b`
: Set the socket MapProxy should listen. Defaults to `localhost:8080`. Accepts either a port number or `hostname:portnumber`.

`--debug`
: Start MapProxy in debug mode. If you have installed Werkzeug (recommended) or Paste, you will get an interactive traceback in the web browser on any unhandled exception (internal error).

Note: This server is sufficient for local testing of the configuration, but it is *not* stable for production or load testing.

The `serve-multiapp-develop` subcommand of `mapproxy-util` works similar to `serve-develop` but takes a directory of MapProxy configurations. See *MultiMapProxy*.

13.2 Production

There are two common ways to deploy MapProxy in production.

Embedded in HTTP server You can directly integrate MapProxy into your web server. Apache can integrate Python web services with the `mod_wsgi` extension for example.

Behind an HTTP server or proxy You can run MapProxy as a separate local HTTP server behind an existing web server (nginx, Apache, etc.) or an HTTP proxy (Varnish, squid, etc).

Both approaches require a configuration that maps your MapProxy configuration with the MapProxy application. You can write a small script file for that.

Running MapProxy as a FastCGI server behind HTTP server, a third option, is no longer advised for new setups since the FastCGI package (flup) is no longer maintained and the Python HTTP server improved significantly.

13.2.1 Server script

You need a script that makes the configured MapProxy available for the Python WSGI servers.

You can create a basic script with `mapproxy-util`:

```
mapproxy-util create -t wsgi-app -f mapproxy.yaml config.py
```

The script contains the following lines and makes the configured MapProxy available as `application`:

```
from mapproxy.wsgiapp import make_wsgi_app
application = make_wsgi_app('examples/minimal/etc/mapproxy.yaml')
```

This is sufficient for embedding MapProxy with `mod_wsgi` or for starting it with Python HTTP servers like `gunicorn` (see further below). You can extend this script to setup logging or to set environment variables.

You can enable MapProxy to automatically reload the configuration if it changes:

```
from mapproxy.wsgiapp import make_wsgi_app
application = make_wsgi_app('examples/minimal/etc/mapproxy.yaml', reloader=True)
```

13.3 Apache mod_wsgi

The Apache HTTP server can directly integrate Python application with the mod_wsgi extension. The benefit is that you don't have to start another server. Read mod_wsgi installation for detailed instructions.

`mod_wsgi` requires a server script that defines the configured WSGI function as `application`. See *above*.

You need to modify your Apache `httpd.conf` as follows:

```
# if not loaded elsewhere
LoadModule wsgi_module modules/mod_wsgi.so

WSGIScriptAlias /mapproxy /path/to/mapproxy/config.py

<Directory /path/to/mapproxy/>
  Order deny,allow
  Allow from all
</Directory>
```

`mod_wsgi` has a lot of options for more fine tuning. `WSGIPythonHome` or `WSGIPythonPath` lets you configure your `virtualenv` and `WSGIDaemonProcess`/`WSGIProcessGroup` allows you to start multiple processes. See the mod_wsgi configuration directives documentation. Using Mapnik also requires the `WSGIApplicationGroup` option.

Note: On Windows only the `WSGIPythonPath` option is supported. Linux/Unix supports `WSGIPythonPath` and `WSGIPythonHome`. See also the mod_wsgi documentation for virtualenv for detailed information when using multiple virtualenvs.

A more complete configuration might look like:

```
# if not loaded elsewhere
LoadModule wsgi_module modules/mod_wsgi.so

WSGIScriptAlias /mapproxy /path/to/mapproxy/config.py
WSGIDaemonProcess mapproxy user=mapproxy group=mapproxy processes=8 threads=25
WSGIProcessGroup mapproxy
# WSGIPythonHome should contain the bin and lib dir of your virtualenv
WSGIPythonHome /path/to/mapproxy/venv
WSGIApplicationGroup %{GLOBAL}

<Directory /path/to/mapproxy/>
```

```
  Order deny,allow
  # For Apache 2.4:
  Require all granted
  # For Apache 2.2:
  # Allow from all
</Directory>
```

13.4 Behind HTTP server or proxy

There are Python HTTP servers available that can directly run MapProxy. Most of them are robust and efficient, but there are some odd HTTP clients out there that (mis)interpret the HTTP standard in various ways. It is therefor recommended to put a HTTP server or proxy in front that is mature and widely deployed (like Apache, Nginx, etc.).

13.4.1 Python HTTP Server

You need start these servers in the background on start up. It is recommended to create an init script for that or to use tools like upstart or supervisord.

Gunicorn

Gunicorn is a Python WSGI HTTP server for UNIX. Gunicorn use multiple processes but the process number is fixed. The default worker is synchronous, meaning that a process is blocked while it requests data from another server for example. You need to choose an asynchronous worker like eventlet.

You need a server script that creates the MapProxy application (see *above*). The script needs to be in the directory from where you start `gunicorn` and it needs to end with `.py`.

To start MapProxy with the Gunicorn web server with four processes, the eventlet worker and our server script (without `.py`):

```
cd /path/of/config.py/
gunicorn -k eventlet -w 4 -b :8080 config:application --no-sendfile
```

An example upstart script (`/etc/init/mapproxy.conf`) might look like:

```
start on runlevel [2345]
stop on runlevel [!2345]

respawn

setuid mapproxy
setgid mapproxy

chdir /etc/opt/mapproxy

exec /opt/mapproxy/bin/gunicorn -k eventlet -w 8 -b :8080 \
    --no-sendfile \
    application \
    >>/var/log/mapproxy/gunicorn.log 2>&1
```

Spawning

Spawning is another Python WSGI HTTP server for UNIX that supports multiple processes and multiple threads.

```
cd /path/of/config.py/
spawning config.application --threads=8 --processes=4 \
  --port=8080
```

13.4.2 HTTP Proxy

You can either use a dedicated HTTP proxy like Varnish or a general HTTP web server with proxy capabilities like Apache with mod_proxy in front of MapProxy.

You need to set some HTTP headers so that MapProxy can generate capability documents with the URL of the proxy, instead of the local URL of the MapProxy application.

- `Host` – is the hostname that clients use to acces MapProxy (i.e. the proxy)
- `X-Script-Name` – path of MapProxy when the URL is not `/` (e.g. `/mapproxy`)
- `X-Forwarded-Host` – alternative to `HOST`
- `X-Forwarded-Proto` – should be `https` when the client connects with HTTPS

Nginx

Here is an example for the Nginx webserver with the included proxy module. It forwards all requests to `example.org/mapproxy` to `localhost:8181/`:

```
server {
  server_name example.org;
  location /mapproxy {
    proxy_pass http://localhost:8181;
    proxy_set_header Host $http_host;
    proxy_set_header X-Script-Name /mapproxy;
  }
}
```

Apache

Here is an example for the Apache webserver with the included `mod_proxy` and `mod_headers` modules. It forwards all requests to `example.org/mapproxy` to `localhost:8181/`

```
<IfModule mod_proxy.c>
  <IfModule mod_headers.c>
        <Location /mapproxy>
                ProxyPass http://localhost:8181
                ProxyPassReverse  http://localhost:8181
                RequestHeader add X-Script-Name "/mapproxy"
        </Location>
  </IfModule>
</IfModule>
```

You need to make sure that both modules are loaded. The `Host` is already set to the right value by default.

13.5 Other deployment options

Refer to http://wsgi.readthedocs.org/en/latest/servers.html for a list of some available WSGI servers.

13.5.1 FastCGI

Note: Running MapProxy as a FastCGI server behind HTTP server is no longer advised for new setups since the used Python package (flup) is no longer maintained. Please refer to the MapProxy 1.5.0 deployment documentation for more information on FastCGI.

13.6 Performance

Because of the way Python handles threads in computing heavy applications (like MapProxy WMS is), you should choose a server that uses multiple processes (pre-forking based) for best performance.

The examples above are all minimal and you should read the documentation of your components to get the best performance with your setup.

13.7 Load Balancing and High Availablity

You can easily run multiple MapProxy instances in parallel and use a load balancer to distribute requests across all instances, but there are a few things to consider when the instances share the same tile cache with NFS or other network filesystems.

MapProxy uses file locks to prevent that multiple processes will request the same image twice from a source. This would typically happen when two or more requests for missing tiles are processed in parallel by MapProxy and these tiles belong to the same meta tile. Without locking MapProxy would request the meta tile for each request. With locking, only the first process will get the lock and request the meta tile. The other processes will wait till the the first process releases the lock and will then use the new created tile.

Since file locking doesn't work well on most network filesystems you are likely to get errors when MapProxy writes these files on network filesystems. You should configure MapProxy to write all lock files on a local filesystem to prevent this. See *globals.cache.lock_dir* and *globals.cache.tile_lock_dir*.

With this setup the locking will only be effective when parallel requests for tiles of the same meta tile go to the same MapProxy instance. Since these requests are typically made from the same client you should enable *sticky sessions* in you load balancer when you offer tiled services (WMTS/TMS/KML).

13.8 Logging

MapProxy uses the Python logging library for the reporting of runtime information, errors and warnings. You can configure the logging with Python code or with an ini-style configuration. Read the logging documentation for more information.

13.8.1 Loggers

MapProxy uses multiple loggers for different parts of the system. The loggers build a hierarchy and are named in dotted-notation. `mapproxy` is the logger for everything, `mapproxy.source` is the logger for all sources, `mapproxy.source.wms` is the logger for all WMS sources, etc. If you configure on logger (e.g. `mapproxy`) then all sub-loggers will also use this configuration.

Here are the most important loggers:

`mapproxy.system` Logs information about the system and the installation (e.g. used projection library).

`mapproxy.config` Logs information about the configuration.

`mapproxy.source.XXX` Logs errors and warnings for service `XXX`.

mapproxy.source.request Logs all requests to sources with URL, size in kB and duration in milliseconds. The duration is the time it took to receive the header of the response. The actual request duration might be longer, especially for larger images or when the network bandwith is limited.

13.8.2 Enabling logging

The *test server* is already configured to log all messages to the console (`stdout`). The other deployment options require a logging configuration.

Server Script

You can use the Python logging API or load an `.ini` configuration if you have a *server script* for deployment.

The example script created with `mapproxy-util create -t wsgi-app` already contains code to load an `.ini` file. You just need to uncomment these lines and create a `log.ini` file. You can create an example `log.ini` with:

```
mapproxy-util create -t log-ini log.ini
```

13.9 MultiMapProxy

New in version 1.2.0.

You can run multiple MapProxy instances (configurations) within one process with the MultiMapProxy application.

MultiMapProxy can dynamically load configurations. You can put all configurations into one directory and MapProxy maps each file to a URL: `conf/proj1.yaml` is available at `http://hostname/proj1/`.

Each configuration will be loaded on demand and MapProxy caches each loaded instance. The configuration will be reloaded if the file changes.

MultiMapProxy as the following options:

config_dir The directory where MapProxy should look for configurations.

allow_listing If set to `true`, MapProxy will list all available configurations at the root URL of your MapProxy. Defaults to `false`.

13.9.1 Server Script

There is a `make_wsgi_app` function in the `mapproxy.multiapp` package that creates configured MultiMapProxy WSGI application. Replace the `application` definition in your script as follows:

```
from mapproxy.multiapp import make_wsgi_app
application = make_wsgi_app('/path/to.projects', allow_listing=True)
```

CHAPTER

FOURTEEN

CONFIGURATION EXAMPLES

This document will show you some usage scenarios of MapProxy and will explain some combinations of configuration options that might be useful for you.

14.1 Merge multiple layers

You have two WMS and want to offer a single layer with data from both servers. Each MapProxy cache can have more than one data source. MapProxy will combine the results from the sources before it stores the tiles on disk. These combined layers can also be requested via tiled services.

The sources should be defined from bottom to top. All sources except the bottom source needs to be transparent.

Example:

```
layers:
  - name: combined_layer
    title: Aerial image + roads overlay
    sources: [combined_cache]

caches:
  combined_cache:
    sources: [base, aerial]

sources:
  base:
    type: wms
    wms_opts:
      featureinfo: True
      version: 1.1.1
    req:
        url: http://one.example.org/mapserv/?map=/home/map/roads.map
        layers: roads
        transparent: true
  aerial:
    type: wms
    req:
        url: http://two.example.org/service?
        layers: aerial
```

Note: If the layers come from the same WMS server, then you can add them direct to the `layers` parameter. E.g. `layers: water,railroads,roads`.

14.1.1 Merge tile sources

You can also merge multiple tile sources. You need to tell MapProxy that all overlay sources are transparent:

```
sources:
  tileoverlay:
    type: tile
    url: http://localhost:8080/tile?x=%(x)s&y=%(y)s&z=%(z)s&format=png
    transparent: true
```

14.2 Access local servers

By default MapProxy will request data in the same format it uses to cache the data, if you cache files in PNG MapProxy will request all images from the source WMS in PNG. This encoding is quite CPU intensive for your WMS server but reduces the amount of data than needs to be transfered between you WMS and MapProxy. You can use uncompressed TIFF as the request format, if both servers are on the same host or if they are connected with high bandwidth.

Example:

```
sources:
  fast_source:
    type: cache_wms
    req:
      url: http://localhost/mapserv/?map=/home/map/roads.map
      layers: roads
      format: image/tiff
      transparent: true
```

14.3 Create WMS from existing tile server

You can use MapProxy to create a WMS server with data from an existing tile server. That tile server could be a WMTS, TMS or any other tile service where you can access tiles by simple HTTP requests. You always need to configure a cache in MapProxy to get a WMS from a tile source, since the cache is the part that does the tile stitching and reprojection.

Here is a minimal example:

```
layers:
 - name: my_layer
   title: WMS layer from tiles
   sources: [mycache]

caches:
  mycache:
    grids: [GLOBAL_WEBMERCATOR]
    sources: [my_tile_source]

sources:
  my_tile_source:
    type: tile
    url: http://tileserver/%(tms_path)s.png
```

You need to modify the `url` template parameter to match the URLs of your server. You can use `x`, `y`, `z` variables in the template, but MapProxy also supports the `quadkey` variable for Bing compatible tile service and `bbox` for WMS-C services. See the *tile source documentation* for all possible template values.

Here is an example of a WMTS source:

```
sources:
  my_tile_source:
    type: tile
    url: http://tileserver/wmts?SERVICE=WMTS&REQUEST=GetTile&
```

```
VERSION=1.0.0&LAYER=layername&TILEMATRIXSET=WEBMERCATOR&
TILEMATRIX=%(z)s&TILEROW=%(y)s&TILECOL=%(x)s&FORMAT=image%%2Fpng
```

Note: You need to escape percent signs (`%`) in the URL by repeating them (`%%`).

You can use the `GLOBAL_WEBMERCATOR` grid for OpenStreetMap or Google Maps compatible sources. Most TMS services should be compatible with the `GLOBAL_MERCATOR` definition that is similar to `GLOBAL_WEBMERCATOR` but uses a different origin (south west (TMS) instead of north west (OSM/WMTS/Google Maps/etc.)). Other tile services might use different SRS, bounding boxes or resolutions. You need to check the capabilities of your service and *configure a compatible grid*.

You also need to create your own grid when you want to change the name of it, which will appear in the WMTS or TMS URL.

Example configuration for an OpenStreetMap tile service:

```
layers:
  - name: my_layer
    title: WMS layer from tiles
    sources: [mycache]

caches:
  mycache:
    grids: [webmercator]
    sources: [my_tile_source]

sources:
  my_tile_source:
    type: tile
    grid: GLOBAL_WEBMERCATOR
    url: http://a.tile.openstreetmap.org/%(z)s/%(x)s/%(y)s.png

grids:
 webmercator:
   base: GLOBAL_WEBMERCATOR
```

Note: Please make sure you are allowed to access the tile service. Commercial tile provider often prohibit the direct access to tiles. The tile service from OpenStreetMap has a strict Tile Usage Prolicy.

14.4 Overlay tiles with OpenStreetMap or Google Maps in OpenLayers

You need to take care of a few options when you want to overlay your MapProxy tiles in OpenLayers with existing OpenStreetMap or Google Maps tiles.

The basic configuration for this use-case with MapProxy may look like this:

```
layers:
  - name: street_layer
    title: TMS layer with street data
    sources: [street_cache]

caches:
  street_cache:
    sources: [street_tile_source]

sources:
  street_tile_source:
    type: tile
```

```
    url: http://osm.omniscale.net/proxy/tiles/ \
      1.0.0/osm_roads_EPSG900913/%(z)s/%(x)s/%(y)s.png
    transparent: true
```

All you need to do now is to configure your OpenLayers client. The first example creates a simple OpenLayers map in webmercator projection, adds an OSM base layer and a TMS overlay layer with our MapProxy tile service.:

```
<script src="http://openlayers.org/api/OpenLayers.js"></script>
<script type="text/javascript">
  var map;
  function init(){
      map = new OpenLayers.Map('map', {
          projection: new OpenLayers.Projection("EPSG:900913")
      });

      var base_layer = new OpenLayers.Layer.OSM();

      var overlay_layer = new OpenLayers.Layer.TMS(
          'TMS street_layer',
          'http://127.0.0.1:8080/tiles/',
          {layername: 'street_layer_EPSG900913',
           type: 'png', isBaseLayer: false}
      );

      map.addLayer(base_layer);
      map.addLayer(overlay_layer);
      map.zoomToMaxExtent();
  };
</script>
```

Note that we used the `/tiles` service instead of `/tms` here. See *the tile service documentation* for more information.

Also remember that OpenStreetMap and Google Maps tiles have the origin in the upper left corner of the map, instead of the lower left corner as TMS does. Have a look at the *example configuration for OpenStreetMap tiles* for more information on that topic. The OpenLayers TMS and OSM layers already handle the difference.

You can change how MapProxy calculates the origin of the tile coordinates, if you want to use your MapProxy tile service with the OpenLayers OSM layer class or if you want to use a client that does not have a TMS layer.

The following example uses the class OpenLayers.Layer.OSM:

```
var overlay_layer = new OpenLayers.Layer.OSM("OSM osm_layer",
    "http://x.osm.omniscale.net/proxy/tiles/ \
    osm_roads_EPSG900913/${z}/${x}/${y}.png?origin=nw",
    {isBaseLayer: false, tileOptions: {crossOriginKeyword: null}}
);
```

The origin parameter at the end of the URL tells MapProxy that the client expects the origin in the upper left corner (north/west). You can change the default origin of all MapProxy tile layers by using the `origin` option of the `tms` service. See the *TMS standard tile origin* for more informations.

14.5 Using existing caches

New in version 1.5.0.

In some special use-cases you might want to use a cache as the source of another cache. For example, you might need to change the grid of an existing cache to cover a larger bounding box, or to support tile clients that expect a different grid, but you don't want to seed the data again.

Here is an example of a cache in UTM that uses data from an existing cache in web-mercator projection.

```
layers:
  - name: lyr1
    title: Layer using data from existing_cache
    sources: [new_cache]

caches:
  new_cache:
    grids: [utm32n]
    sources: [existing_cache]

  existing_cache:
    grids: [GLOBAL_WEBMERCATOR]
    sources: [my_source]

grids:
  utm32n:
    srs: 'EPSG:25832'
    bbox: [4, 46, 16, 56]
    bbox_srs: 'EPSG:4326'
    origin: 'nw'
    min_res: 5700
```

14.6 Reprojecting Tiles

New in version 1.5.0.

When you need to access tiles in a projection that is different from your source tile server, then you can use the *cache as cache source* feature from above. Here is an example that uses OSM tiles as a source and offers them in UTM projection. The *disable_storage* option prevents MapProxy from building up two caches. The *meta_size* makes MapProxy to reproject multiple tiles at once.

Here is an example that makes OSM tiles available as tiles in UTM. Note that reprojecting vector data results in quality loss. For better results you need to find similar resolutions between both grids.

```
layers:
  - name: osm
    title: OSM in UTM
    sources: [osm_cache]

caches:
  osm_cache:
    grids: [utm32n]
    meta_size: [4, 4]
    sources: [osm_cache_in]

  osm_cache_in:
    grids: [GLOBAL_WEBMERCATOR]
    disable_storage: true
    sources: [osm_source]

sources:
  osm_source:
    type: tile
    grid: GLOBAL_WEBMERCATOR
    url: http://a.tile.openstreetmap.org/%(z)s/%(x)s/%(y)s.png

grids:
  utm32n:
    srs: 'EPSG:25832'
    bbox: [4, 46, 16, 56]
    bbox_srs: 'EPSG:4326'
```

```
    origin: 'nw'
    min_res: 5700
```

14.7 Create grayscale images

New in version 1.9.0.

You can create a grayscale layer from an existing source by creating a cache that merges multiple bands into a single band. The band sources can come from caches, but also from any direct source. You can `disable_storage` to make this conversion on-the-fly. The following example mixes the RGB bands of a source with factors that matches the intensity perception of most humans:

```
caches:
 grayscale_cache:
     disable_storage: true
     sources:
         l: [
             {source: dop, band: 0, factor: 0.21},
             {source: dop, band: 1, factor: 0.72},
             {source: dop, band: 2, factor: 0.07},
         ]
```

14.8 Cache raster data

You have a WMS server that offers raster data like aerial images. By default MapProxy uses PNG images as the caching format. The encoding process for PNG files is very CPU intensive and thus the caching process itself takes longer. For aerial images the quality of loss-less image formats like PNG is often not required. For best performance you should use JPEG as the cache format.

By default MapProxy uses *bicubic* resampling. This resampling method also sharpens the image which is important for vector images. Aerial images do not need this, so you can use *bilinear* or even Nearest Neighbor (*nearest*) resampling.

```
caches:
  aerial_images_cache:
    format: image/jpeg
    image:
      resampling_method: nearest
    sources: [aerial_images]
```

You might also want to experiment with different compression levels of JPEG. A higher value of `jpeg_quality` results in better image quality at the cost of slower encoding and lager file sizes. See *mapproxy.yaml configuration*.

```
globals:
  jpeg_quality: 80
```

14.8.1 Mixed mode

You need to store images with transparency when you want to overlay them over other images, e.g. at the boundaries of your aerial image coverage. PNG supports transparency but it is not efficient with arial images, while JPEG is efficient for aerial images but doesn't support transparency.

MapProxy *has a mixed image format* for this case. With the `mixed` format, MapProxy stores tiles as either PNG or JPEG, depending on the transparency of each tile. Images with transparency will be stored as PNG, fully opaque images as JPEG.

Note: The source of your cache must support transparent images and you need to set the corresponding options.

```
caches:
  mixed_cache:
    format: mixed
    sources: [wms_source]
    request_format: image/png

sources:
  wms_source:
    type: wms
    req:
      url: http://localhost:42423/service
      layers: aerial
      transparent: true
```

You can now use the cache in all MapProxy services. WMS GetMap requests will return the image with the requested format. With TMS or WMTS you can only request PNG tiles, but the actual response image is either PNG or JPEG. The HTTP *content-type* header is set accordingly. This is supported by all web browsers.

14.9 Cache vector data

You have a WMS server that renders vector data like road maps.

14.9.1 Cache resolutions

By default MapProxy caches traditional power-of-two image pyramids, the resolutions between each pyramid level doubles. For example if the first level has a resolution of 10km, it would also cache resolutions of 5km, 2.5km, 1.125km etc. Requests with a resolution of 7km would be generated from cached data with a resolution of 10km. The problem with this approach is, that everything needs to be scaled down, lines will get thin and text labels will become unreadable. The solution is simple: Just add more levels to the pyramid. There are three options to do this.

You can set every cache resolution in the `res` option of a layer.

```
caches:
  custom_res_cache:
    grids: [custom_res]
    sources: [vector_source]

grids:
  custom_res_cache:
    srs: 'EPSG:31467'
    res: [10000, 7500, 5000, 3500, 2500]
```

You can specify a different factor that is used to calculate the resolutions. By default a factor of 2 is used (10, 5, 2.5,. . .) but you can set smaller values like 1.6 (10, 6.25, 3.9,. . .):

```
grids:
  custom_factor:
    res_factor: 1.6
```

The third options is a convenient variation of the previous option. A factor of 1.41421, the square root of two, would get resolutions of 10, 7.07, 5, 3.54, 2.5,. . . . Notice that every second resolution is identical to the power-of-two resolutions. This comes in handy if you use the layer not only in classic WMS clients but also want to use it in tile-based clients like OpenLayers, which only request in these resolutions.

```
grids:
  sqrt2:
    res_factor: sqrt2
```

Note: This does not improve the quality of aerial images or scanned maps, so you should avoid it for these images.

14.9.2 Resampling method

You can configure the method MapProxy uses for resampling when it scales or transforms data. For best results with vector data – from a viewers perspective – you should use bicubic resampling. You can configure this for each cache or in the globals section:

```
caches:
  vector_cache:
    image:
      resampling: bicubic
    # [...]

# or

globals:
  image:
    resampling: bicubic
```

14.10 WMS Sources with Styled Layer Description (SLD)

You can configure SLDs for your WMS sources.

```
sources:
  sld_example:
    type: wms
    req:
      url: http://example.org/service?
      sld: http://example.net/mysld.xml
```

MapProxy also supports local file URLs. MapProxy will use the content of the file as the `sld_body`. The path can either be absolute (e.g. `file:///path/to/sld.xml`) or relative (`file://path/to/sld.xml`) to the mapproxy.yaml file. The file should be UTF-8 encoded.

You can also configure the raw SLD with the `sld_body` option. You need to indent whole SLD string.

```
sources:
  sld_example:
    type: wms
    req:
      url: http://example.org/service?
      sld_body:
        <sld:StyledLayerDescriptor version="1.0.0"
        [snip]
        </sld:StyledLayerDescriptor>
```

MapProxy will use HTTP POST requests in this case. You can change `http.method`, if you want to force GET requests.

14.11 Add highly dynamic layers

You have dynamic layers that change constantly and you do not want to cache these. You can use a direct source. See next example.

14.12 Reproject WMS layers

If you do not want to cache data but still want to use MapProxy's ability to reproject WMS layers on the fly, you can use a direct layer. Add your source directly to your layer instead of a cache.

You should explicitly define the SRS the source WMS supports. Requests in other SRS will be reprojected. You should specify at least one geographic and one projected SRS to limit the distortions from reprojection.

```
layers:
  - name: direct_layer
    sources: [direct_wms]

sources:
  direct_wms:
    type: wms
    supported_srs: ['EPSG:4326', 'EPSG:25832']
    req:
      url: http://wms.example.org/service?
      layers: layer0,layer1
```

14.13 FeatureInformation

MapProxy can pass-through FeatureInformation requests to your WMS sources. You need to enable each source:

```
sources:
  fi_source:
    type: wms
    wms_opts:
      featureinfo: true
    req:
      url: http://example.org/service?
      layers: layer0
```

MapProxy will mark all layers that use this source as `queryable`. It also works for sources that are used with caching.

Note: The more advanced features *require the lxml library*.

14.13.1 Concatenation

Feature information from different sources are concatenated as plain text, that means that XML documents may become invalid. But MapProxy can also do content-aware concatenation when *lxml* is available.

HTML

Multiple HTML documents are put into the HTML `body` of the first document. MapProxy creates the HTML skeleton if it is missing.

```
<p>FI1</p>
```

and

```
<p>FI2</p>
```

will result in:

```
<html>
  <body>
    <p>FI1</p>
    <p>FI2</p>
 </body>
</html>
```

XML

Multiple XML documents are put in the root of the first document.

```
<root>
  <a>FI1</a>
</root>
```

and

```
<other_root>
  <b>FI2</b>
</other_root>
```

will result in:

```
<root>
  <a>FI1</a>
  <b>FI2</b>
</root>
```

14.13.2 XSL Transformations

MapProxy supports XSL transformations for more control over feature information. This also requires *lxml*. You can add an XSLT script for each WMS source (incoming) and for the WMS service (outgoing).

You can use XSLT for sources to convert all incoming documents to a single, uniform format and then use outgoing XSLT scripts to transform this format to either HTML or XML/GML output.

Example

Lets assume we have two WMS sources where we have no control over the format of the feature info responses.

One source only offers HTML feature information. The XSLT script extracts data from a table. We force the `INFO_FORMAT` to HTML, so that MapProxy will not query another format.

```
fi_source:
  type: wms
  wms_opts:
    featureinfo: true
    featureinfo_xslt: ./html_in.xslt
    featureinfo_format: text/html
  req: [...]
```

The second source supports XML feature information. The script converts the XML data to the same format as the HTML script. This service uses WMS 1.3.0 and the format is `text/xml`.

```
fi_source:
  type: wms
  wms_opts:
    version: 1.3.0
    featureinfo: true
    featureinfo_xslt: ./xml_in.xslt
```

```
    featureinfo_format: text/xml
  req: [...]
```

We then define two outgoing XSLT scripts that transform our intermediate format to the final result. We can define scripts for different formats. MapProxy chooses the right script depending on the WMS version and the `INFO_FORMAT` of the request.

```
wms:
  featureinfo_xslt:
    html: ./html_out.xslt
    xml: ./xml_out.xslt
  [...]
```

14.14 WMTS service with dimensions

New in version 1.6.0.

The dimension support in MapProxy is still limited, but you can use it to create a WMTS front-end for a multi-dimensional WMS service.

First you need to add the WMS source and configure all dimensions that MapProxy should forward to the service:

```
temperature_source:
  type: wms
  req:
    url: http://example.org/service?
    layers: temperature
  forward_req_params: ['time', 'elevation']
```

We need to create a cache since we want to access the source from a tiled service (WMTS). Actual caching is not possible at the moment, so it is necessary to disable it with `disable_storage: true`.

```
caches:
  temperature:
    grids: [GLOBAL_MERCATOR]
    sources: [temperature_source]
    disable_storage: true
    meta_size: [1, 1]
    meta_buffer: 0
```

Then we can add a layer with all available dimensions:

```
layers:
  - name: temperature
    title: Temperature
    sources: [temperature]
    dimensions:
      time:
        values:
          - "2012-11-12T00:00:00"
          - "2012-11-13T00:00:00"
          - "2012-11-14T00:00:00"
          - "2012-11-15T00:00:00"
      elevation:
        values:
          - 0
          - 1000
          - 3000
        default: 0
```

You can know access this layer with the elevation and time dimensions via the WMTS KVP service. The RESTful service requires a custom URL template that contains the dimensions. For example:

```
services:
  wmts:
    restful_template: '/{Layer}/{Time}/{Elevation}/{TileMatrixSet}
        /{TileMatrix}/{TileCol}/{TileRow}.{Format}'
```

Tiles are then available at `/wmts/temperature/GLOBAL_MERCATOR/1000/2012-11-12T00:00Z/6/33/22.png`.
You can use `default` for missing dimensions, e.g. `/wmts/map/GLOBAL_MERCATOR/default/default/6/33/22.png`

14.15 WMS layers with HTTP Authentication

You have a WMS source that requires authentication. MapProxy has support for HTTP Basic Authentication and HTTP Digest Authentication. You just need to add the username and password to the URL. Since the Basic and Digest authentication are not really secure, you should use this feature in combination with HTTPS. You need to configure the SSL certificates to allow MapProxy to verify the HTTPS connection. See *HTTPS configuration for more information*.

```
secure_source:
  type: wms
  req:
    url: https://username:mypassword@example.org/service?
    layers: securelayer
```

MapProxy removes the username and password before the URL gets logged or inserted into service exceptions.

You can disable the certificate verification if you you don't need it.

```
secure_source:
  type: wms
  http:
    ssl_no_cert_checks: True
  req:
    url: https://username:mypassword@example.org/service?
    layers: securelayer
```

14.16 Access sources through HTTP proxy

MapProxy can use an HTTP proxy to make requests to your sources, if your system does not allow direct access to the source. You need to set the `http_proxy` environment variable to the proxy URL. This also applies if you install MapProxy with `pip` or `easy_install`.

On Linux/Unix:

```
$ export http_proxy="http://example.com:3128"
$ mapproxy-util serve-develop mapproxy.yaml
```

On Windows:

```
c:\> set http_proxy="http://example.com:3128"
c:\> mapproxy-util serve-develop mapproxy.yaml
```

You can also set this in your *server script*:

```
import os
os.environ["http_proxy"] = "http://example.com:3128"
```

Add a username and password to the URL if your HTTP proxy requires authentication. For example `http://username:password@example.com:3128`.

You can use the `no_proxy` environment variable if you need to bypass the proxy for some hosts:

```
$ export no_proxy="localhost,127.0.0.1,196.168.1.99"
```

`no_proxy` is available since Python 2.6.3.

14.17 Serve multiple MapProxy instances

It is possible to load multiple MapProxy instances into a single process. Each MapProxy can have a different global configuration and different services and caches. [1] You can use *MultiMapProxy* to load multiple MapProxy configurations on-demand.

Example `config.py`:

```
from mapproxy.multiapp import make_wsgi_app
application = make_wsgi_app('/path/to/projects', allow_listing=True)
```

The MapProxy configuration from `/path/to/projects/app.yaml` is then available at `/app`.

You can reuse parts of the MapProxy configuration with the *base* option. You can put all common options into a single base configuration and reference that file in the actual configuration:

```
base: mapproxy.yaml
layers:
   [...]
```

14.18 Generate static quadkey / virtual earth cache for use on Multitouch table

Some software running on Microsoft multitouch tables need a static quadkey generated cache. Mapproxy understands quadkey both as a client and as a cache option.

Example part of `mapproxy.yaml` to generate a quadkey cache:

```
caches:
  osm_cache:
    grids: [GLOBAL_WEBMERCATOR]
    sources: [osm_wms]
    cache:
      type: file
      directory_layout: quadkey
```

14.19 HQ/Retina tiles

MapProxy has no native support for delivering high-resolution tiles, but you can create a second tile layer with HQ tiles, if your source supports rendering with different scale-factor or DPI.

At first you need two grids. One regular grid and one with half the resolution but twice the tile size. The following example configures two webmercator compatible grids:

```
grids:
  webmercator:
    srs: "EPSG:3857"
    origin: nw
    min_res: 156543.03392804097
  webmercator_hq:
    srs: "EPSG:3857"
```

[1] This does not apply to *srs.proj_data_dir*, because it affects the proj4 library directly.

```
    origin: nw
    min_res: 78271.51696402048
    tile_size: [512, 512]
```

Then you need two layers and two caches:

```
layers:
  - name: map
    title: Regular map
    sources: [map_cache]
  - name: map_hq
    title: HQ map
    sources: [map_hq_cache]

caches:
  map_cache:
    grids: [webmercator]
    sources: [map_source]
  map_hq_cache:
    grids: [webmercator_hq]
    sources: [map_hq_source]
```

And finally two sources. The source for the HQ tiles needs to render images with a higher scale/DPI setting. The `mapnik` source supports this with the `scale_factor` option. MapServer for example supports a `map_resolution` request parameter.

```
sources:
  map_source:
    type: mapnik
    mapfile: ./mapnik.xml
    transparent: true

  map_hq_source:
    type: mapnik
    mapfile: ./mapnik.xml
    transparent: true
    scale_factor: 2
```

With that configuration `/wmts/mapnik/webmercator/0/0/0.png` returns a regular webmercator tile:

`/wmts/mapnik_hq/webmercator_hq/0/0/0.png` returns the same tile with 512x512 pixel:

14.20 Serve multiple caches for a single layer

New in version 1.8.2.

You have a data set that you need to serve with different grids (i.e. WMTS tile matrix sets).

You can create a cache with multiple grids and use this as a layers source:

```
layers:
  - name: map
    title: Layer with multiple grids
    sources: [cache]

caches:
  cache:
    grids: [full_grid, sub_grid]
    sources: [source]
```

This *map* layer is available in WMS and in tile services. The grids are available as separate tile matrix sets in the

WMTS. However, this is limited to a single cache for each layer. You can't reuse the tiles from the *full_grid* for the *sub_grid*.

You need to use `tile_sources` to make multiple caches available as a single layer. `tile_sources` allows you to override `sources` for tile services. This allows you to use caches that build up on other caches.

For example:

```
layers:
  - name: map
    title: Layer with sources for tile services and for WMS
    tile_sources: [full_cache, inspire_cache]
    sources: [full_cache]

caches:
  full_cache:
    grids: [full_grid]
    sources: [source]
  inspire_cache:
    grids: [sub_grid]
    sources: [full_cache]
    disable_storage: true
```

CHAPTER

FIFTEEN

INSPIRE VIEW SERVICE

MapProxy can act as an INSPIRE View Service. A View Service is a WMS 1.3.0 with an extended capabilities document.

New in version 1.8.1.

15.1 INSPIRE Metadata

A View Service can either link to an existing metadata document or it can embed the service and layer metadata. These two options are described as Scenario 1 and 2 in the Technical Guidance document.

15.1.1 Linked Metadata

Scenario 1 uses links to existing INSPIRE Discovery Services (CSW). You can link to metadata documents for the service and each layer.

For services you need to use the `inspire_md` block inside `services.wms` with `type: linked`. For example:

```
services:
  wms:
    md:
      title: Example INSPIRE View Service
    inspire_md:
      type: linked
      metadata_url:
        media_type: application/vnd.iso.19139+xml
        url: http://example.org/csw/doc
      languages:
        default: eng
```

The View Services specification uses the WMS 1.3.0 extended capabilities for the layers metadata. Refer to the *layers metadata documentation*.

For example:

```
layers:
  - name: example_layer
    title: Example Layer
    md:
      metadata:
       - url:    http://example.org/csw/layerdoc
         type:   ISO19115:2003
         format: text/xml
```

15.1.2 Embedded Metadata

Scenario 2 embeds the metadata directly into the capabilities document. Some metadata elements are mapped to an equivalent element in the WMS capabilities. The Resource Title is set with the normal *title* option for example. Other elements need to be configured inside the `inspire_md` block with `type: embedded`.

Here is a full example:

```
services:
  wms:
    md:
      title: Example INSPIRE View Service
      abstract: This is an example service with embedded INSPIRE metadata.
      online_resource: http://example.org/
      contact:
        person: Your Name Here
        position: Technical Director
        organization: Acme Inc.
        address: Fakestreet 123
        city: Somewhere
        postcode: 12345
        country: Germany
        phone: +49(0)000-000000-0
        fax: +49(0)000-000000-0
        email: info@example.org
      access_constraints: constraints
      fees: 'None'
      keyword_list:
        - vocabulary: GEMET
          keywords:   [Orthoimagery]

    inspire_md:
      type: embedded
      resource_locators:
        - url: http://example.org/metadata
          media_type: application/vnd.iso.19139+xml
      temporal_reference:
        date_of_creation: 2015-05-01
      metadata_points_of_contact:
        - organisation_name: Acme Inc.
          email: acme@example.org
      conformities:
        - title:
            COMMISSION REGULATION (EU) No 1089/2010 of 23 November 2010 implementing Directive 20
          date_of_publication: 2010-12-08
          uris:
            - OJ:L:2010:323:0011:0102:EN:PDF
          resource_locators:
          - url: http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=OJ:L:2010:323:0011:0102:EN
            media_type: application/pdf
          degree: notEvaluated
      mandatory_keywords:
        - infoMapAccessService
        - humanGeographicViewer
      keywords:
        - title: GEMET - INSPIRE themes
          date_of_last_revision: 2008-06-01
          keyword_value: Orthoimagery
      metadata_date: 2015-07-23
      metadata_url:
        media_type: application/vnd.iso.19139+xml
        url: http://example.org/csw/doc
```

You can express all dates as either `date_of_creation`, `date_of_publication` or

`date_of_last_revision`.

The View Services specification uses the WMS 1.3.0 extended capabilities for the layers metadata. Refer to the *layers metadata documentation* for all available options.

For example:

```
layers:
  - name: example_layer
    title: Example Layer
    legendurl: http://example.org/example_legend.png
    md:
      abstract: Some abstract
      keyword_list:
        - vocabulary: GEMET
          keywords:   [Orthoimagery]
      metadata:
       - url:    http://example.org/csw/layerdoc
         type:   ISO19115:2003
         format: text/xml
      identifier:
       - url:    http://www.example.org
         name:   example.org
         value:  "http://www.example.org#cf3c8572-601f-4f47-a922-6c67d388d220"
```

15.2 Languages

A View Service always needs to indicate the language of the layer names, abstracts, map labels, etc.. You can only configure a single language as MapProxy does not support multi-lingual configurations. You need to set the default language as a ISO 639-2/alpha-3 code:

```
inspire_md:
  languages:
    default: eng
  ....
```

CHAPTER

SIXTEEN

WMS LABELING

The tiling of rendered vector maps often results in issues with truncated or repeated labels. Some of these issues can be reduced with a proper configuration of MapProxy, but some require changes to the configuration of the source WMS server.

This document describes settings for MapProxy and MapServer, but the problems and solutions are also valid for other WMS servers. Refer to their documentations on how to configure these settings.

16.1 The Problem

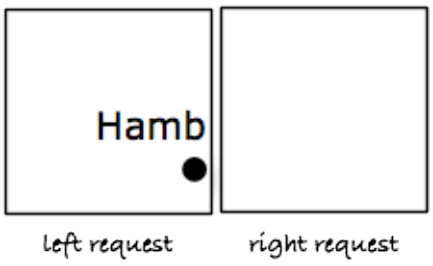
MapProxy always uses small tiles for caching. MapProxy does not pass through incoming requests to the source WMS [1], but it always requests images/tiles that are aligned to the internal grid. MapProxy combines, scales and reprojects these tiles for WMS requests and for tiled requests (TMS/KML) the tiles are combined by the client (OpenLayers, etc).

When tiles are combined, the text labels at the boundaries need to be present at both tiles and need to be placed at the exact same (geographic) location.

There are three common problems here.

16.1.1 No placement outside the BBOX

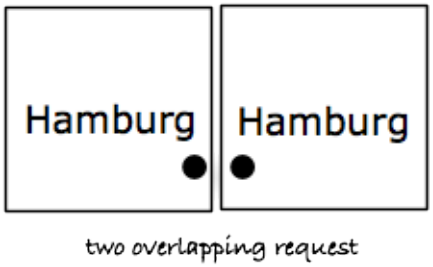
WMS servers do not draw features that are outside of the map bounds. For example, a city label that extends into the neighboring map tile will not be drawn in that other tile, because the geographic feature of the city (a single point) is only present in one tile.



16.1.2 Dynamic label placement

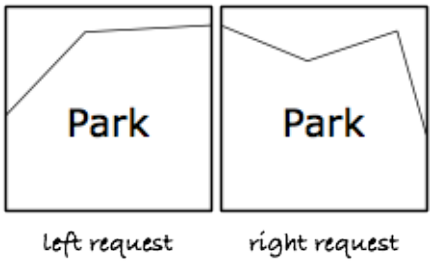
WMS servers can adjust the position of labels so that more labels can fit on a map. For example, a city label is not always displayed at the same geographic location, but moved around to fit in the requested map or to make space for other labels.

[1] Except for uncached, cascaded WMS requests.



16.1.3 Repeated labels

WMS servers render labels for polygon areas in each request. Labels for large areas will apear multiple times, once in each tile.



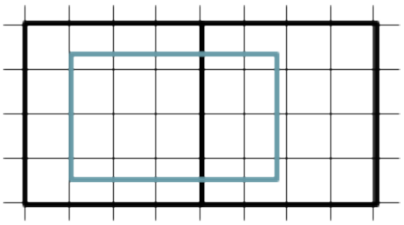
16.2 MapProxy Options

There are two options that help with these issues.

16.2.1 Meta Tiles

You can use meta tiles to reduce the labeling issues. A meta tile is a collection of multiple tiles. Instead of requesting each tile with a single request, MapProxy requests a single image that covers the area of multiple tiles and then splits that response into the actual tiles.

The following image demonstrates that:



The thin lines represent the tiles. The WMS request (inner box) consists of 20 tiles and without metatiling each tile results in a request to the WMS source. With a meta tile size of 4x4, only two larger requests to the source WMS are required (thick black box).

Because you are requesting less images, you have less boundaries where labeling issues can appear. In this case it reduces the number of tile/image boundaries from 31 to only one.

But, it only reduces the problem and does not solve it. Nonetheless, it should be used because it also reduces the load on the source WMS server.

You can configure the meta tile size in the `globals.cache` section and for each `cache`. It defaults to `[4, 4]`.

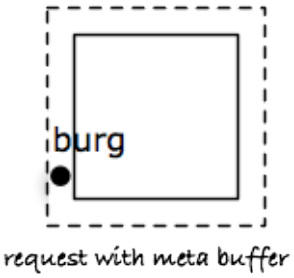
```
globals:
  cache:
    meta_size: [6, 6]

caches:
  mycache:
    sources: [...]
    grids: [...]
    meta_size: [8, 8]
```

This does also work for tiles services. When a client like OpenLayers requests the 20 tiles from the example above in parallel, MapProxy will still requests the two meta tiles. Locking ensures that each meta tile will be requested only once.

16.2.2 Meta Buffer

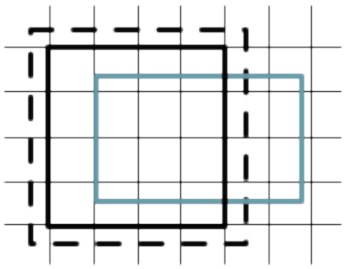
In addition to meta tiles, MapProxy implements a meta buffer. The meta buffer adds extra space at the edges of the requested area. With this buffer, you can solve the first issue: no placement outside the BBOX.



request with meta buffer

You can combine meta tiling and meta buffer. MapProxy then extends the whole meta tile with the configured buffer.

A meta buffer of 100 will add 100 pixels at each edge of the request. With a meta size of 4x4 and a tile size of 256x256, the requested image is extended from 1024x1024 to 1224x1224. The BBOX is also extended to match the new geographical extent.



To solve the first issue, the value should be at least half of your longest labels: If you have text labels that are up to 200 pixels wide, than you should use a meta buffer of around 120 pixels.

You can configure the size of the meta buffer in the `globals.cache` section and for each `cache`. It defaults to `80`.

```
globals:
  cache:
    meta_buffer: 100

caches:
  mycache:
    sources: [...]
    grids: [...]
    meta_buffer: 150
```

16.3 WMS Server Options

You can reduce some of the labeling issues with meta tiling, and solve the first issue with the meta buffer. The issues with dynamic and repeated labeling requires some changes to your WMS server.

In general, you need to disable the dynamic position of labels and you need to allow the rendering of partial labels.

16.4 MapServer Options

MapServer has lots of settings that affect the rendering. The two most important settings are

`PROCESSING "LABEL_NO_CLIP=ON"` from the `LAYER` configuration. With this option the labels are fixed to the whole feature and not only the part of the feature that is visible in the current map request. Default is off.

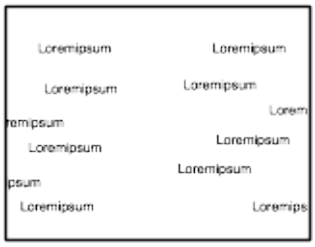
and

`PARTIALS` from the `LABEL` configuration. If this option is true, then labels are rendered beyond the boundaries of the map request. Default is true.

16.4.1 `PARTIAL FALSE`

The easiest option to solve all issues is `PARTIAL FALSE` with a meta buffer of 0. This prevents any label from truncation, but it comes with a large downside: Since no labels are rendered at the boundaries of the meta tiles, you will have areas with no labels at all. These areas form a noticeable grid pattern on your maps.

The following images demonstrates a WMS request with a meta tile boundary in the center.



You can improve that with the right set of configuration options for each type of geometry.

16.4.2 Points

As described above, you can use a meta buffer to prevent missing labels. You need to set `PARTIALS TRUE` (which is the default), and configure a large enough meta buffer. The labels need to be placed at the same position with each request. You can configure that with the `POSITION` options. The default is `auto` and you should set this to an explicit value, `cc` or `uc` for example.

`example.map`:

```
LABEL
  [...]
  POSITION cc
  PARTIALS TRUE
END
```

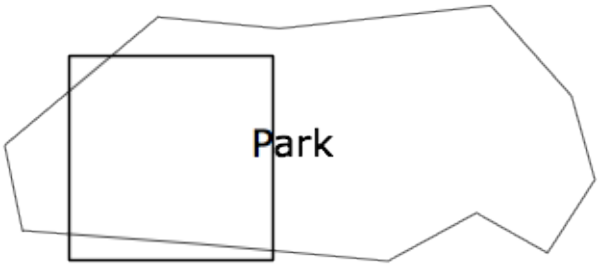
`mapproxy.yaml`:

```
caches:
  mycache:
    meta_buffer: 150
    [...]
```

16.4.3 Polygons

Meta tiling reduces the number of repeated labels, but they can still apear at the border of meta tiles.

You can use the `PROCESSING "LABEL_NO_CLIP=ON"` option to fix this problem. With this option, MapServer places the label always at a fixed position, even if that position is outside the current map request.



If the `LABEL_NO_CLIP` option is used, `PARTIALS` should be `TRUE`. Otherwise label would not be rendered if they overlap the map boundary. This options also requires a meta buffer.

`example.map`:

```
LAYER
  TYPE POLYGON
  PROCESSING "LABEL_NO_CLIP=ON"
  [...]
  LABEL
    [...]
    POSITION cc
    PARTIALS TRUE
  END
END
```

`mapproxy.yaml`:

```
caches:
  mycache:
    meta_buffer: 150
    [...]
```

16.4.4 Lines

By default, labels are repeated on longer line strings. Where these labels are repeated depends on the current view of that line. That placement might differ in two neighboring image requests for long lines.

Most of the time, the labels will match at the boundaries of the meta tiles, when you use `PARTIALS TRUE` and a meta buffer. But, you might notice truncated labels on long line strings. In practice these issues are rare, though.

`example.map`:

```
LAYER
  TYPE LINE
  [...]
  LABEL
    [...]
    PARTIALS TRUE
  END
END
```

`mapproxy.yaml`:

```
caches:
  mycache:
    meta_buffer: 150
    [...]
```

You can disable repeated labels with `PROCESSING LABEL_NO_CLIP="ON"`, if don't want to have any truncated labels. Like with polygons, you need set `PARTIALS TRUE` and use a meta buffer. The downside of this is that each lines will only have one label in the center of that line.

`example.map`:

```
LAYER
  TYPE LINE
  PROCESSING "LABEL_NO_CLIP=ON"
  [...]
  LABEL
    [...]
    PARTIALS TRUE
  END
END
```

`mapproxy.yaml`:

```
caches:
  mycache:
    meta_buffer: 150
    [...]
```

There is a third option. If you want repeated labels but don't want any truncated labels, you can set `PARTIALS FALSE`. Remember that you will get the same grid pattern as mentioned above, but it might not be noted if you mix this layer with other point and polygon layers where `PARTIALS` is enabled.

You need to compensate the meta buffer when you use `PARTIALS FALSE` in combination with other layers that require a meta buffer. You need to set the option `LABELCACHE_MAP_EDGE_BUFFER` to the negative value of your meta buffer.

```
WEB
  [...]
  METADATA
    LABELCACHE_MAP_EDGE_BUFFER "-100"
  END
END

LAYER
  TYPE LINE
  [...]
  LABEL
    [...]
    PARTIALS FALSE
  END
END
```

`mapproxy.yaml`:

```
caches:
  mycache:
    meta_buffer: 100
    [...]
```

16.5 Other WMS Servers

The most important step for all WMS servers is to disable to dynamic placement of labels. Look into the documentation how to do this for you WMS server.

If you want to contribute to this document then join our mailing list or use our issue tracker.

CHAPTER

SEVENTEEN

AUTHENTICATION AND AUTHORIZATION

Authentication is the process of mapping a request to a user. There are different ways to do this, from simple HTTP Basic Authentication to cookies or token based systems.

Authorization is the process that defines what an authenticated user is allowed to do. A datastore is required to store this authorization information for everything but trivial systems. These datastores can range from really simple text files (all users in this text file are allowed to do everything) to complex schemas with relational databases (user A is allowed to do B but not C, etc.).

As you can see, the options to choose when implementing a system for authentication and authorization are diverse. Developers (of SDIs, not the software itself) often have specific constraints, like existing user data in a database or an existing login page on a website for a Web-GIS. So it is hard to offer a one-size-fits-all solution.

Therefore, MapProxy does not come with any embedded authentication or authorization. But it comes with a flexible authorization interface that allows you (the SDI developer) to implement custom tailored systems.

Luckily, there are lots of existing toolkits that can be used to build systems that match your requirements. For authentication there is the repoze.who package with plugins for HTTP Basic Authentication, HTTP cookies, etc. For authorization there is the repoze.what package with plugins for SQL datastores, etc.

Note: Developing custom authentication and authorization system requires a bit Python programming and knowledge of WSGI and WSGI middleware.

17.1 Authentication/Authorization Middleware

Your auth system should be implemented as a WSGI middleware. The middleware sits between your web server and the MapProxy.

17.1.1 WSGI Filter Middleware

A simple middleware that authorizes random requests might look like:

```
class RandomAuthFilter(object):
    def __init__(self, app, global_conf):
        self.app = app

    def __call__(self, environ, start_response):
        if random.randint(0, 1) == 1:
          return self.app(environ, start_response)
        else:
          start_response('403 Forbidden',
            [('content-type', 'text/plain')])
          return ['no luck today']
```

You need to wrap the MapProxy application with your custom auth middleware. For deployment scripts it might look like:

```
application = make_wsgi_app('./mapproxy.yaml')
application = RandomAuthFilter(application)
```

For PasteDeploy you can use the `filter-with` option. The `config.ini` looks like:

```
[app:mapproxy]
use = egg:MapProxy#app
mapproxy_conf = %(here)s/mapproxy.yaml
filter-with = auth

[filter:auth]
paste.filter_app_factory = myauthmodule:RandomAuthFilter

[server:main]
...
```

You can implement simple authentication systems with that method, but you should look at repoze.who before reinventing the wheel.

17.1.2 Authorization Callback

Authorization is a bit more complex, because your middleware would need to interpret the request to get information required for the authorization (e.g. layer names for WMS GetMap requests). Limiting the GetCapabilities response to certain layers would even require the middleware to manipulate the XML document. So it's obvious that some parts of the authorization should be handled by MapProxy.

MapProxy can call the middleware back for authorization as soon as it knows what to ask for (e.g. the layer names of a WMS GetMap request). You have to pass a callback function to the environment so that MapProxy knows what to call.

Here is a more elaborate example that denies requests to all layers that start with a specific prefix. These layers are also hidden from capability documents.

```
class SimpleAuthFilter(object):
    """
    Simple MapProxy authorization middleware.

    It authorizes WMS requests for layers where the name does
    not start with `prefix`.
    """
    def __init__(self, app, prefix='secure'):
        self.app = app
        self.prefix = prefix

    def __call__(self, environ, start_response):
        # put authorize callback function into environment
        environ['mapproxy.authorize'] = self.authorize
        return self.app(environ, start_response)

    def authorize(self, service, layers=[], environ=None, **kw):
        allowed = denied = False
        if service.startswith('wms.'):
            auth_layers = {}
            for layer in layers:
                if layer.startswith(self.prefix):
                    auth_layers[layer] = {}
                    denied = True
                else:
                    auth_layers[layer] = {
                        'map': True,
                        'featureinfo': True,
                        'legendgraphic': True,
```

```
                }
                allowed = True
        else: # other services are denied
          return {'authorized': 'none'}

        if allowed and not denied:
            return {'authorized': 'full'}
        if denied and not allowed:
            return {'authorized': 'none'}
        return {'authorized': 'partial', 'layers': auth_layers}
```

And here is the part of the `config.py` where we define the filter and pass custom options:

```
application = make_wsgi_app('./mapproxy.yaml')
application = SimpleAuthFilter(application, prefix='secure')
```

17.2 MapProxy Authorization API

MapProxy looks in the request environment for a `mapproxy.authorize` entry. This entry should contain a callable (function or method). If it does not find any callable, then MapProxy assumes that authorization is not enabled and that all requests are allowed.

The signature of the authorization function:

authorize(*service*, *layers*=[ ], *environ=None*, ***kw*)

Parameters

- **service** – service that should be authorized
- **layers** – list of layer names that should be authorized
- **environ** – the request environ

Return type dictionary with authorization information

The arguments might get extended in future versions of MapProxy. Therefore you should collect further arguments in a catch-all keyword argument (i.e. `**kw`).

Note: The actual name of the callable is insignificant, only the environment key `mapproxy.authorize` is important.

The `service` parameter is a string and the content depends on the service that calls the authorize function. Generally, it is the lower-case name of the service (e.g. `tms` for TMS service), but it can be different to further control the service (e.g. `wms.map`).

The function should return a dictionary with the authorization information. The expected content of that dictionary can vary with each service. Only the `authorized` key is consistent with all services.

The `authorized` entry can have four values.

full The request for the given *service* and *layers* is fully authorized. MapProxy handles the request as if there is no authorization.

partial Only parts of the request are allowed. The dictionary should contains more information on what parts of the request are allowed and what parts are denied. Depending on the service, MapProxy can then filter the request based on that information, e.g. return WMS Capabilities with permitted layers only.

none The request is denied and MapProxy returns an HTTP 403 (Forbidden) response.

unauthenticated The request(er) was not authenticated and MapProxy returns an HTTP 401 response. Your middleware can capture this and ask the requester for authentication. `repoze.who`'s `PluggableAuthenticationMiddleware` will do this for example.

New in version 1.1.0: The `environment` parameter and support for `authorized: unauthenticated` results.

17.2.1 `limited_to`

You can restrict the geographical area for each request. MapProxy will clip each request to the provided geometry – areas outside of the permitted area become transparent.

Depending on the service, MapProxy supports this clipping for the whole request or for each layer. You need to provide a dictionary with `bbox` or `geometry` and the `srs` of the geometry. The following geometry values are supported:

BBOX: Bounding box as a list of minx, miny, maxx, maxy.

WKT polygons: String with one or more polygons and multipolygons as WKT. Multiple WKTs must be delimited by a new line character. Return this type if you are getting the geometries from a spatial database.

Shapely geometry: Shapely geometry object. Return this type if you already processing the geometries in your Python code with Shapely.

Here is an example callback result for a WMS *GetMap* request with all three geometry types. See below for examples for other services:

```
{
  'authorized': 'partial',
  'layers': {
    'layer1': {
      'map': True,
      'limited_to': {
        'geometry': [-10, 0, 30, 50],
        'srs': 'EPSG:4326',
      },
    },
    'layer2': {
      'map': True,
      'limited_to': {
        'geometry': 'POLYGON((...))',
        'srs': 'EPSG:4326',
      },
    },
    'layer3': {
      'map': True,
      'limited_to': {
        'geometry': shapely.geometry.Polygon(
          [(-10, 0), (30, -5), (30, 50), (20, 50)]),
        'srs': 'EPSG:4326',
      }
    }
  }
}
```

Performance

The clipping is quite fast, but if you notice that the overhead is to large, you should reduce the complexity of the geometries returned by your authorization callback. You can improve the performance by returning the geometry in the projection from `query_extent`, by limiting it to the `query_extent` and by simplifing the geometry. Refer to the `ST_Transform`, `ST_Intersection` and `ST_SimplifyPreserveTopology` functions when you query the geometries from PostGIS.

17.3 WMS Service

The WMS service expects a `layers` entry in the authorization dictionary for `partial` results. `layers` itself should be a dictionary with all layers. All missing layers are interpreted as denied layers.

Each layer contains the information about the permitted features. A missing feature is interpreted as a denied feature.

Here is an example result of a call to the authorize function:

```
{
  'authorized': 'partial',
  'layers': {
    'layer1': {
      'map': True,
      'featureinfo': False,
    },
    'layer2': {
      'map': True,
      'featureinfo': True,
    }
  }
}
```

17.3.1 `limited_to`

New in version 1.4.0.

The WMS service supports `limited_to` for *GetCapabilities*, *GetMap* and *GetFeatureInfo* requests. MapProxy will modify the bounding box of each restricted layer for *GetCapabilities* requests. *GetFeatureInfo* requests will only return data if the info coordinate is inside the permitted area. For *GetMap* requests, MapProxy will clip each layer to the provided geometry – areas outside of the permitted area become transparent or colored in the *bgcolor* of the WMS request.

You can provide the geometry for each layer or for the whole request.

See *limited_to* for more details.

Here is an example callback result with two limited layers and one unlimited layer:

```
{
  'authorized': 'partial',
  'layers': {
    'layer1': {
      'map': True,
      'limited_to': {
        'geometry': [-10, 0, 30, 50],
        'srs': 'EPSG:4326',
      },
    },
    'layer2': {
      'map': True,
      'limited_to': {
        'geometry': 'POLYGON((...))',
        'srs': 'EPSG:4326',
      },
    },
    'layer3': {
      'map': True,
    }
  }
}
```

Here is an example callback result where the complete request is limited:

```
{
  'authorized': 'partial',
  'limited_to': {
    'geometry': shapely.geometry.Polygon(
      [(-10, 0), (30, -5), (30, 50), (20, 50)]),
    'srs': 'EPSG:4326',
  },
  'layers': {
    'layer1': {
      'map': True,
    },
  }
}
```

17.3.2 Service types

The WMS service uses the following service strings:

wms.map

This is called for WMS GetMap requests. `layers` is a list with the actual layers to render, that means that group layers are resolved. The `map` feature needs to be set to `True` for each permitted layer. The whole request is rejected if any requested layer is not permitted. Resolved layers (i.e. sub layers of a requested group layer) are filtered out if they are not permitted.

New in version 1.1.0: The `authorize` function gets called with an additional `query_extent` argument:

authorize(*service*, *environ*, *layers*, *query_extent*, ***kw*)

Parameters **query_extent** – a tuple of the SRS (e.g. `EPSG:4326`) and the BBOX of the request to authorize.

Example

With a layer tree like:

```
- name: layer1
  layers:
    - name: layer1a
      sources: [l1a]
    - name: layer1b
      sources: [l1b]
```

An authorize result of:

```
{
  'authorized': 'partial',
  'layers': {
    'layer1':  {'map': True},
    'layer1a': {'map': True}
  }
}
```

Results in the following:

- A request for `layer1` renders `layer1a`, `layer1b` gets filtered out.
- A request for `layer1a` renders `layer1a`.
- A request for `layer1b` is rejected.

- A request for `layer1a` and `layer1b` is rejected.

wms.featureinfo

This is called for WMS GetFeatureInfo requests and the behavior is similar to `wms.map`.

wms.capabilities

This is called for WMS GetCapabilities requests. `layers` is a list with all named layers of the WMS service. Only layers with the `map` feature set to `True` are included in the capabilities document. Missing layers are not included.

Sub layers are only included when the parent layer is included, since authorization interface is not able to reorder the layer tree. Note, that you are still able to request these sub layers (see `wms.map` above).

Layers that are queryable and only marked so in the capabilities if the `featureinfo` feature set to `True`.

With a layer tree like:

```
- name: layer1
  layers:
    - name: layer1a
      sources: [l1a]
    - name: layer1b
      sources: [l1b]
    - name: layer1c
      sources: [l1c]
```

An authorize result of:

```
{
  'authorized': 'partial',
  'layers': {
    'layer1':  {'map': True, 'feature': True},
    'layer1a': {'map': True, 'feature': True},
    'layer1b': {'map': True},
    'layer1c': {'map': True},
  }
}
```

Results in the following abbreviated capabilities:

```
<Layer queryable="1">
  <Name>layer1</Name>
  <Layer queryable="1"><Name>layer1a</Name></Layer>
  <Layer><Name>layer1b</Name></Layer>
</Layer>
```

17.4 TMS/Tile Service

The TMS service expects a `layers` entry in the authorization dictionary for `partial` results. `layers` itself should be a dictionary with all layers. All missing layers are interpreted as denied layers.

Each layer contains the information about the permitted features. The TMS service only supports the `tile` feature. A missing feature is interpreted as a denied feature.

Here is an example result of a call to the authorize function:

```
{
  'authorized': 'partial',
  'layers': {
```

```
    'layer1': {'tile': True},
    'layer2': {'tile': False},
  }
}
```

The TMS service uses `tms` as the service string for all authorization requests.

Only layers with the `tile` feature set to `True` are included in the TMS capabilities document (`/tms/1.0.0`). Missing layers are not included.

The `authorize` function gets called with an additional `query_extent` argument for all tile requests:

authorize(*service*, *environ*, *layers*, *query_extent=None*, ***kw*)

Parameters **query_extent** – a tuple of the SRS (e.g. `EPSG:4326`) and the BBOX of the request to authorize, or `None` for capabilities requests.

17.4.1 limited_to

New in version 1.5.0.

MapProxy will clip each tile to the provided geometry – areas outside of the permitted area become transparent. MapProxy will return PNG images in this case.

Here is an example callback result where the tile request is limited:

```
{
  'authorized': 'partial',
  'limited_to': {
    'geometry': shapely.geometry.Polygon(
      [(-10, 0), (30, -5), (30, 50), (20, 50)]),
    'srs': 'EPSG:4326',
  },
  'layers': {
    'layer1': {
      'tile': True,
    },
  }
}
```

New in version 1.5.1.

You can also add the limit to the layer and mix it with properties used for the other services:

```
{
  'authorized': 'partial',
  'layers': {
    'layer1': {
      'tile': True,
      'map': True,
      'limited_to': {
        'geometry': shapely.geometry.Polygon(
          [(-10, 0), (30, -5), (30, 50), (20, 50)]),
        'srs': 'EPSG:4326',
      },
    'layer2': {
      'tile': True,
      'map': False,
      'featureinfo': True,
      'limited_to': {
        'geometry': shapely.geometry.Polygon(
          [(0, 0), (20, -5), (30, 50), (20, 50)]),
        'srs': 'EPSG:4326',
      },
```

```
    },
  }
}
```

See *limited_to* for more details.

17.5 KML Service

The KML authorization is similar to the TMS authorization, including the `limited_to` option.

The KML service uses `kml` as the service string for all authorization requests.

17.6 WMTS Service

The WMTS authorization is similar to the TMS authorization, including the `limited_to` option.

The WMTS service uses `wmts` as the service string for all authorization requests.

17.7 Demo Service

The demo service only supports `full` or `none` authorization. `layers` is always an empty list. The demo service does not authorize the services and layers that are listed in the overview page. If you permit a user to access the demo service, then he can see all services and layers names. However, access to these services is still restricted to the according authorization.

The service string is `demo`.

17.8 MultiMapProxy

The *MultiMapProxy* application stores the instance name in the environment as `mapproxy.instance_name`. This information in not available when your middleware gets called, but you can use it in your authorization function.

Example that rejects MapProxy instances where the name starts with `secure`.

```
class MultiMapProxyAuthFilter(object):
    def __init__(self, app, global_conf):
        self.app = app

    def __call__(self, environ, start_response):
        environ['mapproxy.authorize'] = self.authorize
        return self.app(environ, start_response)

    def authorize(self, service, layers=[]):
        instance_name = environ.get('mapproxy.instance_name', '')
        if instance_name.startswith('secure'):
            return {'authorized': 'none'}
        else:
            return {'authorized': 'full'}
```

CHAPTER

EIGHTEEN

DECORATE IMAGE

MapProxy provides the ability to update the image produced in response to a WMS GetMap or Tile request prior to it being sent to the client. This can be used to decorate the image in some way such as applying an image watermark or applying an effect.

Note: Some Python programming and knowledge of WSGI and WSGI middleware is required to take advantage of this feature.

18.1 Decorate Image Middleware

The ability to decorate the response image is implemented as WSGI middleware in a similar fashion to how *authorization* is handled. You must write a WSGI filter which wraps the MapProxy application in order to register a callback which accepts the ImageSource to be decorated.

The callback is registered by assigning a function to the key `decorate_img` in the WSGI environment. Prior to the image being sent in the response MapProxy checks the environment and calls the callback passing the ImageSource and a number of other parameters related to the current request. The callback must then return a valid ImageSource instance which will be sent in the response.

18.1.1 WSGI Filter Middleware

A simple middleware that annotates each image with information about the request might look like:

```
from mapproxy.image import ImageSource
from PIL import ImageColor, ImageDraw, ImageFont


def annotate_img(image, service, layers, environ, query_extent, **kw):
    # Get the PIL image and convert to RGBA to ensure we can use black
    # for the text
    img = image.as_image().convert('RGBA')

    text = ['service: %s' % service]
    text.append('layers: %s' % ', '.join(layers))
    text.append('srs: %s' % query_extent[0])

    text.append('bounds:')
    for coord in query_extent[1]:
        text.append('  %s' % coord)

    draw = ImageDraw.Draw(img)
    font = ImageFont.load_default()
    fill = ImageColor.getrgb('black')

    line_y = 10
    for line in text:
```

```
        line_w, line_h = font.getsize(line)
        draw.text((10, line_y), line, font=font, fill=fill)
        line_y = line_y + line_h

    # Return a new ImageSource specifying the updated PIL image and
    # the image options from the original ImageSource
    return ImageSource(img, image.image_opts)

class RequestInfoFilter(object):
    """
    Simple MapProxy decorate_img middleware.

    Annotates map images with information about the request.
    """
    def __init__(self, app, global_conf):
        self.app = app

    def __call__(self, environ, start_response):
        # Add the callback to the WSGI environment
        environ['mapproxy.decorate_img'] = annotate_img

        return self.app(environ, start_response)
```

You need to wrap the MapProxy application with your custom decorate_img middleware. For deployment scripts it might look like:

```
application = make_wsgi_app('./mapproxy.yaml')
application = RequestInfoFilter(application)
```

For PasteDeploy you can use the `filter-with` option. The `config.ini` looks like:

```
[app:mapproxy]
use = egg:MapProxy#app
mapproxy_conf = %(here)s/mapproxy.yaml
filter-with = requestinfo

[filter:requestinfo]
paste.filter_app_factory = mydecoratemodule:RequestInfoFilter

[server:main]
...
```

18.2 MapProxy Decorate Image API

The signature of the decorate_img function:

decorate_img(*image*, *service*, *layers*=[ ], *environ=None*, *query_extent=None*, ***kw*)

Parameters

- **image** – ImageSource instance to be decorated
- **service** – service associated with the current request (e.g. `wms.map`, `tms` or `wmts`)
- **layers** – list of layer names specified in the request
- **environ** – the request WSGI environment
- **query_extent** – a tuple of the SRS (e.g. `EPSG:4326`) and the BBOX of the request

Return type ImageSource

The `environ` and `query_extent` parameters are optional and can be ignored by the callback. The arguments might get extended in future versions of MapProxy. Therefore you should collect further arguments in a catch-all keyword argument (i.e. `**kw`).

Note: The actual name of the callable is insignificant, only the environment key `mapproxy.decorate_img` is important.

The function should return a valid ImageSource instance, either the one passed or a new instance depending the implementation.

CHAPTER

NINETEEN

DEVELOPMENT

You want to improve MapProxy, found a bug and want to fix it? Great! This document points you to some helpful information.

19.1 Source

Releases are available from the PyPI project page of MapProxy. There is also an archive of all releases.

MapProxy uses Git as a source control management tool. If you are new to distributed SCMs or Git we recommend to read Pro Git.

The main (authoritative) repository is hosted at http://github.com/mapproxy/mapproxy

To get a copy of the repository call:

```
git clone https://github.com/mapproxy/mapproxy
```

If you want to contribute a patch, please consider creating a "fork" instead. This makes life easier for all of us.

19.2 Documentation

This is the documentation you are reading right now. The raw files can be found in `doc/`. The HTML version user documentation is build with Sphinx. To rebuild this documentation install Sphinx with `pip install sphinx sphinx-bootstrap-theme` and call `python setup.py build_sphinx`. The output appears in `build/sphinx/html`. The latest documentation can be found at `http://mapproxy.org/docs/lates/`.

19.3 Issue Tracker

We are using the issue tracker at GitHub to manage all bug reports, enhancements and new feature requests for MapProxy. Go ahead and create new tickets. Feel free to post to the mailing list first, if you are not sure if you really found a bug or if a feature request is in the scope of MapProxy.

19.4 Tests

MapProxy contains lots of automatic tests. If you don't count in the `mapproxy-seed`-tool and the WSGI application, the test coverage is around 95%. We want to keep this number high, so all new developments should include some tests.

MapProxy uses Nose as a test loader and runner. To install Nose and all further test dependencies call:

```
pip install -r requirements-tests.txt
```

To run the actual tests call:

```
nosetests
```

19.4.1 Available tests

We distinguish between doctests, unit, system tests.

Doctests

Doctest are embedded into the source documentation and are great for documenting small independent functions or methods. You will find lots of doctest in the `mapproxy.core.srs` module.

Unit tests

Tests that are a little bit more complex, eg. that need some setup or state, are put into `mapproxy.tests.unit`. To be recognized as a test all functions and classes should be prefixed with `test_` or `Test`. Refer to the existing tests for examples.

System tests

We have some tests that will start the whole MapProxy application, issues requests and does some assertions on the responses. All XML responses will be validated against the schemas in this tests. These test are located in `mapproxy.tests.system`.

19.5 Communication

19.5.1 Mailing list

The preferred medium for all MapProxy related discussions is our mailing list mapproxy@lists.osgeo.org You must subscribe to the list before you can write. The archive is available here.

19.5.2 IRC

There is also a channel on Freenode: `#mapproxy`. It is a quiet place but you might find someone during business hours (central european time).

19.6 Tips on development

You are using *virtualenv* as described in *Installation*, right?

Before you start hacking on MapProxy you should install it in development-mode. In the root directory of MapProxy call `pip install -e ./`. Instead of installing and thus copying MapProxy into your *virtualenv*, this will just link to your source directory. If you now start MapProxy, the source from your MapProxy directory will be used. Any change you do in the code will be available if you restart MapProxy. If you use the `mapproxy-util serve-develop` command, any change in the source will issue a reload of the MapProxy server.

19.7 Coding Style Guide

MapProxy generally follows the Style Guide for Python Code. With the only exception that we permit a line width of about 90 characters.

CHAPTER

TWENTY

MAPPROXY 2.0

MapProxy will change a few defaults in the configuration between 1.8 and 2.0. You might need to adapt your configuration to have MapProxy 2.0 work the same as MapProxy 1.8 or 1.7.

Most changes are made to make things more consistent, to make it easier for new users and to discourage a few deprecated things.

> **Warning:** Please read this document carefully. Also check all warnings that the latest 1.8 version of *mapproxy-util serve-develop* will generate with your configuration before upgrading to 2.0.

20.1 Grids

20.1.1 New default tile grid

MapProxy now uses GLOBAL_WEBMERCATOR as the default grid, when no grids are configured for a cache or a tile source. This grid is compatible with Google Maps and OpenStreetMap, and uses the same tile origin as the WMTS standard.

The old default GLOBAL_MERCATOR uses a different tile origin (lower-left instead of upper-left) and you need to set this grid if you upgrade from MapProxy 1 and have caches or tile sources without an explicit grid configured.

MapProxy used the lower-left tile in a tile grid as the origin. This is the same origin as the TMS standard uses. Google Maps, OpenStreetMap and now also WMTS are counting tiles from the upper-left tile. MapProxy changes

20.1.2 Default origin

The default origin changes from 'll' (lower-left) to 'ul' (upper-left). You need to set the origin explicitly if you use custom grids. The origin will stay the same if your custom grid is *base'd on the 'GLOBAL_** grids.

20.2 WMS

20.2.1 SRS

The WMS does not support EPSG:900913 by default anymore to discourage the use of this deprecated EPSG code. Please use EPSG:3857 instead or add it back to the WMS configuration (see *srs*).

20.2.2 Image formats

PNG and JPEG are the right image formats for almost all use cases. GIF and TIFF are therefore no longer enabled by default. You can enable them back in the WMS configuration if you need them (*image_formats*)

20.3 Other

This document will be extended.

CHAPTER

TWENTYONE

INDICES AND TABLES

- *genindex*
- *search*

Symbols

A

C

D

G

K

L

M

O

P

R

S

T

W