HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

COMPUTER SCIENCE

IFS INSTITUTE FOR SOFTWARE

Web Map Tile Services
Tiny Tile Server

Bachelor Thesis

Department of Computer Science
University of Applied Science Rapperswil

Spring Term 2013

Author:	Carmen Campos Bordons
Advisor:	Prof. Stefan Keller, HSR
Project Partner:	Klokan Technologies, Baar
External Co-Examiner:	Claude Eisenhut, Burgdorf
Internal Co-Examiner:	Prof. Dr. Andreas Rinkel, HSR

Abstract

Tiny Tile Server is a Python server that permits the user to display local MBTiles maps on the internet. It extracts the data from the SQLite database where the map information is stored in tables containing all the tiles, UTFGrid and metadata.
The tiles are the map images, smaller than the screen for better performance. The UTFGrid is some extra information related with points in the map that appears in an infobox when the user interact with these points. The metadata is the information about the map: name, description, bounds, legend, center, minzoom, maxzoom.
Tiny Tile Server shows the tiles composing the map on a website and the UTFGrid data on top of the tiles. It can also be used to show the getCapabilities information from Web Map Tile Service in XML format extracted by the metadata table.

Tiny Tile Server supports two protocols to access the tiles: direct access with XYZ tile request to tiles in a directory or to MBTiles database; or Web Map Tile Service from a MBTiles database.
The server is a part in a website whose purpose is to show how it works and provide templates for the user who wants to employ it, so he will not need to have programming knowledge in order to use Tiny Tile Server, just to follow a simple installation tutorial.

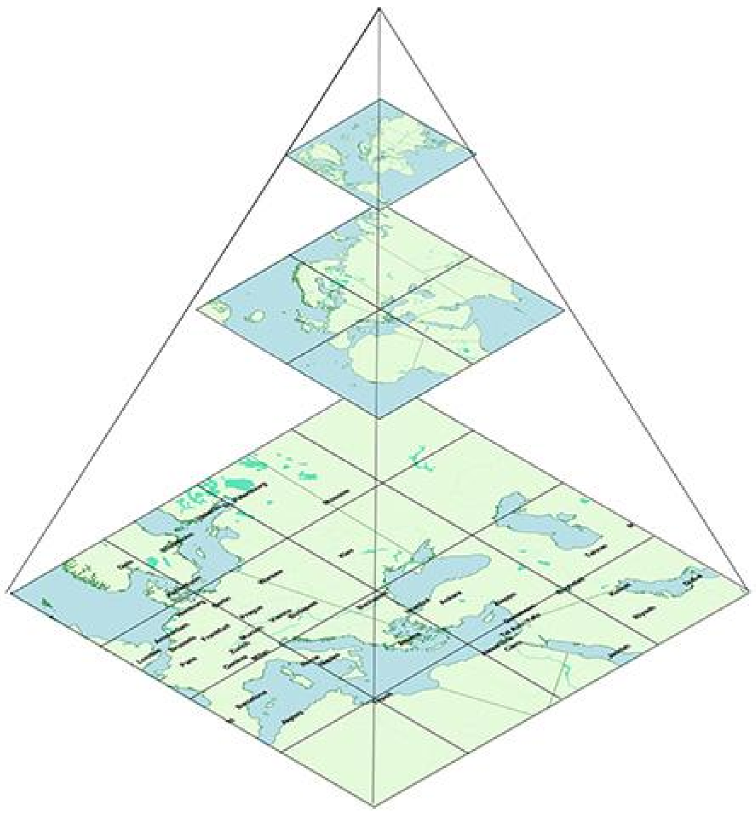
To display the map, the user can choose between two different kind of templates: using OpenLayers or Leaflet, which are libraries specialized in creating maps.
Because the user will need to have a map in a MBTiles database, tutorials about how to create a map there are also provided.

Website: http://152.96.56.43/
Repository: https://github.com/carmencampos/WMTS

Management summary

Introduction

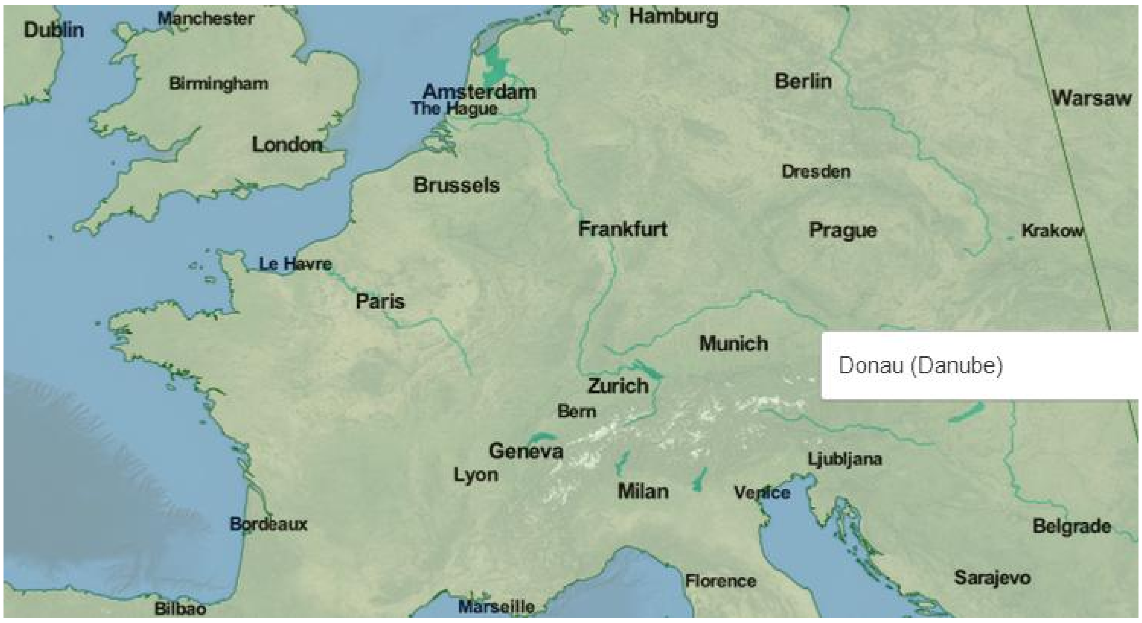
This project has been performed with the main objective of displaying local maps in MBTiles format on the internet. That is the purpose of Tiny Tile Server. But also a website has been developed with templates showing examples that the final user will only need to adapt to display his maps. There are also tutorials about how to create your own map.



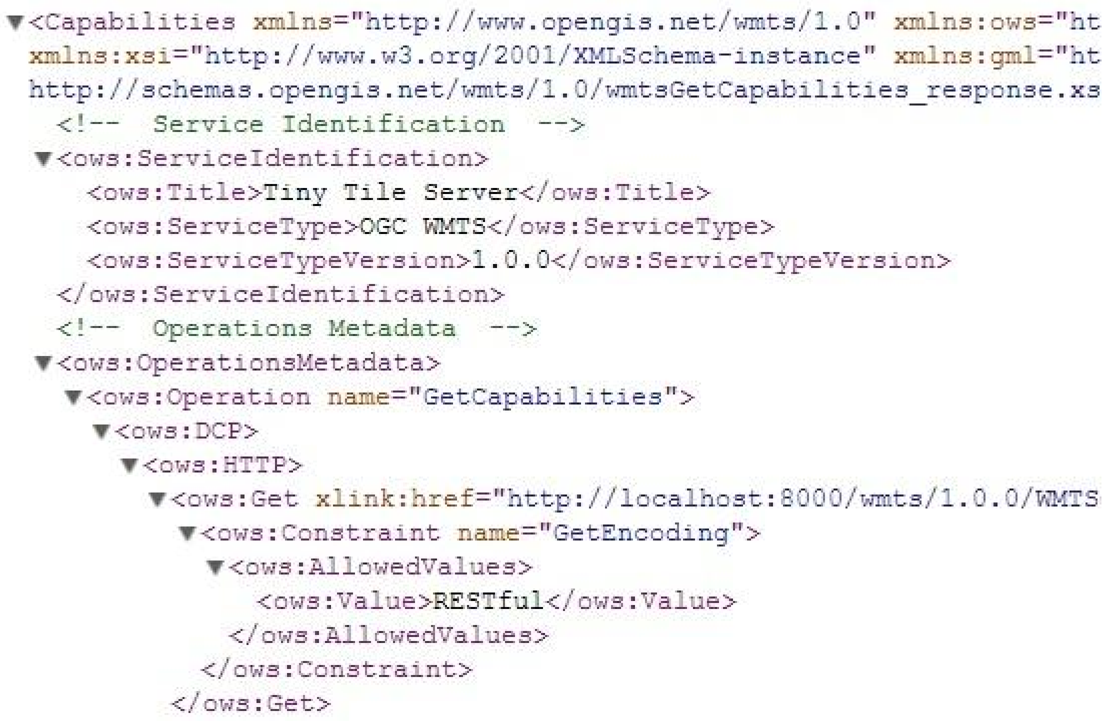
There are already similar projects related with serving tiles stored in a MBTiles file. What makes Tiny Tile Server different is that it is also focused on users who do not need to have programming skills, which is why it is simpler and easier to use.
Tiny Tile Server could also be used by developers to add new functionalities on it.

Results

Tiny Tile Server extracts the data from the SQLite database where the map information is stored in tables containing all the tiles, UTFGrid and metadata.



Then it shows the tiles composing the map on a website and the UTFGrid data on top of the tiles. It can also be used to show the getCapabilities information from Web Map Tile Service in XML format extracted by the metadata table.


```
▼<Capabilities xmlns="http://www.opengis.net/wmts/1.0" xmlns:ows="ht
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:gml="ht
 http://schemas.opengis.net/wmts/1.0/wmtsGetCapabilities_response.xs
   <!--  Service Identification  -->
 ▼<ows:ServiceIdentification>
     <ows:Title>Tiny Tile Server</ows:Title>
     <ows:ServiceType>OGC WMTS</ows:ServiceType>
     <ows:ServiceTypeVersion>1.0.0</ows:ServiceTypeVersion>
   </ows:ServiceIdentification>
   <!--  Operations Metadata  -->
 ▼<ows:OperationsMetadata>
   ▼<ows:Operation name="GetCapabilities">
     ▼<ows:DCP>
       ▼<ows:HTTP>
         ▼<ows:Get xlink:href="http://localhost:8000/wmts/1.0.0/WMTS
           ▼<ows:Constraint name="GetEncoding">
             ▼<ows:AllowedValues>
                <ows:Value>RESTful</ows:Value>
              </ows:AllowedValues>
            </ows:Constraint>
          </ows:Get>
```

Tiny Tile Server supports two protocols to access the tiles: direct access with XYZ tile request to tiles in a directory or to MBTiles database; or Web Map Tile Service from a MBTiles database.

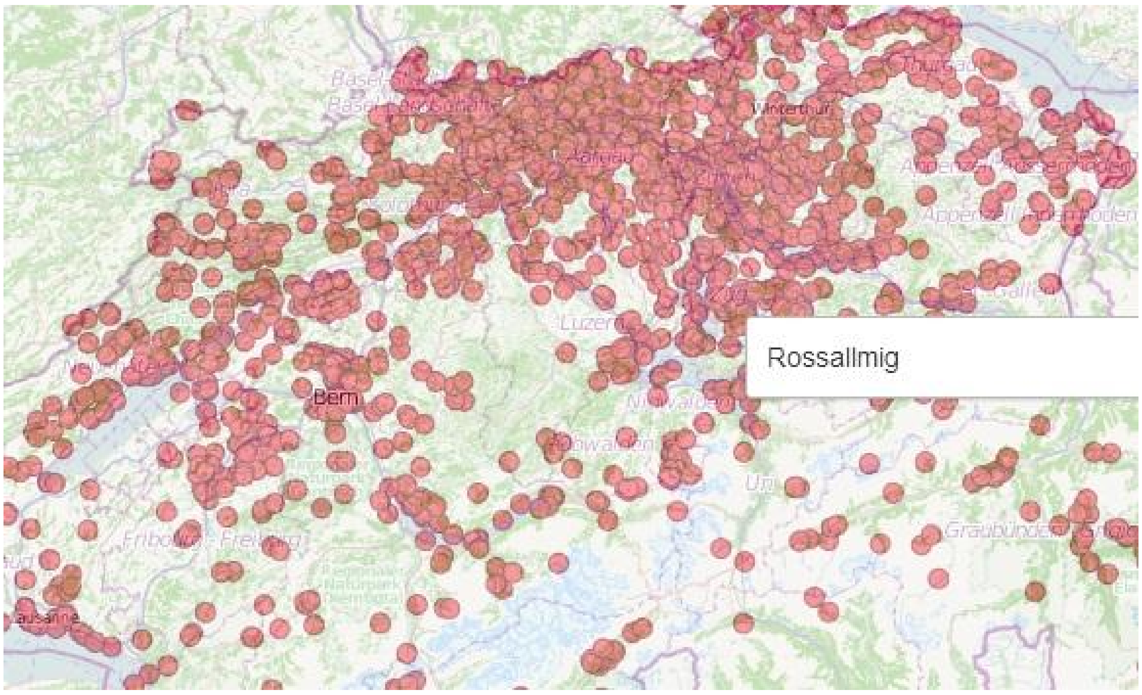
Technologies

The frontend is programmed using HTML5, CSS3 and JavaScript. There are also used two libraries specialized in creating maps: Leaflet and OpenLayers. These libraries permits us to display our thematic map on the internet on top of a base layer with topographic information,

for that purpose is used OpenStreetMap. They also permit to add some functionalities in our map, like zoom, scale, legend.
The backend is written in Python and using Bottle as only external dependency. Bottle is necessary to handle requests/responses to the server, to get and to operate with the variables and to simplify routing.

Outlook

The website can be used for any kind of user who wants to display its thematic maps on the internet. The workspace is easy to reproduce and do not need any external reference apart from installing Python. With the templates, which are well documented, the user only need to change the part of the code to reference its maps. There are also some tutorials to create a map in MBTiles format, in case the user needs it.



Tiny Tile Server can be installed in a different scope instead of a Python website. Using Tiny Tile Server in a different scope that the website could be difficult for a normal user with no programming knowledge. This is completely possible, but it is more focused for web developers who prefer to integrate this server in another application or use a different web framework instead of Bottle.

Website: http://152.96.56.43/
Repository: https://github.com/carmencampos/WMTS

Index

Definition of the project

Background

The target audience of this thesis are people with basic GIS knowledge who want to publish their data (raster=> GeoTIFF and/or vector => Spatialite/ShapeFiles) by using open-source software. This thesis also contains a website with a gallery and practical exercises of map tile rendering and online publishing in the cloud (free services) or on own servers. For online publishing on own servers templates for server and clients need to be developed. Finally, the thesis optionally evaluates and demonstrates 'enhanced' raster tiles (UTFGrid) and pure vector tiling approaches (Kothic, OpenStreetMap Watch List/OWL).

State-of-the-art web mapping applications available today are using tiles as their core API for publishing maps. This thesis explains the basics about tiling services and shows the open-source software typically used for generating and distributing tiled maps on any ordinary web server (like Apache) in standardized tiling formats and with minimal or no need to install, configure or maintain any additional software.

Hosting of the rendered maps is also possible from the cloud, such as MapBox, Amazon S3/CloudFront or RackSpace Files. The published custom maps can be used in HTML5/JavaScript applications, on mobile devices (iOS: MapKit/RouteMe, Android: OSMDroid), or even in desktop GIS systems such as QGIS (open source) and ESRI ArcGIS Desktop (commercial).

Tasks Assignment and Deliverables

There are three main goals for this thesis:

1. Delivering an overview of a state-of-the-art of (web) map tiling standards, open source software and (free) services.
2. Producing a gallery of examples and introductory texts to selected map tiling software.
3. Programming templates to be used by web programmers who want to publish a topic as interactive web maps on own servers.
4. Part of the third goal is the implementation of an own tiling server software, called TinyTileServer (Python).

In the following there are main chapters to be covered by this thesis.

Getting to know TileMill, the "Map Design Studio"

1. About TileMill and CartoCSS (see also http://giswiki.hsr.ch/TileMill)
2. Preparing own vector data (files, databases)
3. Working with TileMill: Publishing rendered own vector data on a hosted server: Uploading data to MapBox,
    1. then publishing at MapBox directly (Ex.0 see http://api.tiles.mapbox.com/v3/carmencampos.example.html).
    2. Publishing an own server: After exporting as MBTiles,
    3. Uploading data to MapBox and integrating in own webpage.
4. Publishing maps from own server. Examples:
    1. Simple example in Leaflet (Webpage Ex.1) / OpenLayers (Webpage Ex.2) with OSM basemap
    2. Example using SQL with OSM basemap

    3. Example with own thematic data/picnic sites) (Webpage Ex. 3)
5. Publishing maps from own server using own TinyTileServer to read MBTiles
    1. clients with Leaflet plus jQuery and Wax (Webpage Ex. 4)
    2. clients with OpenLayers plus jQuery and Wax (Webpage Ex. 5)
    3. Own basemap "Switzerland" as complex example (OpenStreetMap style) (Webpage Ex. 6)
6. Working with raster (GeoTIFF) and grid files.

About Tiling and Raster Formats

1. GeoTIFF
2. MBTiles format (including UTFGrid)
3. Convert raster to tiles using command line tools like GDAL2Tiles, MapTiler, raster2mbtiles, mbutil, MapCache.
4. Convert raster to tiles using GUIs like TileMill and QGIS (generating MBTiles, generating GeoTIFF hierarchies)
5. An example (GeoTIFF)
6. Caching and seeding: MapProxy (can read also MBTiles), MapCache and GeoWebCache

Converting the MBTiles Format

1. Converting File Raster Formats to MBTiles (evaluating 'raster2mb')
2. Converting MBTiles to Raster Formats

About Raster and Tiles on the QGIS Desktop

1. Raster Images in QGIS (using example as raster file)
2. Web Map Tiling Services in QGIS (using example as WMS)
3. MBTiles in QGIS (via GDAL; own scripting task using Python console (ev. plugin?))

About Tiles in the PostgreSQL database

1. Overview of PostGIS Raster

Beyond Web Map Tiling

1. Overview
2. Vector Tiles - an overview (evaluate server code)

TinyTileServer - a Tiny Web Map Tiling Webserver (Software Project)

- Overview
- Configuration (e.g. for GetCapabilities)
- Usage

Deliverables

(basically everything in english):

- BA thesis documentation according to the guidelines of the department and advisor (incl. a general part I and a SW engineering part II).
- Sep. document containing exercises
- Website with a show case and a gallery
- Ev. other helper tools
- Source Code (MIT license if possible)

Licensing agreement

Rapperswil, 14.06.2013	……………………………………………………..
	The student
Rapperswil, 14.06.2013	……………………………………………………..
	The advisor

Author declaration (Eigenständigkeitserklärung)

Carmen Campos Bordons

Ich erkläre hiermit,

- dass ich die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt habe, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde,
- dass ich sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben habe,
- dass ich keine durch Copyright geschützten Materialen (z.B. Bilder) in dieser Arbeit in unerlaubter Weise genutzt habe.

Ort, Datum: Rapperswil, 14.06.2013

Name, Unterschrift: Carmen Campos Bordons,

Technical report

1. Introduction

1.2. Presentation of the problem

The main idea is showing our thematic maps online. A thematic map is a type of map or chart especially designed to show a particular theme connected with a specific geographic area. These maps can portray physical, social, political, cultural, economic, sociological, agricultural, or any other aspects of a city, state, region, nation, or continent.
A topographic map portraits the data in the relief, such as landforms, seas, rivers, lines of countries or cities… But in a thematic map we show a particular set of data about any specific theme on top of a previous general map that already exists.
We could display where are located the Lidl supermarkets around the world. Or the companies which started a new business in USA recently. Or which countries are catholic in Europe. Or where are bicycle streets in our city. Or the picnic sites in Switzerland.
There is also the possibility to make comparisons. To indicate that there is a different amount of Lidl supermarket in different countries, we can assign a different color depending on the amount of supermarket. But often we use the same color with different intensity, so when it is most intense it indicates more supermarkets.
Other possibility is to use a bigger form when the company was founded recently and a smaller one if it was some years ago. And instead of only show the catholic countries, we could show the kind of religion in every country using different colors; and we attached a legend where we indicate with which religion corresponds every color.
The possibilities are endless. Any idea you have can be showed in a thematic map.

1.3. Goals

And the possibilities to create these maps are endless too: QuantumGIS, ArcGIS, Mapnik, Maptitude, SpatialKey, Excel… But we would concentrate in TileMill.
TileMill is a fast and easy to use design environment to create interactive maps. It uses CartoCSS to apply the style in our maps, which let us modify and add visual elements in every layer.
We can load the data from different sources like ESRI Shapefile, KML, GeoJSON, GeoTIFF, PostGIS, CSV or SQLite. You can find out more here http://www.mapbox.com/tilemill/docs/manual/adding-layers/. If the information is just in a file, you just need to write the path to that file. Once you install TileMill there are some files that you can access easily from the MapBox folder. If you want to create your own data, or you need to edit the file you are going to use, you will need a geospatial information system (GIS). For example, to manipulate a shapefile you can use QuantumGIS.
If you decide to use an SQLite database to load the data, apart from the path to the file, you will also need to specify a SQL sentence where you indicate the table or subquery from where you will get the information. Here http://www.mapbox.com/tilemill/docs/tutorials/sqlite-work/ there is more information.

You could also use a PostGIS database to add your data; and in this case, beside the SQL sentence you will have to configure the connection with the database. Here http://www.mapbox.com/tilemill/docs/guides/postgis-work/ there is more information.
Then we could add our own style to show the layer, or just edit the one that appears by default. For every layer we get two kinds of information: 1) The location of the data; the boundary in case is a polygon, like a country or a city; the completely location in case is a line, like a river or a street; or the exact location in case is a point, like the situation of a museum or a hospital. 2) Some extra information about the data; for example, if we are considering countries: the name, the main city, the population, the language, the currency...
The last step is to export the map. TileMill can export maps to MBTiles, PNG, PDF, SVG, or Mapnik XML formats. We are going to concentrate ourselves in the MBTiles format, which is a database where we can store millions of tiles in a single file. Apart from the images of the map, UTFGrid information (some extra data we could use to add interaction to our map) and metadata about the map (like zoom levels, a description, a legend…) are also stored in the file.
Once we have our map, we are going to show it on the internet. In order to do that, we could store it in a website like MapBox or we could use our own server. For this purpose I have created Tiny Tile Server, a server in Python that extract the data from the MBTiles file and display the map on the internet.
Apart from the Python code, we also need a library to use in our client in JavaScript. There are several options like Modest Maps, Leaflet, Google Maps, ESRI, OpenLayers or Polymaps. I have concentrated in Leaflet and OpenLayers, so I have created templates for this two libraries. So when an user decide to display its maps, he will only need to change in the template the part of the code where is the path to the MBTiles file.

1.4. Conditions, environment, definitions and delimitations

The final target is that the maps created will be integrated with OpenStreetMap (OSM), which is a collaborative project to create a free editable map of the world. Rather than the map itself, the data generated by the OpenStreetMap project are considered their primary output. The data are then available for use in both traditional applications, like their usage by Craigslist and Foursquare to replace Google Maps, and more unusual roles, like replacing default data included with GPS receivers. These data have been favorably compared with proprietary datasources, though data quality varies worldwide.
The initial map data were collected from scratch by volunteers performing systematic ground surveys using a handheld GPS unit and a notebook, digital camera, or a voice recorder. These data were then entered into the OpenStreetMap database. More recently, the availability of aerial photography and other data sources from commercial and government sources has greatly increased the speed of this work and has allowed land-use data to be collected more accurately by the process of digitization. When especially large datasets become available, a technical team manages the conversion and import of the data.
OpenStreetMap uses a topological data structure, with four core elements (also known as data primitives):

- Nodes are points with a geographic position, stored as coordinates (pairs of a latitude and a longitude) according to WGS 84. Outside of their usage in ways, they are used to represent map features without a size, such as points of interest or mountain peaks.
- Ways are ordered lists of nodes, representing a polyline, or possibly a polygon if they form a closed loop. They are used both for representing linear features such as streets and rivers; and areas, like forests, parks, parking areas and lakes.
- Relations are ordered lists of nodes and ways (together called "members"), where each member can optionally have a "role" (a string). Relations are used for representing the relationship of existing nodes and ways. Examples include turn restrictions on roads, routes that span several existing ways (for instance, a long-distance motorway), and areas with holes.
- Tags are key-value pairs (both arbitrary strings). They are used to store metadata about the map objects (such as their type, their name and their physical properties). Tags are not free-standing, but are always attached to an object: to a node, a way, a relation, or to a member of an relation. A recommended ontology of map features (the meaning of tags) is maintained on a wiki.

1.5. Construction work

Tiny Tile Server has several points in common with the tools that already exists to display maps on the internet. Displaying maps on the internet is nothing new, and here only maps in MBTiles databases will be commented. MBTiles got really popular because you can store all the tiles in a single file, with is easy to handle; and being a database simplifies the access to the tiles.

Most of the tools related to Tiny Tile Server are focused in displaying maps from MBTiles hosted in another server, instead of serving tiles from your own server, like you do with Tiny Tile Server.

This is a small project, not only because of the little time to develop it, but also because that makes it a manageable application.

Python is also pretty common between those tools, but the reason for choosing it as the language programming is because of the server where the application was going to be deployed.

2. Initial situation

2.1. Existing approaches

The use of MBTiles is so extended because it possible to store and transfer web maps in a single file. And because SQLite is available on many platforms. That is why there are so many tools to work with them. It may happen that we have out map in other format, a set of tiles or a GeoTIFF file, and we want to transform it in a MBTiles file.
There are some tools related to serve MBTiles:

- Wax: add functionalities to a map shown in our website.
- tileserver-php : extract MBTiles for a local file and show them on a website.
- Leaflet.utfgrid: create a UTFGrid layer.
- TileStache
- TileCloud

These are some tools we could be interested in related with MBTiles:

- MBUutil: extract the tiles from a MBTiles database, or generate the MBTiles of a tiles directory.
- raster2mb: generates the MBTiles from a GeoTIFF file.
- GDAL2Tiles: generates a set of tiles from a GeoTIFF file.
- MapTiler: generates a set of tiles from a GeoTIFF file.
- landez: to manipulate MBTiles .
- python-mbtiles: to work with MBTiles databases.

Another projects:

- MapProxy
- MapCache

2.2. Brief description and characterization

Wax to add functionalities to our map

This tool can be found in this repository https://github.com/mapbox/wax or in its website http://www.mapbox.com/wax/.
Wax is a library that adds common utilities to minimal mapping libraries and aims to give developers and designers ultimate control and flexibility. It provides zoom controls and other stuff that people already expect, functionalities that can be found in some portals, but that our map does not show by default when we use our own server.
After downloading Wax, you just need to add in your HTML file the reference to the CSS and JavaScript files corresponding to the library you are going to use:

- Google Maps API v3 can be consulted here https://developers.google.com/maps/
- Leaflet 0.x.x can be consulted here http://leafletjs.com/
- Modest Maps 1.x.x can be consulted here http://modestmaps.com/
- OpenLayers 2.11 can be consulted here http://openlayers.org/

- ESRI ArcGIS API 2.8 can be consulted here http://developers.arcgis.com/en/javascript/

In the website there are specified the link of every file for the different libraries. Depending on the library you use, there are more or less functionalities. You could have:

- Legend: explains how you should interpret the map. Normally it is a small box or table on the image that explains what the symbols you can find mean.
- Tooltip: shows a particular information about every different point on the map (that has some information associated). It could be a point, a line or a polygon. And you can write any HTML you want that would be the same for all the points, and choose the characteristic(s) you want to display: like the name, the longitude and latitude, the number of people who lives there, the dimension… It is a small box that could appear always in the same place or near the point of what you want to show the information, in this case is called movetip.
- Zoom: a zoom control with in and out buttons.
- Point selector: adds points in the map by clicking in a specific places and then removes it by clicking in the point.
- Attribution: displays attribution information about the map.

Tileserver to serve MBTiles on a website

This tool can be found in this repository https://github.com/klokantech/tileserver-php/
It is a PHP server which distributes maps to desktop and web applications from an Apache + PHP web hosting.
It supports the following protocols:

- Direct access to XYZ tiles.
- OpenGIS WMTS 1.0.0, that can be consulted here http://www.opengeospatial.org/standards/wmts/
- OSGeo TMS 1.0.0, that can be consulted here http://wiki.osgeo.org/wiki/Tile_Map_Service_Specification
- TileJSON.js, that can be consulted here http://www.mapbox.com/developers/tilejson/

This server is similar to Tiny Tile Server, the initial idea was to do a server with the same functionality but programmed in Python.
The differences are that Tileserver also support the protocol TMS for accessing tiles, but it does not provide UTFGrid information.

Leaflet.utfgrid: a new kind of layer in Leaflet

This tool can be found in this repository https://github.com/danzel/Leaflet.utfgrid
It is for Leaflet, and you just need to download the code file to your directory and reference it when you want to use this layer.
It is used to create a layer in order to show the UTFGrid data on the map. It is used like any other kind of layer, just indicating the source when it needs to get the info.
You can add an event listener to specify what action should be performed depending on what the user does: hover, move, click… You can also show a different infobox depending on the case.
OpenLayers has already integrated the possibility of adding a UTFGrid layer.

TileStache

This tool can be found in its website http://tilestache.org/ or in this repository https://github.com/migurski/TileStache
It has many functionalities and it is mainly focused on the intersection between synthetic imagery, composites of existing imagery, and delivery of raw vector data to browsers. It is similar to TileCache but easier and more oriented to designers and creating a new kind of map. It is also written in Python, and it uses ModestMaps as a client.
Between its features, it serves tiles out of MBTiles tileset. And it also cache to MBTiles.

TileCloud

This tool can be found in this repository https://github.com/twpayne/tilecloud
It is a powerful utility for generating, managing, transforming, visualising and map tiles in multiple formats. It can create, read update, delete tiles in multiple back ends, which include MBTiles.
You can transform from a format to a different one, server tiles or render tiles from a server and save them in a MBTiles file.
It can also serve UTFGrid information or metadata in .json format.

Importing and exporting MBTiles using MBUtil

This tool can be found in this repository https://github.com/mapbox/mbutil
MBUtil can be used to generate a tiles directory from the MBTiles database, or the opposite, to generate the MBTiles database from a tiles directory.
In order to use it we need to have Python installed, and then we can copy the repository to our system. Also in the repository can be found some documentation about the description, installation and the usage.
Export a .mbtiles file to tiles on the filesystem:
→ mb-util name_of_file.mbtiles name_of_directory
Import a directory into a .mbtiles file
→ mb-util name_of_directory name_of_file.mbtiles

Create a mbtiles database of a GeoTIFF using raster2mb

This tool can be found in this repository https://github.com/crschmidt/raster2mb
As its name indicates, it is used to generate a MBTiles database using a raster file, like GeoTIFF, as input file.
In order to use it we need GDAL and Python, and then we have to copy the repository to our system. Also in the repository can be found some documentation about the description, installation and the usage.
We just need to type:
→ raster2mb name_of_file.tiff new_name.mbtiles

Create tiles of a GeoTIFF map using gdal2tiles

GDAL2Tiles is distributed together with GDAL library, that can be found here http://www.gdal.org/

One of the problems with high-resolution raster data is that it takes a lot of memory to display it. And if you are pushing it out over the net, you have bandwidth concerns as well. To address that problem, you have to create tiles.
GDAL2Tiles is a command line tool that allows easy publishing of raster maps on the Internet. The raster image is converted into a directory structure of small tiles which you can copy to your webserver.
Simple web pages with viewers based on Google Maps and OpenLayers are generated as well - so anybody can comfortably explore your maps on-line and you do not need to install or configure any special software; and the map is displayed very fast in the web browser. You only need to upload the generated directory onto a web server.
GDAL2Tiles also creates the necessary metadata for Google Earth (KML SuperOverlay), in case the supplied map uses EPSG:4326 projection.
You can use any image, but you need to know the boundaries of the image, the latitude and longitude of each of the corners of the image.

Once you have installed the GDAL libraries and selected the image, the first thing you have to do is to get some information about the image so that you can georeference it. Specifically, you need the pixel and line positions of each corner of the image.
GDAL provides a handy utility, gdalinfo, for capturing this information. At the command line, simply type

```
→ gdalinfo name_of_your_file
```

You will get something like this:

```
Driver: GTiff/GeoTIFF
Files: mundo_creado.tif
Size is 16200, 8100
Coordinate System is `'
GCP Projection =
GEOGCS["WGS 84",
  DATUM["WGS_1984",
    SPHEROID["WGS 84",6378137,298.257223563,
      AUTHORITY["EPSG","7030"]],
    AUTHORITY["EPSG","6326"]],
  PRIMEM["Greenwich",0],
  UNIT["degree",0.0174532925199433],
  AUTHORITY["EPSG","4326"]]
GCP[  0]: Id=1, Info=     (0,0) -> (-180,90,0)
GCP[  1]: Id=2, Info=       (16200,0) -> (180,90,0)
GCP[  2]: Id=3, Info=       (16200,8100) -> (180,-90,0)
Metadata:
 AREA_OR_POINT=Area
 EXIF_ColorSpace=1
 EXIF_DateTime=2007:06:27 19:49:19
 EXIF_Orientation=1
 EXIF_PixelXDimension=16200
 EXIF_PixelYDimension=8100
```

```
  EXIF_ResolutionUnit=2
  EXIF_Software=Adobe Photoshop CS2 Macintosh
  EXIF_XResolution=(72)
  EXIF_YResolution=(72)
Image Structure Metadata:
  INTERLEAVE=PIXEL
Corner Coordinates:
Upper Left  (    0.0,    0.0)
Lower Left  (    0.0, 8100.0)
Upper Right (16200.0,    0.0)
Lower Right (16200.0, 8100.0)
Center      ( 8100.0, 4050.0)
Band 1 Block=16200x1 Type=Byte, ColorInterp=Red
Band 2 Block=16200x1 Type=Byte, ColorInterp=Green
Band 3 Block=16200x1 Type=Byte, ColorInterp=BlueDriver: GTiff/GeoTIFF
```

The important information for this case are the Upper Left, Lower Left, Upper Right, Lower Right lines. These tell you the pixel and line values of each corner. In this case: the Upper Left is at 0,0; the Lower Left is at 0,8100; the Upper Right is at 16200,0; and the Lower Right is at 16200,8100

The next step will be georeferencing the image. Georeferencing in this case means to create metadata describing the geographic position of each of the corners of the image. Using the information gained before and gdal_translate, you can assign georeference information to the file.

→ gdal_translate -a_srs EPSG:4326 -gcp 0 0 -180 90 -gcp 16200 0 180 90 -gcp 16200 8100 180 -90 name_of_your_file.jpg new_name.tif

-a_srs assigns a spatial reference system to the file. That tells any application consuming it what coordinate system is being used. In this case, it is using EPSG:4326, which is the same as WGS84.

-gcp, or ground control point, assigns coordinates to positions in the file. For -gcp, define the gcp by setting the pixel and then line number, and then the longitude and latitude. Each of those is separated by a space.

The last two parameters are the origin file and the target file.

There are more options you can use, just type gdal_translate and you can see them.

The original image was not created for a round globe, it was created to appear to lie flat. In GIS terms, it is projected, which means that it is a two-dimensional representation of a three-dimensional object. Projection requires distorting the image so that it appears how you would expect a flat image of the Earth to look.

In order to get it to look right, you have to warp the image to fit the globe. Fortunately GDAL provides a great tool for that too. Simply type

→ gdalwarp -t_srs EPSG:4326 new_name.tif new_name _4326.tif

-t_srs indicates the spatial reference system we are going to define. This will create a new file, which provides metadata about the warping procedure.

There are more options you can use, just type gdalwarp and you can see them.

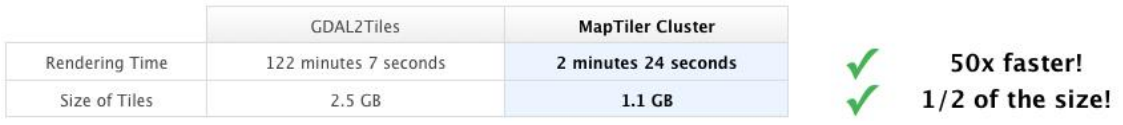
Finally, you just need to use GDAL2Tiles to generate the map tiles:
→ gdal2tiles new_name _4326.tif name_of_folder
There are a lot of options you can use here:

```
gdal2tiles [-p profile] [-r resampling] [-s srs] [-z zoom]
           [-e] [-a nodata] [-v] [-h] [-k] [-n] [-u url]
           [-w webviewer] [-t title] [-c copyright]
           [-g googlekey] [-b bingkey] input_file [output_dir]
```

-s SRS, --s_srs=SRS: The spatial reference system used for the source input data.
-z ZOOM, --zoom=ZOOM: Zoom levels to render (format:'2-5' or '10').
-w WEBVIEWER, --webviewer=WEBVIEWER: Web viewer to generate (all, google, openlayers, none) - default 'all'.
-t TITLE, --title=TITLE: Title of the map.
-c COPYRIGHT, --copyright=COPYRIGHT: Copyright for the map.
-k, -n, -u are options only for KML (Google Earth).
And now in your output_dir would we your new tileset.

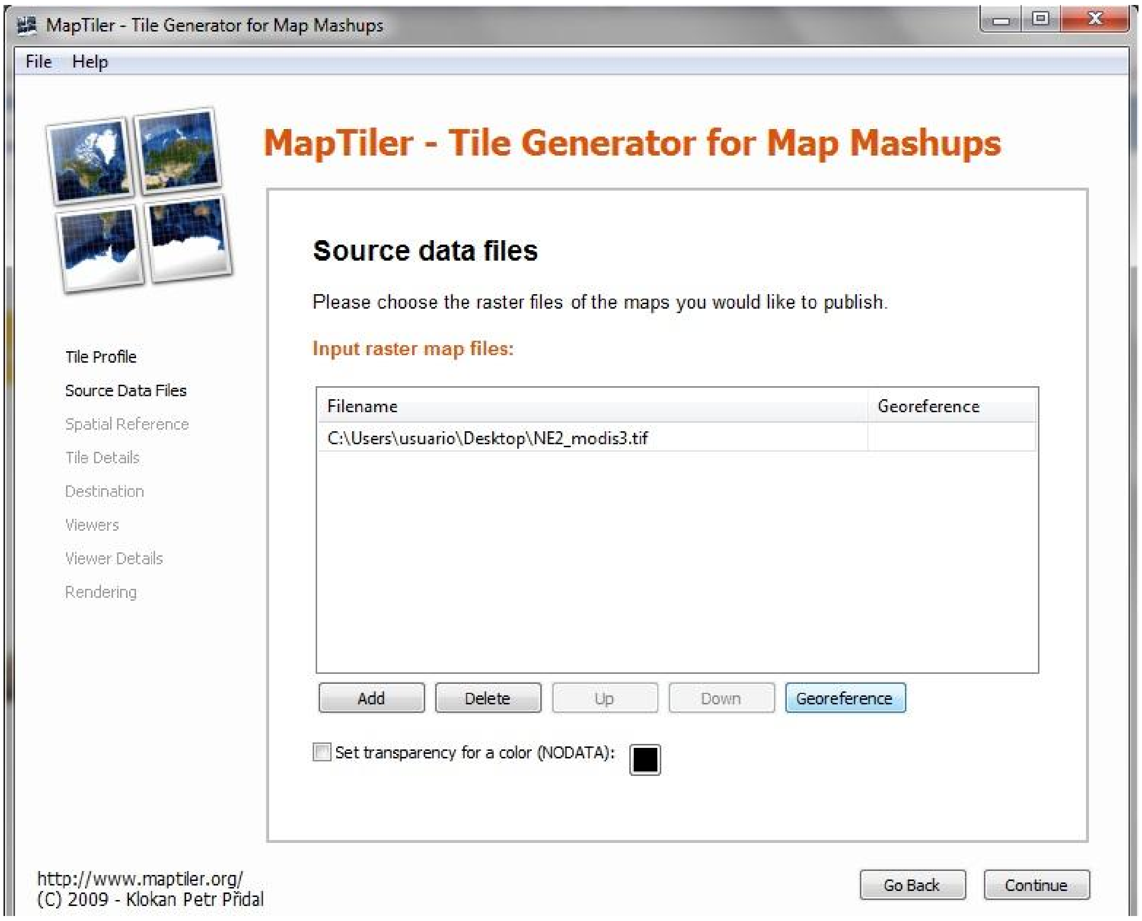
Create tiles of a GeoTIFF map using MapTiler

You can download MapTiler here http://www.maptiler.org/.
MapTiler is a desktop application for the creation of map tiles for rapid raster map publishing. Geodata is transformed to tiles compatible with Google Maps and Earth - ready for publishing via direct upload to any webserver or a cloud storage (such as Amazon S3).
No extensive configuration on the server side is necessary, any simple file hosting is fine. Dynamic interaction such as panning and zooming, overlay of markers and vector data is provided by powerful browser functionality.
The application directly generates a ready to use simple viewer based on OpenLayers and Google Maps API and can be easily customized.
MapTiler is a graphical interface for GDAL2Tiles utility, but much faster and needing less space to store the tiles. As an example to compare: rendering of map (51025x73135 pixels) with raster reprojection on a standard computer with 16 cores against GDAL2Tiles:



	GDAL2Tiles	MapTiler Cluster		
Rendering Time	122 minutes 7 seconds	**2 minutes 24 seconds**	✓	**50x faster!**
Size of Tiles	2.5 GB	**1.1 GB**	✓	**1/2 of the size!**

How to use it:
Firstly, you have to select the Tile profile. Choose Google Maps Compatible (Spherical Mercator) for standard web publishing. Choose Google Earth (KML SuperOverlay) if you also want to generate a KML file for use in Google Earth. Click the “Continue” button.
Then you need to choose the Source Data Files. Browse to select the raster image you want to tile. It is also possible to select a NOData color that will appear as transparent in the resulting image. The NOData color allows you to hide or highlight these values. Usually the NOData color is set to transparent, so that the NOData values are hidden. You have to click “Add” and select the file you want to create tiles for.
Once you have choose you file, you click “Georeference”. Here you can specify the bounding box for your image.



Source data files

Please choose the raster files of the maps you would like to publish.

Input raster map files:

Filename	Georeference
C:\Users\usuario\Desktop\NE2_modis3.tif	

Set transparency for a color (NODATA): ■

http://www.maptiler.org/
(C) 2009 - Klokan Petr Přidal

After that, click the "Continue" button.

Then you need to specify the Spatial Reference System / Coordinate System of the image. Most of the geodata formats (like GeoTIFF) contains the definition of the Coordinate Reference System already - then it is used by MapTiler automatically. In case it is not like this, you can find more information in MapTiler help (http://help.maptiler.org/coordinates/). Click the "Continue" button.

At this point, you can set the minimum and maximum zoom levels, and choose the file format. The default settings for zoom levels and file format are often best. Click the "Continue" button.

Then, you could specify details about the Destination folder and Addresses / URLs for the tileset. If you do not know these, they can be added into the default googlemaps.html and openlayers.html files after tile generation. Click the "Continue" button.

After that, tick the Viewers that should be generated. By default, a googlemaps.html and openlayers.html file are generated. You can also choose to generate a KML SuperOverlay file for Google Earth. Click the "Continue" button.

Finally, you can specify the details for generating the Viewers, such as the title, copyright notice, and API keys. If you do not know these, they can be added into the default googlemaps.html / openlayers.html files after tile generation. Click the "Continue" button.

The last step is to click "Render" to start rendering the image. When complete, MapTiler provides a link to the finished tileset. Open the googlemaps.html or openlayers.html files in a web browser to view the tileset as an overlay.

Publish tiles in a server:
You only need to copy the entire tileset and all subdirectories to a web server.
Then you can edit the googlemaps.html or openlayers.html files as required to present this on the web.

Landez operates and creates MBTiles files

Landez can be found in this repository https://github.com/makinacorpus/landez.
It is programmed in Python and it does not need any dependency, apart from Mapnik in case you want to render tiles locally. To install it, you just need to copy the repository.
To build the MBTiles file it uses mb-util at the end.
Landez permits several different operations related with MBTiles:

- Building MBTiles files: using a remote tile service, a local rendering, another MBTiles file you already have. Or a WMS server.
- Merging tiles together: you can build a new MBTiles file taking at the same time tiles from different sources.
- Exporting image: assemble and arrange the tiles together in a single image.
- Extracting MBTiles contents: you can get a particular tile, the UTFGrid information from a tile, or the metadata store in a MBTile file.
- Post-processing filters: convert map tile to gray scale or replace a specific color by transparent.

Python-mbtiles helps you working with MBTiles databases

This tool can be found in this repository https://github.com/perrygeo/python-mbtiles.
It is also programmed in Python, does not have any external dependency, and to install it you just need to copy the repository.
What you can do with this tool:

- Access to the tiles and their UTFGrid from the MBTiles in an abstract way, just calling the function that it is provided.
- Tile web server that serves the tiles and their UTFGrid data.
- Convert the MBTiles file to png or json files, extracting the information from the database.

MapProxy

Mapproxy is a Python proxy server for geospatial images. It can read data from WMS, tiles, MapServer and Mapnik, and cache and serve that data as WMS,WMTS,TMS and KML. It can also do reprojections between different coordinate reference systems.
You can check here (http://mapproxy.org/docs/latest/install.html) to know how to install MapProxy.

Configuration

MapProxy configuration is set up in the mapproxy.yaml file. The MapProxy configuration is a dictionay, 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. They define how the tiles are organized.
- layers: configure the layers that MapProxy offers. Each layer can consist of multiple sources and caches. We can have several layers, in that case we need to use the transparent option.
- sources: define where MapProxy can retrieve new data, where our tiles are stored. One of the options will be the tiles url.
- caches: here you can configure the internal caches. 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 required data source.
- grids: MapProxy aligns all cached images (tiles) to a grid. Here you can define that grid. Among other options, you can define a factor between each resolution. It should be either a number or the term sqrt2. The default factor is 2.
- globals: here you can define some internals of MapProxy and default values that are used in the other configuration directives.

To start the development server we use the command:

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

Seeding and caching

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. In the file we can find these options.

- Seeds: configure seeding tasks. Here you can create multiple seeding tasks that define what should be seeded. You can specify a list of caches for seeding with caches. If you have specified multiple grids for one cache in your MapProxy configuration, you can select these caches to seed.
- Cleanups: configure cleanup tasks, to remove the old tiles stored. You can clean up caches, grids, levels, coverages, all tiles before a date.
- Coverages: configure coverages for seeding and cleanup tasks. You can define areas where data is available or where the interesting data is.

The tool can seed one or more polygon areas for each cached layer. It can be really useful for lake or sea areas, where the tiles are exactly the same to each others.
Apart from using normal files as cache, you can also use a single SQLite file for the cache. It uses the MBTiles specification(https://github.com/mapbox/mbtiles-spec).

```
caches:
# name of our cache
  mbtiles_cache:
# A list of data sources for this cache. You can use sources defined in  the sources and caches section. MapProxy will merge multiple sources from left (bottom) to right (top) before they are stored on disk.
      sources: [my_source]
```

```
# You can configure one or more grids for each cache. MapProxy will create one cache for each
grid.
      grids: [my_grid]
# Configure the type of the background tile cache.
      cache:
# You configure the type with thetype option. The default type is file
        type: mbtiles
# This is where your .mbtiles file is located
        filename: /path/to/cache.mbtiles
```

You could also set the sources to an empty list, if you use an existing MBTiles file and do not have a source.
You could also 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 do not want to seed the data again.

```
# Each layer contains information about the layer and where the data comes from.
layers:
# The name of the layer. You can omit the name for group layers.
  - name: layer1
# Readable name of the layer, the one that will appear in the website.
    title: Layer using data from existing_cache
# A list of data sources for this layer. You can use sources defined in the sources
and caches section. MapProxy will merge multiple sources.
    sources: [new_cache]

caches:
  new_cache:
    grids: [new_grid]
    sources: [existing_cache]
  existing_cache:
    grids: [old_grid]
    sources: [my_source]

# Here you can define the tile grids that MapProxy uses for the internal caching.
grids:
  utm32n:
# The spatial reference system used for the internal cache, written as EPSG:xxxx.
    srs: 'EPSG:27700'
# The extent of your grid. You can use either a list or a string with the lower left and upper
right coordinates.
    bbox: [4, 46, 16, 56]
# The SRS of the grid bbox.
    bbox_srs: 'EPSG:4326'
```

```
# 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). 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.
    origin: 'nw'
# The resolutions of the first level.
    min_res: 5700
  osm_grid:
# With this option, you can base the grid on the options of another grid you already defined.
    base: GLOBAL_MERCATOR
    origin: nw
```

Grids

By default 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 provide the ability to inherit from another grid. We let our grid inherit from the previously used GLOBAL_GEODETIC grid and add five fixed resolutions to it.

```
grids:
  res_grid:
    base: GLOBAL_GEODETIC
# A list with all resolutions that MapProxy should cache.
    res: [1, 0.5, 0.25, 0.125, 0.0625]
```

The resolutions are always in the unit of the SRS, in this case in degree per pixel.
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.

Let 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 (European Petroleum Survey Group).

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

In bbox we consider: lat_min,lon_min,lat_max,lon_max; which means minimal latitude is 6, minimal longitude is 47.3, maximal latitude is 15.1, maximal longitude is 55.

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 andtile_size.
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. For example, with resolutions from 1000 to 10 and 6 levels you would get 1000, 398, 158, 63, 25, 10 (rounded here).
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.
You can set every cache resolution in the res option of a layer.

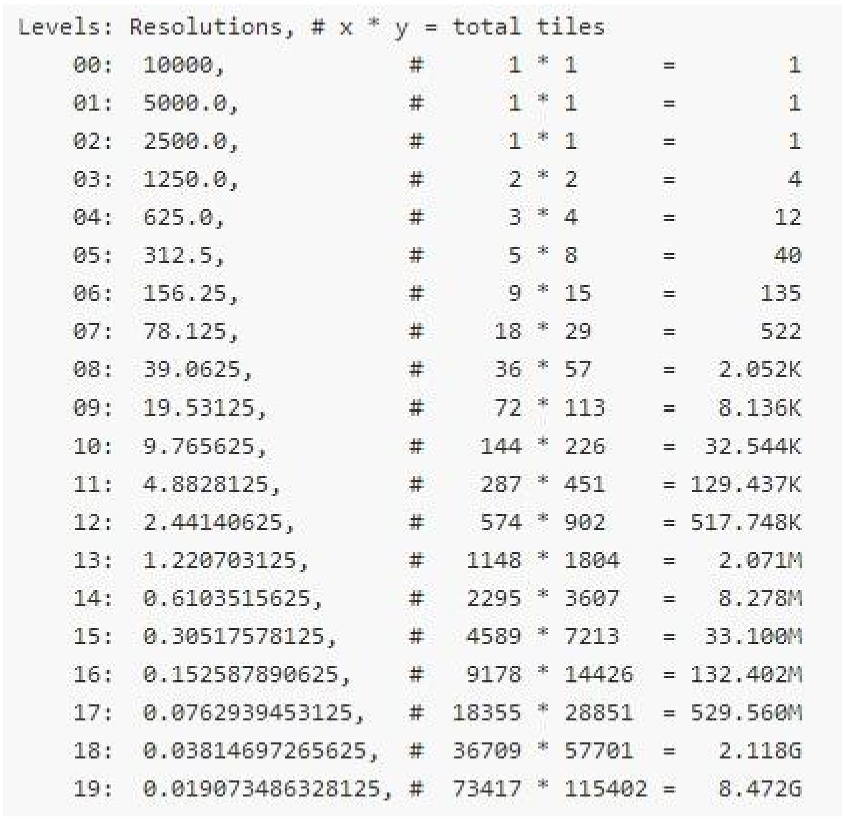
```
caches:
  custom_res_cache:
    grids: [custom_res]
    sources: [vector_source]

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, but you can set smaller values:

```
grids:
  custom_factor:
# Here you can define a factor between each resolution. It should be either a number or the term sqrt2.
    res_factor: 1.6
```

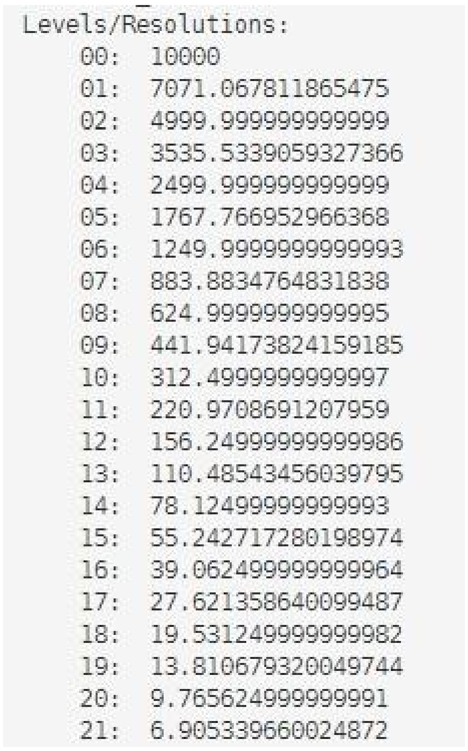
Other option is a convenient variation of the previous option. A factor of 1.41421, the square root of two, would get resolutions of 10, 7.07, 5, 3.54, 2.5,…. Notice that every second resolution is identical to the power-of-two resolutions. This comes in handy if you use the layer not only in classic 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
```



```
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
```

In this example, we have only chosen min_res: 10000; and the res_factor will be 2, which is the default value. And bbox: [5, 50, 10, 55]



```
Levels/Resolutions:
    00:  10000
    01:  7071.067811865475
    02:  4999.999999999999
    03:  3535.5339059327366
    04:  2499.999999999999
    05:  1767.766952966368
    06:  1249.9999999999993
    07:  883.8834764831838
    08:  624.9999999999995
    09:  441.94173824159185
    10:  312.4999999999997
    11:  220.9708691207959
    12:  156.24999999999986
    13:  110.48543456039795
    14:  78.12499999999993
    15:  55.242717280198974
    16:  39.062499999999964
    17:  27.621358640099487
    18:  19.531249999999982
    19:  13.810679320049744
    20:  9.765624999999991
    21:  6.905339660024872
```

Here we select min_res: 10000 and res_factor: 'sqrt2'

Reprojecting

When the grids are not compatible, for example, 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.
It will reproject data if it needs to get data from this layer in any other SRS.
If MapProxy needs to reproject and the source has multiple supported_srs, then it will use the fist projected SRS for requests in projected SRS, or the fist geographic SRS for requests in geographic SRS. For example, when supported_srs is ['EPSG:4326','EPSG:31467'] caches with EPSG:900913 will use EPSG:32467.
Here is an example that uses OSM tiles as a source and offers them in UTM projection. The disable_storageoption prevents MapProxy from building up two caches. The meta_size makes MapProxy to reproject multiple tiles at once.
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]

# MapProxy does not make a single request for every tile but 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 WMS.
    meta_size: [4, 4]
    sources: [osm_cache_in]

  osm_cache_in:
    grids: [osm_grid]
# 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.
    disable_storage: true
    sources: [osm_source]

# 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.
sources:
  osm_source:
# Use the type tile to request data from from existing tile servers like TileCache and GeoWebCache.
    type: tile
```

```
# The grid of the tile source. Defaults to GLOBAL_MERCATOR, a grid that is compatible with
popular web mapping applications.
    grid: osm_grid
# This source takes a url option that contains a URL template. The template format
is %(key_name)s.
    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

  osm_grid:
    base: GLOBAL_MERCATOR
    origin: nw
```

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:
# Use the type wms to for WMS servers.
  type: wms
# 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.
  supported_srs: ['EPSG:4326', 'EPSG:25832']
# 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://wms.example.org/service?
   layers: layer0, layer1
```

MapCache

MapCache is a server that implements tile caching to speed up access to wms layers.
It can be found in this repository https://github.com/mapserver/mapcache/.
It serves tiles using a variety of different request protocols: TMS, KML, OGC WMTS, OGC WMS, GoogleMaps xyz, Virtual Earth Tile.
To enable a particular service you should add this in your service configuration, where type indicate the protocol.
<service type="tms" enabled="true"/>
There are several types of cache you could use:

- Disk cache: the easiest to configure and the fastest to access to tiles. It is the best option when caching small maps, but it could cause problems when storing millions of tiles, because of the creation of so many directories. You could use the default structure or create your template to indicate how you want your tiles to store.
- Sqlite cache: they store the tiles as blobs inside a single database file, so the number of tiles does not affect. There is also a default schema or you could use the MBTiles schema.
- GeoTIFF cache: storing the tiles in a tiff file. This cache should be considered read-only, because although written is possible, there may be file corruption troubles. Until all the tiles have been cached, the tiff file is missing a number of jpeg tiles.

A configuration example:

```
<mapcache>
  <!-- A grid is the matrix that maps tiles on an area, and consists of a spatial reference,
   a geographic extent, resolutions, and tile size. -->
  <grid name=”grid_4326”>
    <!-- number of pixels indicating the width and height of the tiles -->
    <size>256 256</size>
    <!--  the bounds covered by the grid -->
    <extent>-180 -90 180 90</extent>
    <!-- the spatial reference system of the grid, it is used to look which grid to use when
     getting a WMS request. You must make sure that the source that is queried is capable of
    returning image data for this srs -->
    <srs>epsg:4326</srs>
    <!-- the units used by the grid’s projection, it could be meters, decimal degrees or feet  -->
    <units>dd</units>
     <!—the resolutions for every zoom level included in the grid. The biggest value will
    correspond to the grid’s zoom level 0. Resolutions are expressed in “units-per-pixel” -->
     <resolutions>0.703125000000000 0.351562500000000 0.175781250000000
      8.78906250000000e-2 4.39453125000000e-2</resolutions>
  </grid>

  <!- - this is the service that will obtain the image data. Right now only the WMS service is
   available -->
  <source name="swiss_map" type="wms">
  <!-- this are the parameters that will be added to the GetMap request, you can specify any
   parameter here, e.g. FORMAT to indicate the kind of tiles.
   request   -->
  <getmap>
    <params>
      <!—this parameter is mandatory -->
      <LAYERS>basic</LAYERS>
      <FORMAT>image/png</FORMAT>
      <!-- these parameter are used by default and can be override -->
     <REQUEST>GetMap</REQUEST>
     <SERVICE>WMS</SERVICE>
      <STYLES></STYLES>
      <VERSION>1.1.0</VERSION>
    </params>
  </getmap>
  <!-- http url and parameters that will be used when making WMS requests -->
  <http>
    <!-- url of the wms service, without any parameters -->
    <url>http:// 152.96.56.43/wms/swiss_map</url>
    <!-- http headers added here will take precedence over any default headers added to the
    request. Typical headers that can be added here are User-Agent and Referer. -->
   <headers>
     <User-Agent>mod-mapcache</User-Agent>
   </headers>
   <!-- timeout in seconds before bailing out from a request -->
   <connection_timeout>60</connection_timeout>
  </http>
</source>

 <!--here will be configured the directory where the tiles will be stored. The cache could be a
    sqlite database and also a MBTiles file; that would be specified as the type -->
<cache name="maps" type="disk">
  <!-- directory where the tile structure will be stored. It needs to be readable and writable by
    the user -->
<base>/mapserver/cache</base>
  <!—when this appears it indicates that blank tiles will be detected at creation time and
     linked to a single blank tile on disk not to use disk space -->
  <symlink_blank/>
  <!—you can use a template to indicate how to map a tile (by tileset, grid name, dimension,
    format, x, y, and z) to a filename on the filesystem.
    the following replacements are performed:
    - {tileset} : the tileset name
    - {grid} : the grid name
    - {dim} : a string that concatenates the tile's dimension
    - {ext} : the filename extension for the tile's image format
    - {x},{y},{z} : the tile x,y,z values
    - {inv_x}, {inv_y}, {inv_z} : inverted x,y,z values (inv_x = level->maxx - x - 1). This
       is mainly used to support grids where one axis is inverted (e.g. the google schema)
       and you want to create on offline cache  -->
  <template>/tmp/template-test/{tileset}#{grid}#{dim}/{z}/{x}/{y}.{ext}</template>
</cache>

<!-- the image format used in the tiles: PNG, JPEG or both mixed -->
<format name="particular_png" type ="PNG">
   <!--  png compression: best or fast
     "best" compression is cpu intensive for little gain over the default compression, that is
     obtained by leving out this tag  -->
  <compression>fast</compression>
  <!-- if supplied, this enables png quantization which reduces the number of colors in an
  image to atain higher compression.  the number of colors can be between 2 and 256 -->
  <colors>256</colors>
</format>

<!-- the set of tiles coming from a source, stored in a cache, and returned to the client in a
   given format -->
<tileset>
   <!-- the "name" attribute of a preconfigured <source> -->
   <source>swiss_map</source>
   <!-- the "name" attribute of a preconfigured <cache> -->
   <cache>map</cache>
   <!-- the "name" attribute of a preconfigured <grid>   -->
  <grid>grid_4326</grid>
  <!-- format to use when storing a tile. this should be a format with high
   compression, eg. png with compression "best", as the compression operation is only
   done once at creation time  -->
  <format>PNG</format>
  <!--number of columns and rows to use for metatiling, see
    http://geowebcache.org/docs/current/concepts/metatiles.html   -->
  <metatile>5 5</metatile>
  <!-- area around the tile or metatile that will be cut off to prevent some edge artifacts   -->
  <metabuffer>10</metabuffer>
  <!-- this is expressed in a number of seconds   after the creation date of the tile that the tile
  should expire   -->
  <expires>3600</expires>
</tileset>

 <!-- indicate the type of request that it has to respond to; the options are: wms, wmts, tms,
 kml, gmaps, ve, demo -->
<service type="wms" enabled="true">
   <!-- configure response to wms requests that are not aligned to a tileset's grids.
```

```
      responding to requests that are not in the SRS of a configured grid is not supported, but
      this should never happen as only the supported SRSs are in the capabilities   document.
      allowed values are:
        - error: return a 404 error (default)
        - assemble: build the full image by assembling the tiles from the cache
        - forward: forward the request to the configured source  -->
    <full_wms>assemble</full_wms>
    <!--  filter applied when resampling tiles for full wms requests. It could be nearest  (fastest
     but poor quality) or bilinear (slower with higher quality)   -->
    <resample_mode>bilinear</resample_mode>
    <!-- image format to use when assembling tiles  -->
    <format>myjpeg</format>
  </services>
</mapcache>
```

2.3. Further development

The next step in the word of maps consists in changing tiles for vector tiles. In this case, instead of tiling raster images, vector data is tiling into tiles. They would be used as normal tiles we used before, but now we can attach JavaScript behavior to the vectors. So when the map is zoomed and panned, the JavaScript code fetches the tiled data on demand and they will display the corresponding shapes.
MapBox has developed an open source vector format to power the future of our web maps. Vector tiles rethink web maps from the ground up, providing a single efficient format to power raster tiles, interactive features, GeoJSON streams, mobile renderers, and much more.
Vector tiles are the idea for making efficient maps. A lot of geographic data is in fact vector oriented, like lines and polygons. Vector tiles are quite simple. Only instead of serving a png image the URL serves a GeoJSON file describing the geometry inside the tile's bounding box.

A vector tile can consist of one or more layers which are identified by its name and that contain one or more features. A feature can contain attributes and geometries: like a point, a linestring, or a polygon. Features expect single geometries, that is why multipolygons or multilinestrings are represented as multiple features, each storing a single geometry part. Geometries are stored indicating its x, y position.
Feature attributes are encoded as key:value pairs which are dictionaries inside the layer level for compact storage of any repeated keys or values. Values use variant type encoding supporting both unicode strings, boolean values, and various integer and floating point types.
Many map servers render vector data into raster images that are then served to clients. But serving the vector data directly to the user's browser for rendering on the client can make maps that are more flexible and more efficient.

One tricky thing about vector tiles is what to do about features that cross tiles. There are three possibilities: you can cut the geometry to the tile boundary and rely on the overlapping lines being drawn to make a seamless map. It is also possible not to cut, which results in redundant data but keeps features intact. A third option can be to cut geometry and re-unify it on the client before rendering.

Vector tiles have the advantages of image tiles, and also add the flexibility of source vectors. But mainly it tries to solve the problem of the size. Complex geometries can need a huge amount of space, which means too much time to load all the data. Even considering that the request are only done for the tiles that are visible on the screen and some around them.

3. Review

3.1. Criteria

A lot of people who are not in the technological world think that computer science is complicated to understand, normally, because it is in another language. And it is true, it is written in a programming language. So in order that everyone could understand, I tried to make an effort in making enough comments of my code and in explaining things in detail.
There are a lot of programs with really good documentation. But there are also others great tools with no documentation at all. And I think that is a mistake, because if we do something good, we should also try to make it accessible for most people.

The first part of this project, how to create interactive maps, is focus in help anyone who wants to create a map, no matter if he knows or not about programming. The tutorials start from the beginning, so everyone could follow. TileMill is an easy to use and well documented develop environment. Maybe the most complicated part is how to apply the style using CartoCSS to the map, because it is really powerful, but for a user with no programming knowledge it might be difficult to get the exact result he is waiting for.
Also the templates for clients, in Leaflet or OpenLayers, are enough commented so the user can know exactly what is doing every line of code. The templates are simple, with only the necessary information on them. If the user want to do any change, like remove the zoom slider or the scale in the map, he will only have to delete the requires lines of code. In the opposite case, if he wants to add some new functionality, he will need to look into the tutorial of the corresponding library. The important part, where the reference to the user map is done, it is clearly specified so he knows how the path for its map has to be.

The second part, the Python code to server our local maps on the internet, is also to be used for any kind of user. But the difference is that this is only to be used, not modified. But still, I have commented it enough so in case somebody need to make something different, he could modify it, instead of start its own code for the beginning.
Tiny Tile Server meets the objectives regarding displaying a map with UTFGrid information. But it can be extended, for example, by supporting more access protocols to the tiles. The server is well prepared for the change, so no modifications would be need in the existing code. Just adding the necessary code for the new protocol.

3.2. Conclusions

Creation of interactive maps is a current issue. There are so many possibilities and so many interesting ideas, that anyone could have a brilliant thought and wanted to carry it out. So it is necessary that the people who develop tools to create maps keep in mind all the potential users, and not only the ones related with computer science.

A common problem is that most of the projects are focused in provide a specific functionality and be integrated with other tools to solve the whole problem. That is good, and the normal way of acting, but at the end the user ends up with a lot of references and trying to get everything together to work fine.

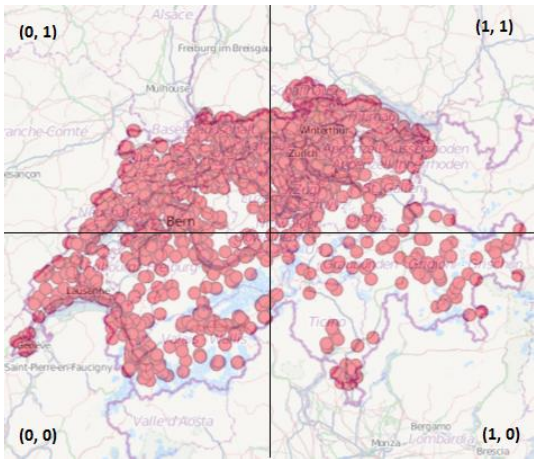
In this case, the website is completely, and all the necessary tools are already integrated or are referenced. The necessary change, of course, is for the user to add his maps and modify the path where it is called for the name of its map.

4. Implementation plan

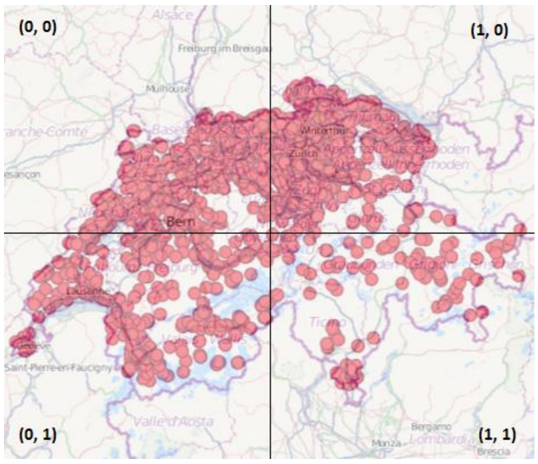
4.1. Rough outline of my own solution concept

Tiny Tile Server permits to extract a map from a MBTiles database using two distinct protocols: direct XYZ access to the tiles or Web Map Tile Service. These protocols do not consider the same origin for the tiles because they use different access to them.

XYZ origin is in bottom left.



WMTS origin is in top left.



But apart from that, there is no other difference when accessing tiles. So we can use the same function in both cases, just checking if it is a WMTS access to transform the y coordinate value. The access to the database consists in select the tile for the corresponding zoom level, tile column and tile row.

Regarding the grids extraction, this is the same in both cases, because it depends on the UTFGrid specification, which states that the origin is on top left corner. In this case, the access is more complex, because there is not one table containing the whole information, this is distributed. And the tricky part is that the information is in JSON format, so before we can

display it, or just use it, we need to convert it.
To generate the whole Web Map Tile Service functionality, it is also needed to provide the XML file with the getCapabilities information. In order to do that we need to extract the information from the metadata table in the database, and we need to transform the coordinates presented in latitude longitude in WGS84 ellipsoid to Mercator Map projection on the sphere.
The Global Mercator class has been developed by Klokan Petr Pridal and can be found here http://www.maptiler.org/google-maps-coordinates-tile-bounds-projection/

4.2. Restrictions to consider

Currently there are some restrictions when using this server.
This server is prepared to extract the map from a MBTiles database, and when using that we have to consider:

- All tiles must be 256x256 pixels. UTFGrid uses JSON as a container format, and it is exclusively oriented towards square 256x256 pixel tiles.
- Only the Spherical Mercator projection is supported for presentation, when displaying the tiles.
- Only latitude-longitude coordinates are supported for metadata, such as bounds and centers.

Tiny Tile Server is configured to work with maps with only one layer.

It is also important to remember:

- MBTiles database expects origin in bottom left corner when extracting tiles.
- Web Map Tile Service expects origin in top left corner when extracting tiles.
- In any case, the UTFGrid origin would be in top left corner.

5. Results, evaluation and outlook

5.1. Achievement of objectives

Not all the initial objectives have been accomplished. Tiny Tile Server was supposed to serve tiles, UTFGrid and metadata from a MBTiles database. But the UTFGrid part, displaying textual information needed for infobox, does not work completely.

- When UTFGrid is displayed:
    1. At the beginning when the map is loaded.
    2. After zoom-in or zoom-out.
    3. After doing double click in a point to zoom-in.
    4. After doing shift + select a zone to show it.
    5. After zoom by using the mouse wheel.
    6. After reloading the page.
    7. Writing in the URL the new location, like /exampleLLlocal#5/51.124/39.155
- When UTFGrid does not work:
    1. After doing single click and dragging the map to a new position.

The point is that it also does not work when using Wax with a map hosted in MapBox, so it is clear that Tiny Tile Server is working fine. It could be that the client part with Leaflet or OpenLayers is not well done, but observing this example http://www.mapbox.com/wax/interaction-leaf.html, that also does not work like it is supposed to, makes me think that the problem could be inside Wax and not in my templates.

5.2. Outlook: further development

There are two possibilities that would be interesting to develop in the future, because they are related with Tiny Tile Server, so it would mean to add it more functionality, and because there are not too many options already developed.

The first is to generate Wax functionality regarding how to display infoboxes with the UTFGrid information on a map. Tiny Tile Server can extract the UTFGrid information and it can be display on a file, but not as a part of the map. Recreating all the Wax functionalities would be too complex and that is not the idea. Just the part regarding how to extract the value of the data associated to a pixel. There are several possibilities around how and where to show the value. One easy option is to select a position, for example In the top right corner, and write the value there.

Another possible development is to support more protocols to serve maps apart from direct access in XYZ and Web Map Tile Service. Two possibilities are Tile Map Service and Web Map Service.

Software project documentation

1. Vision

Once we have created our map, we want to display it on the internet. And in this case we are going to use the Python code in order to do that. And what do we need?

- Of course, we need to have a map in a MBTiles file.
- Because of the server code is written in Python, we have to install Python.
- Apart from Python, we also need to install Bottle. Bottle is a fast, simple and lightweight WSGI micro web-framework for Python. It is distributed as a single file module and has no dependencies other than the Python Standard Library.

And now we have already finished installing things. The next step will be to adapt the templates to our case. I am also using Bootstrap in my website, but that is not necessary, it is just to make a good and clear division of the code.

- We have to change the URL where we are calling the map in the clients(Leaflet or OpenLayers), to adapt it to the path where our map is located.
- We have to change the config_url and the title of our website.
- Optionally, we may want to change the HTML or some extra characteristics of the map.

MBTiles database

Tiles in MBTiles are assumed to be in Spherical Mercator as Coordinates Reference System. This projection treats the Earth as a sphere, instead of as a ellipsoid, like the World Geodetic System does.
The database consists in 9 tables: grid_data, grid_key, grid_utfgrid, grids, images, keymap, map, metadata and tiles. Some of them are views: virtual tables as a result from a select statement from other tables, normally doing a join.

map this table contains five columns: zoom_level, which is the zoom number in which a particular tile or grid would be displayed; tile_column, which is the x coordinate where the tile would displayed on the map; tile_row, which is the y coordinate where the tile would displayed on the map; tile_id, which is the id of the tile that would be displayed and it is stored in the **images** table; grid_id, which is the id of the grid that would be displayed and it is stored in the **grid_utfgrid** table. This table has a unique index generated by a zoom_level, a tile_column and a tile_row, we need the three of them in order to get only one tile or one grid.

metadata this table contains two columns: name and value. It is used two make a formal definition about the map, specified as a key/value pair. The user can decide which keys wants to define, what there are five that are required: name, whose value is the name of the tileset; type, whose value can be “baselayer” or “overlay” (it is displayed on top of another layer); version, whose value is the version of the tileset; description, whose value is a descript about the map; format, whose value is “png” or “jpg” and indicates the file format of the tiles.
Some others possible keys are: bounds, whose value is the maximum extent of the rendered map area. Bounds are represented in WGS:84, latitude and longitude values, in the OpenLayers Bounds format, left, bottom, right, top. Example of the full earth: -180.0,-

85,180,85; center; whose value is the map center at the beginning; maxzoom, whose value is the maximum zoom the map permits; minzoom, whose value is the minimum zoom the map permits; tiles, whose value is the URL where individual tiles can be accessed; grids, whose value is the URL where individual grids can be accessed; legend, whose value is the map legend; formatter, whose value is the JavaScript function for formatting feature data from UTFGrid JSON, this is required in case our map need to support interactivity.
When we export our map to MBTiles in TiliMill the metadata table does not contain the name "tiles" neither "grids", because they are relatives to the directory where the MBTiles would be located. That is why the database cannot be used to extract the tiles or grids directly if we do not modify it. In order to do that, we need to execute these SQL sentences:

```
>sqlite3 example.mbtiles "insert into metadata values ('tiles', 'http://url-specific-in-the-application /example/{z}/{x}/{y}.png'')"
>sqlite3 example.mbtiles "insert into metadata values ('grids', 'http://url-specific-in-the-application /example/{z}/{x}/{y}.grid.json'')"
```

example, in all the cases, is the name of our MBTiles database.

images this table contains two columns: tile_data, which is a individual image that would be one map tile; tile_id, which is the image identifier.

tiles this view is generated for a join between **map** and **images**. It contains four columns: zoom_level, tile_column and tile_row are exactly the same as explain in **map** table; tile_data is the image stored in **images** table, which is obtained using the tile_id being the same in **map** and **images** tables.

grids this view is generated for a join between **map** and **grids**. It contains four columns: zoom_level, tile_column and tile_row are exactly the same as explain in **map** table; grid is the grid_utfgrid stored in **grid_utfgrid** table, which is obtained using the grid_id being the same in **map** and **grid_utfgrid** tables. It is not the whole information in the .grid.json file, only the corresponding with "grid" and "key". The related with "data" containing the key_name and key_json are acquired from **keymap**

keymap this table contains two columns: key_name, which is a number that represent the identifier; key_json, which is the UTFGrid information in json format. The key_json contains one or more key/value pairs, as many as the number of different UTFGrid we want to display. The key indicate the feature we are displaying, the value indicate what is that feature in every case.
As an example we can consider a world map where we want to display the name of the country, the language speaking there and its flag when the user hover in a specific country. In this case we have three pairs, and the keys could be "Name", "Language" and "Flag". For every country we would have a specific value for the keys, these can be the same for different keys, like in this example "Language": "English" would appear in a lot of countries. In this example, if we consider Switzerland we would obtain { "Name": "Switzerland", "Language": "German", "Flag": text_containing_the_image_reference }

```
>sqlite3 example.mbtiles "select * from keymap where key_name = "1142""
1142|{"Name":"Negro"}
```

grid_data this view is generated for a join between **map**, **grid_key** and **keymap**. It contains five columns: zoom_level, tile_column, tile_row are exactly the same as explain in **map** table; key_name is again a number that represent the json identifier; key_json is again the key_json contained in **keymap** table.

```
>sqlite3 example.mbtiles "select * from grid_data where zoom_level=5 and tile_column=10 and tile_row=15"
5|10|15|1142|{"Name":"Negro"}
5|10|15|1280|{"Name":"Guapor"}
5|10|15|1291|{"Name":"Madeira"}
5|10|15|463|{"Name":"Amazonas"}
```

grid_key this table contains two columns: grid_id, which is the identifier used to represent a grid; key_name, which is a number that represents the identifier.

grid_utfgrid this table contains two columns: grid_id, which is the identifier used to represent a grid; grid_utfgrid, which is value for the "grid" as it is shown in the .grid.json file.

More information can be found in the MBTiles specification https://github.com/mapbox/mbtiles-spec and in the UTFGrid specification https://github.com/mapbox/utfgrid-spec/.

2. Analysis and requirements specification

2.1. Functional requirements

Functional requirements describe the internal behave of the software: calculations, technical details, data manipulations…

The main objective of the Tiny Tile Server is to display a map on a website. Here we have to consider some sub objectives:

- Display the tiles. This is the main and first part, because consists in showing the images that are put together to form the map.
- Display the UTFGrid. This is the main part when adding interactivity, and consists in showing infobox with information about specific data related to every individual point in the map that have some information attached to it. This is optional in a map, not all the maps needs to provide it.
- Display the metadata. This is particular important in the Web Map Tile Service, because it is used to be shown in the Capabilities XML file.
- Add extra features. There are some functionalities that the user normally expects in a map: zoom slider, scale or a legend.

The following protocols are provided:

- Direct access with XYZ tile requests to existing tiles in a MBTiles file.
- Web Map Tile Service implementation standard created by Open Geospatial Consortium.
- TileJSON for metadata structure.

2.2. Use cases

A use case describes a way in which a real-world actor interacts with the system. What you can do with this application, how to use it: which steps you have to follow in order to get the result you are waiting for, and which are the possible errors you could find in case you make a mistake.

Previously, you should have a map in a MBTiles file. In case you want to create your own map using TileMill, you could follow this tutorial. If you already know how to use TileMill and you want to create something more advanced, you could follow this tutorial.

Actor	**Description**
User	Person who wants to show a map on a website

Use Case	Show simple map on the internet

Dependencies	None
Description	The system should behaved as described in the following use case when an user wants to show a previously created map in a MBTiles file on a website.
Precondition	• User needs to have a map in a MBTiles file. • User needs to have Python and Bottle installed in the website.
Ordinary sequence	1. User adds a tile which url will be the path to the local MBTiles file. 2. User adds the code to create a map, which div attribute has to exist in the html file. 3. If User wants to add optional controls to the map, the use case Add control is performed.
Postcondition	• User`s map will be accessible from your website.
Exceptions	1. If the layer is wrong, the system will show the Error: 500 Internal Server Error, then this use case end. 1. If the url is wrong, the system will show the Error: 404 Not found, then this use case end. 2. If the div is wrong or there is no div, the system will not show the map, then this use case end.

Use Case	Show map with UTFGrid interaction on the internet
Dependencies	
Description	The system should behaved as described in the following use case when an user wants to show a previously created map in a MBTiles file on a website.
Precondition	• User needs to have a map in a MBTiles file. • User needs to have Python and Bottle installed in the website.
Ordinary sequence	1. User adds a tile which url will be the path to the local MBTiles file. 2. User adds a tilejson object which url be the path to the local MBTiles file. 3. User adds the code to create a map. 4. User adds a control to show the UTFGrid information on the map. 5. If User wants to add optional controls to the map, the use case Add control is performed
Postcondition	• User`s map will be accessible from your website
Exceptions	1. If the layer is wrong, the system will show the Error: 500 Internal Server Error, then this use case end. **1.** If the url is wrong, the system will show the Error: 404 Not found, then this use case end.

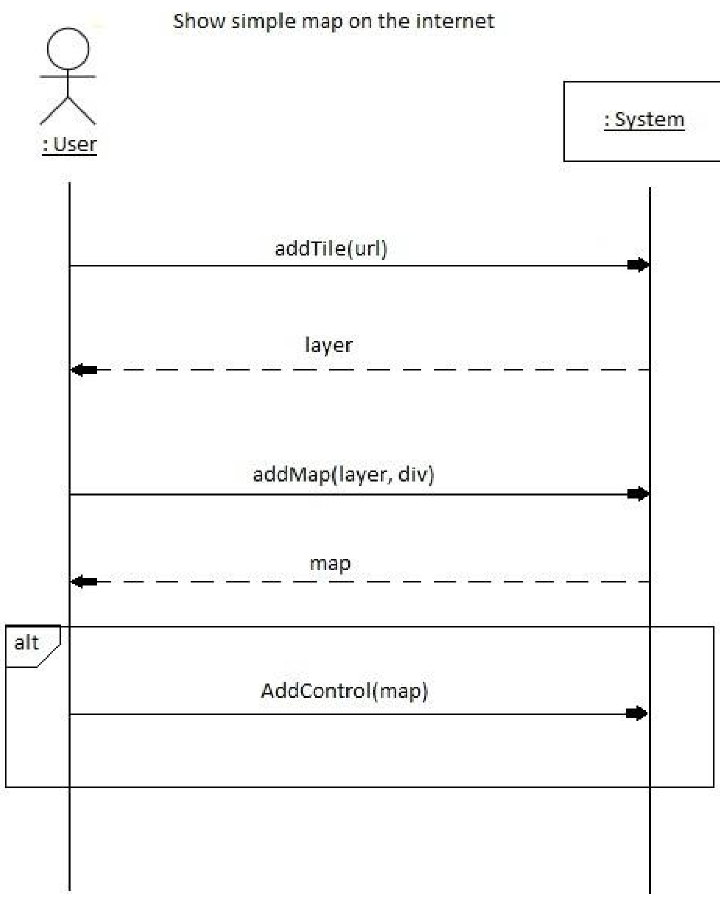
2. If the url is wrong, the system will show the Error: 404 Not found, then this use case end.
3. If the div is wrong or there is no div, the system will not show the map, then this use case end.

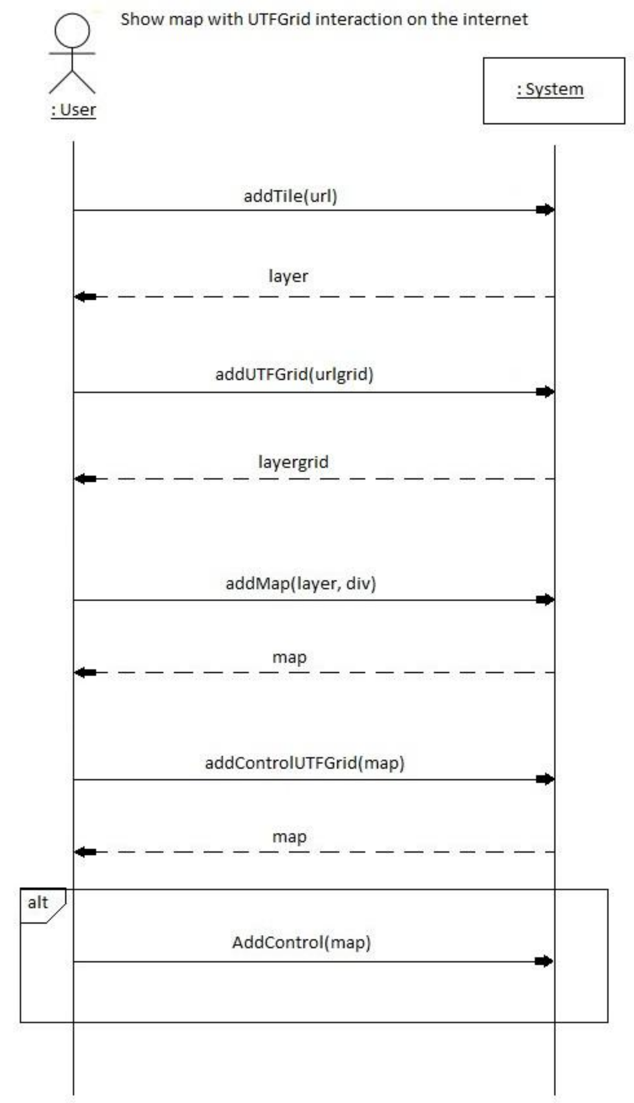
Use Case	Add control
Dependencies	Show simple map on the internet and Show map with UTFGrid interactionon the internet
Description	The system shall behave as described in the following abstract use case during the performance of the following use cases: Show simple map on the internet or Show map with UTFGrid interactionon the internet.
Precondition	None
Ordinary sequence	1. User adds a specific control: • LayerSwitcher • ZoomBar • ScaleLine • Navigation
Postcondition	• Your map will have the chosen controls.
Exceptions	

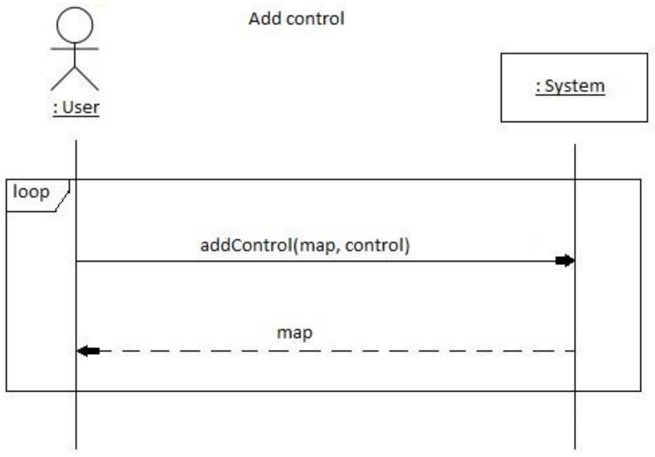
2.3. System sequence diagrams

A system sequence diagrams (SDD) is a sequence of iterations which determine system events and operations, and identify system functionality (operations) to be designed. In a system sequence diagram will be showed the events generated by an actor, in which order they occurred and possible interaction with the system for a particular scenario of an use case

Here is showed the positive scenario of every of the use cases previously described. A positive scenario is when no exceptions occurred and the user gain the postcondition.


Show simple map on the internet
: User
: System
addTile(url)
layer
addMap(layer, div)
map
alt
AddControl(map)


Show map with UTFGrid interaction on the internet
: User
: System
addTile(url)
layer
addUTFGrid(urlgrid)
layergrid
addMap(layer, div)
map
addControlUTFGrid(map)
map
alt
AddControl(map)


Add control
: User
: System
loop
addControl(map, control)
map

2.4. Non-functional requirements

A non-functional requirement or quality attribute is a criteria that can be analyzed in order to decide if the system operations are appropriate. They can be divided into two main categories: execution qualities, such as security and usability, which are observable at run time; evolution qualities, such as testability, maintainability, extensibility and scalability, which are embodied in the static structure of the software system.

- Functionality: Tiny Tile Server provides the user well explained options, indicating which operations can be performed. The different options are presented to the user who can choose which one to carry out. There are tutorials in the website and the Tiny Tile Server has a well documented code.
- Easy to use: because of the kind of structure of the application, all the calls to internal functions are listed in the same file (api.py), so when an user wants to utilize Tiny Tile Server just need to look at that file to know how to make the calls. Templates are also provided, what make even easier to use the application. The user interface of the website is also easy to use and intuitive.
- Maintainability: the different modules in the website are completely independent, they can be replaced or tested without other dependences. Tiny Tile Server has the minimum dependences necessary in each classes. And that is because of same methods that can be used in different classes.
- Portability: the website has been tested in different operating systems: Macintosh, Windows and Linux; and different browsers: Google Chrome, Mozilla Firefox, Safari and Internet Explorer. Tiny Tile Server has no dependences apart from Python, and it is easy to install, you just need to put all the files together and reference the directory in the files where you want to use the functionality.

3. Design

3.1. Architecture

I have developed Tiny Tile Server, which is a server in Python to serve local maps in MBTiles format to a website. But that is only a part of this project. There are also templates to use OpenLayers or Leaflet as a client to display the maps. And maps created with TileMill as examples. And the website in general.

The website has been developed using Bootstrap, which is a Framework developed by Twitter design to simplify the process of web design. It is offer some CSS by default and some JavaScript files that provide us a basic layout that we can change later.

- Theses interfaces work perfectly well in the new browser and well enough in the older ones.
- The design can be visualized in different devices, using different scales or resolutions.
- There is a better integration with the libraries that are used normally.
- It is a good design based in current tools and standards, like CSS3 and HTML5.

We have a HTML file that will act as layout for the whole website. In it, we will define the header, the footer, the navigation bar… And also we will include all the CCS and JavaScript common for all the files. We cannot forget to include the CCS necessary to use Bootstrap. For every single html file we will reuse the layout and we will add their specific CSS and JavaScript.
A really good advantage is the use of Grid system, to divide the screen in a certain number of columns. In this case, it allows 12 columns by default which are configurable. They can be static or fluid. The columns can be combined to generate tables or be a different number depending on the device using to visualize the website.
It is Object Oriented CSS: everything is organized by modules, which are independent and they can be reutilize in each project. Classes are used to name the objects, so you can apply the CSS to them instead of the HTML directly. The visual effects are defined separately, so you can reuse them in different projects, or use several together. The objects have to look the same independent on where they are located, you have to use classes to describe them.
The templates for the client side are done in OpenLayers or in Leaflet, which are open source JavaScript libraries to create interactive maps.

The website is programmed is Python, a script language, that can be used for lineal, structured or object oriented programming, which is the most used. Python is multi-platform and can be utilize to develop web or desktop applications.
The server uses WSGI: mod_wsgi implements a simple to use Apache module which can host any Python application which supports the Python WSGI interface, it permits us to attach our application to an Apache server. In order for that to work, we will need to specify in our Apache configuration that we are using a VirtualHost that enables WSGIDaemonProcess and WSGIScriptAlias. And we must indicate the WSGIProcessGroup and WSGIApplicationGroup. As mod_wsgi supports the WSGI interface specification, any Python web framework or

application which is compatible with the WSGI interface specification should be able to be hosted on top of mod_wsgi.

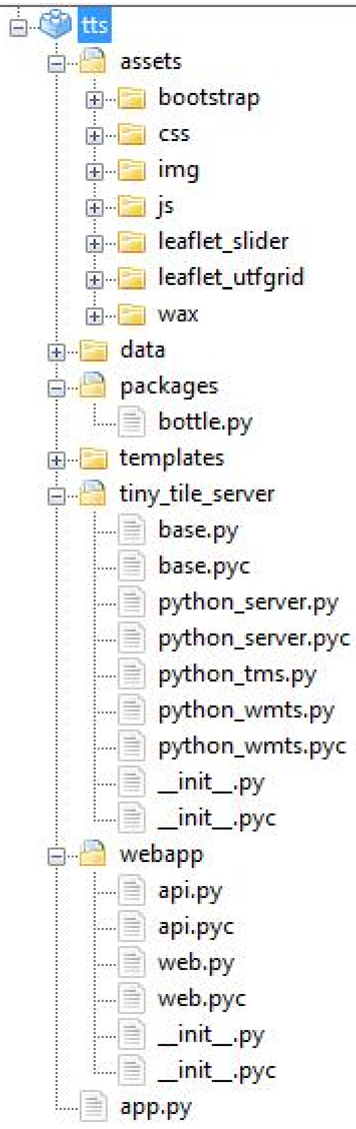
Along with Python is used Bottle, a simple web framework for web development. With Bottle are handled most of the functionalities related with the server.
The entry point to start the application is a .py file which load the main application and the other modules, o sub-applications. And it must contain the Bottle run() function, that when it is called without parameters, starts a local development server on port 8080. If you run it by command line, it will inform you that you can access your application in http://localhost:8080/.
There are some options you can select, like run on debug mode, or change the host or the port by default. If you want your application available on the internet, you need to indicate the IP of the interface where the server should be listening to, in this case run(host=' 152.96.56.43').
The default server in Bottle is based on WSGIServer, which is single-threaded. When the load in the server is increased, it should be change to a Multi-threaded or asynchronous one, and add Multi-processing.

The routing in the application is handled by the Bottle route() function. It allows static or dynamic routing in a nice and easy way. The static routes are only valid for one URL that will match it completely. The dynamic routes contain one or more variables, and will accept any URL that fixes on that pattern at the same time.
To define more specific wildcards, filters can be used, like indicating that the wildcards must be an integer or a float, or that it has to pattern a specific regular expression. Each wildcard matches one or more characters, but stops at the first slash (/).
Every wildcard is used later as value of a parameter in the function that is called and will be executed once the URL is matched with a route.
Also routing can be used only to redirect to another HTML. In this case, we utilize the Bottle view() function to indicate the HTML we want to load. As we are using two functions: one to recognize the URL written in the browser and another to select the HTML file where the code is, we do not need to reference in the website the same name of our file.

Before sending a response, we can use response.content_type to indicate the header that should be used in the document and how to encode the Unicode strings. The header by defaults is "text/html; charset=UTF8". Some used in this application are "image/png" to display the tiles, or "application/json" to show the UTFGrid or the metadata information, or "application/xml" in the case of WMTS.
Bottle permits to check the response.status attribute to know if everything is working correctly: "200 OK", which is the status by default. If that is not the case, we can use the Bottle abort() function or return an HTTPResponse instance with the appropriate error status code.

3.2. Object Browser

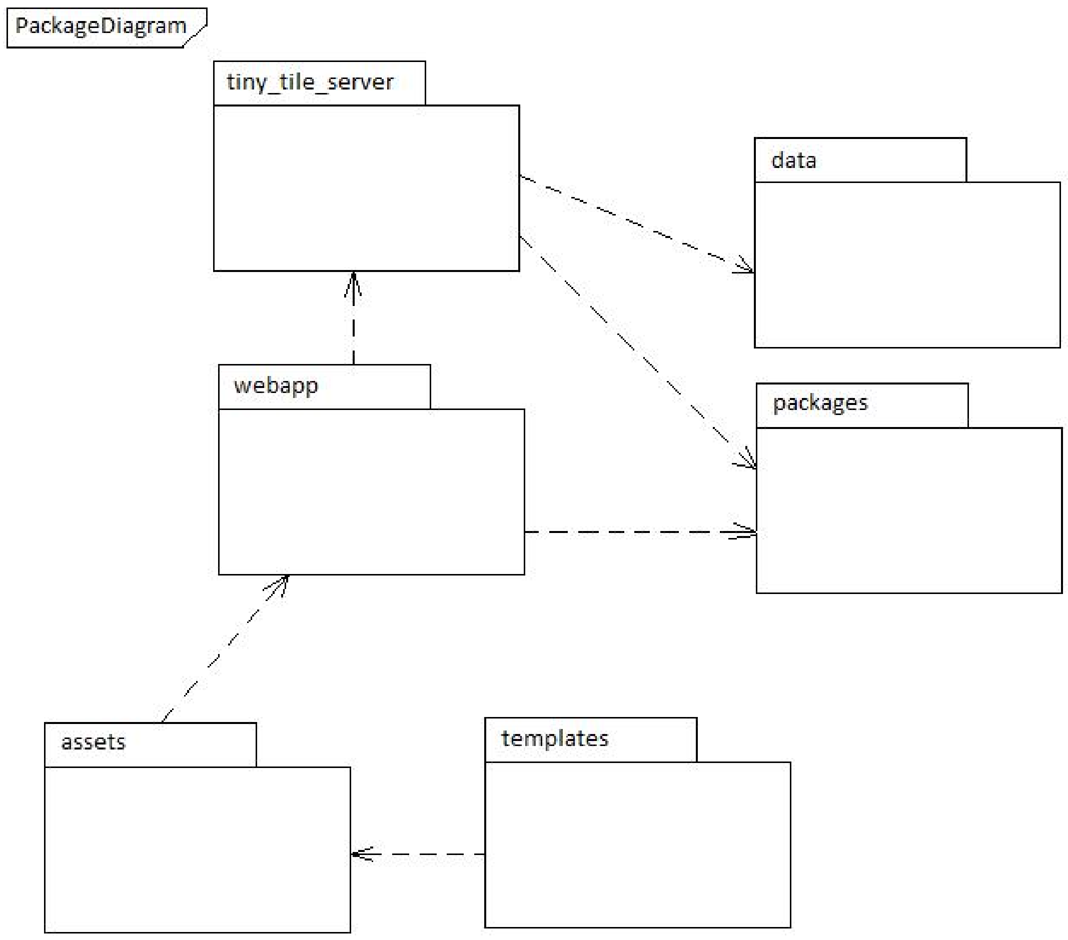
This is the project structur, as we can see is divided into packages that intended to have the less possible relation between them, so they can be reused without many modifications.


```
tts
  assets
    bootstrap
    css
    img
    js
    leaflet_slider
    leaflet_utfgrid
    wax
  data
  packages
    bottle.py
  templates
  tiny_tile_server
    base.py
    base.pyc
    python_server.py
    python_server.pyc
    python_tms.py
    python_wmts.py
    python_wmts.pyc
    __init__.py
    __init__.pyc
  webapp
    api.py
    api.pyc
    web.py
    web.pyc
    __init__.py
    __init__.pyc
  app.py
```

Now I will explain how the packages are internally structured and later I will specify the different functions of every object.

3.3. Package and class diagrams

Here are showed the dependencies between the different packages. As can be seen, we are following the low coupling evaluative pattern. There is a lower dependency between the packages, so in case the is a change in one package, the impact in the others is lower. And exists a higher reuse potential.



Now we are going to examine the packages.

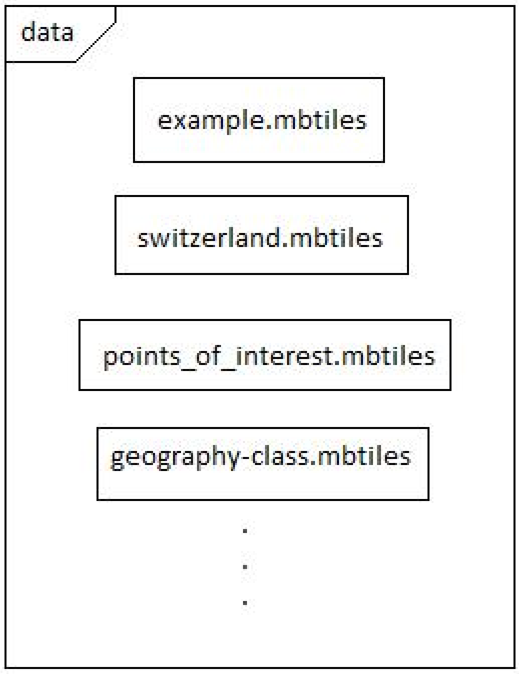
The templates package contains the HTML of the application, here are all the pages in the website. But only the HTML part, the scripts are in its appropriate package and in the HTML file is a reference to it.

The assets package contains the CSS, the JavaScript and the images required in the applications. There are two groups:

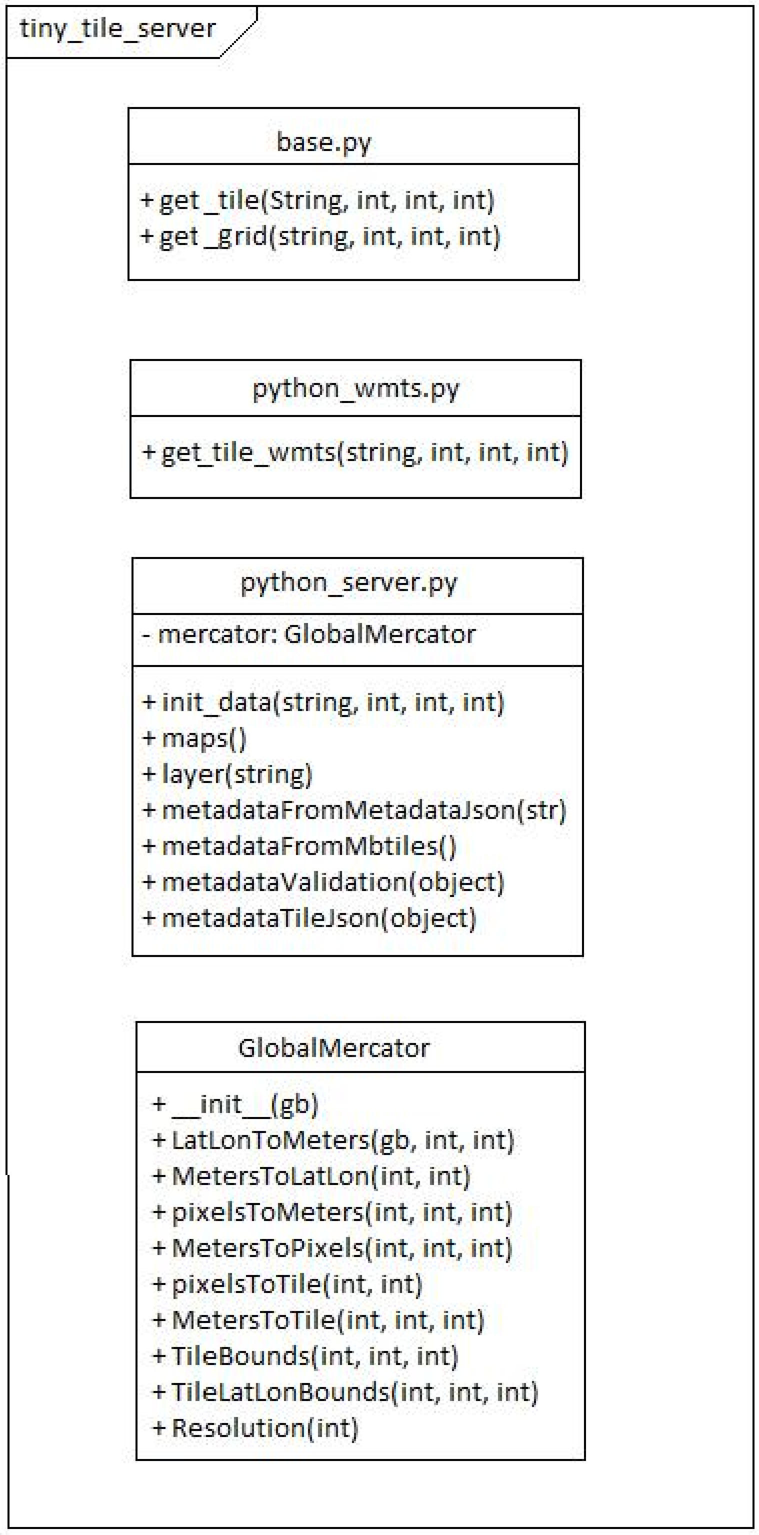
- The external code: the code that is used to configure and design the application, like Bootstrap. And the code that adds some functionality like Wax.
- The internal code: the one that I have developed, so it is specific of my application. Here is included the CSS code and the JavaScript necessary to create the maps.

In the “package” package is included the Bottle file.

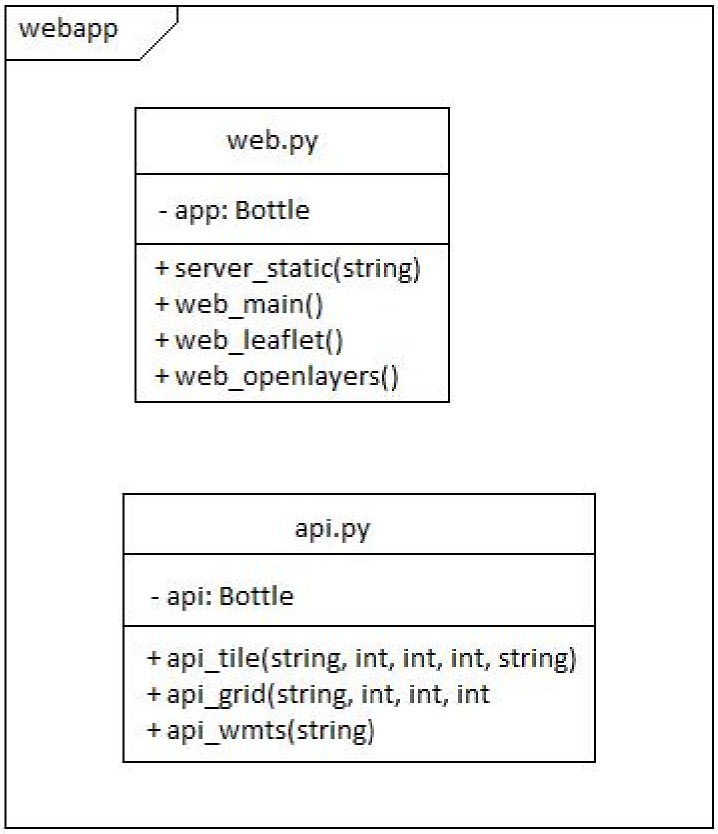
In the data package we will only include ours maps in MBTiles format.


data
example.mbtiles
switzerland.mbtiles
points_of_interest.mbtiles
geography-class.mbtiles
.
.
.

In the tiny_tile_server package are the necessary files to display a map on the internet. This would be explain in detail later.


tiny_tile_server
base.py
+ get_tile(String, int, int, int)
+ get_grid(string, int, int, int)
python_wmts.py
+ get_tile_wmts(string, int, int, int)
python_server.py
- mercator: GlobalMercator
+ init_data(string, int, int, int)
+ maps()
+ layer(string)
+ metadataFromMetadataJson(str)
+ metadataFromMbtiles()
+ metadataValidation(object)
+ metadataTileJson(object)
GlobalMercator
+ __init__(gb)
+ LatLonToMeters(gb, int, int)
+ MetersToLatLon(int, int)
+ pixelsToMeters(int, int, int)
+ MetersToPixels(int, int, int)
+ pixelsToTile(int, int)
+ MetersToTile(int, int, int)
+ TileBounds(int, int, int)
+ TileLatLonBounds(int, int, int)
+ Resolution(int)

In the webapp package are the files that handle the routing in the application. This would also be explained in detail in the next chapter.



As we can see, there is a high cohesion in every package because the responsibilities of a given element are strongly related and highly focused. Breaking programs into classes and subsystems increase the cohesive properties of a system. High cohesion make the system easy to use and maintain, and it is prepared for the change.

4. Development

4.1. Implementation

Tiny Tile Server

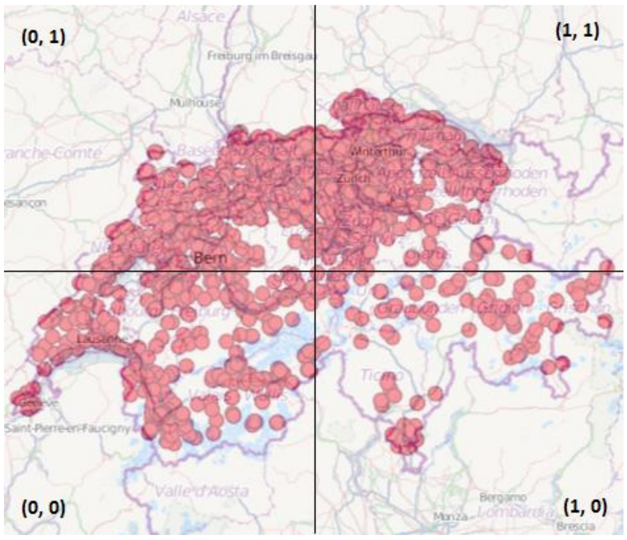
This is the main part of the application: the server. Here is handled all the functionality related with extracting the map inside our MBTiles database and returning the information so the client part can display it. This part has been divided according with the tables inside the database that are accessed. In the base.py file is the necessary code to extract the information out of the MBTiles database.

The tiles table has four attributes: the zoom, the x coordinate, the y coordinate and the png image associated with those values. To extract the tiles I have defined the get_tile() function, that receives the name of the database to connect to, the x, y, and zoom; and return the tile in that position.

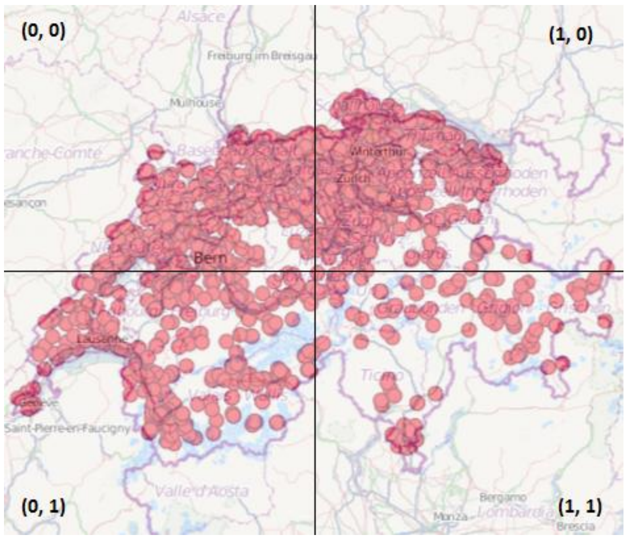
The grids table has four attributes: the zoom, the x coordinate, the y coordinate and the UTFGrid information associated with those values. To extract the grids I have defined the get_grid() function, that receives the name of the database to connect to, the x, y, and zoom; and return the UTFGrid in that position.
This is divided into two steps. First, we get only the grid part, that for every pixel shows the character associated to it. And then we get the data: every character is represented by its number, that is the key, and the value is the data that will appear on the map when we hover or click in that pixel. The data is stored in others two different tables, so we need to make a join selecting by ids.

The metadata table has two attributes: name and value. And there is stored different information about the map: the bounds, the legend, a description, the URL to access to the tiles… To extract the metadata information I have defined the get_metadata() function, that recives the name of the database to connect to, and return the whole metadata associated to the map.
To only print the metadata there is def_metadata() function that displays the metadata information about the map whose name is indicated. This function is just provided in case the user wants to analyze it MBTiles database, and it does not return anything, it just displays it on the command line.

We have to consider two different approach depending of the kind of access we are doing to the database because of the two protocols used, direct access to XYZ tiles and Web Map Tile Service, have a different origin.



XYZ origin is on left-bottom.



WMTS origin is on left-top

In the python_wmts.py file is the code to work with the Web Map Tile Service protocol. The purpose is to display a ServiceMetadata document as a result of a GetCapabilities operation. This document contain service metadata about the server, including specific information about the tiles of the layers that are available to be requested. In the document is explained how to perform valid requests for tiles. More information can be found here http://www.opengeospatial.org/standards/wmts/

The get_tile_wmts() function receive the name of the name, the value for the row in the tile (x coordinate), the column in the tile (y coordinate) and the matrix in the tile (zoom); and return the XML document with the whole information about our map. This function has two purposes.

First, to obtain the values from the metadata of our map and to make the operations in order to calculate all the results that will be showed. This is perform by some functions that will be explained later.
Second, to display the XML document about our map showing the previous obtained values following the WMTS specification.

In python_server.py are the functions to use the WMTS protocol. This functions are common for several protocol and can be reuse. Also in that file the user has to specify some details about its configuration: config_url is the URL where the application is going to run, and title is the name of the application.

- layer(mylayer): check is the metadata information will be extracted from the metadata.json file or from the metadata table in the MBTiles database.
- metadataFromMetadataJson(jsonFileName): open and read the metadata from the file that receive as parameter, check that it is right and add in metadata the basename with the value of that file.
- metadataFromMbtiles(): connect to the MBTiles database and extract all the metadata information from the metadata table, check that it is right and add in metadata the basename with the value of the layer name.
- metadataValidation(metadata): check and correct the metadata; split the values from the bounds so we get an array with the four values, assign 'mercator' as the value by default for profile in case this is missing, if there is not minzoom it should be 0, if there is it specifies that should be an integer, if there is not maxzoom it should be 18, if there is it specifies that should be an integer, , assign 'png' as the value by default for format in case this is missing.
- metadataTileJson(metadata): add some more values to the metadata information: the 'tilejson', the kind of the 'scheme' and the URL for the 'tiles'.

The Global Mercator class has been developed by Klokan Petr Pridal and can be found here http://www.maptiler.org/google-maps-coordinates-tile-bounds-projection/

Webapp

Here is handled all the routing in the application. This is mainly done using Bottle, and it was the most important reason for deciding in using Bottle. It really simplifies the access to another file, the content type for the response and the variables that are requested. And it is done in a nicely and readable way.

In web.py are performed the calls for the different pages in the website, there is a function to redirect for every page. In @app.route('/name_browser') is specified the name the user write in the browser to request to the page. In @bottle.view('name_file') is specified the name of the html file that must be loaded as a response of the user's request.

In api.py are performed the calls for the functions in tiny_tile_server to get the maps information.

- api_tile will match with a URL that starts with "tile" and it will receive 5 variables: the name of the map, the x coordinate, the y coordinate, the zoom and the image extension; and it will call the get_tile() function in base.py
- api_grid will match with a URL that starts with "grid" and it will receive 4 variables: the name of the map, the x coordinate, the y coordinate, and the zoom; and it will call the get_grid() function in base.py
- api_metadata will match with a URL that starts with "metadata" and finish like "metadata.json". It will receive only 1 variable: the name of the map; and it will call the get_metadata() function in base.py
- api_def_metadata will match with a URL that starts with "metadata_info" and finish like "metadata.json". It will receive only 1 variable: the name of the map; and it will call the def_metadata() function in base.py
- api_tilewmts will match with a URL that starts with "tilewmts" and it will receive 5 variables: the name of the map, the x coordinate, the y coordinate, the zoom and the tiles extension; and it will call the get_tile() function in base.py
- api_wmts will match with a URL that starts with "wmtstile" and it will receive 5 variables: the name of the map, the x coordinate, the y coordinate, the zoom and the tiles extension; and it will call the init_data() function in python_server.py to obtain the getCapabilities of the map.

JavaScript files

Here is the important part of the client side: the templates using Leaflet and OpenLayers to display the map, and Wax to add interactivity. There are different kind of examples, according with which of the following characteristics they own.

- It is in Leaflet.
- It is in OpenLayers.
- XYZ direct access.
- WMTS protocol.
- MBTiles local, using Tiny Tile Server.
- MBTiles hosted in MapBox. A small example in case the user wants to see the difference when using a external server.
- Extracting tiles URL and UTFGrid URL from metadata.
- Creating a TileJSON variable with the tiles URL and the UTFGrid URL. This example is shown because the metadata of the MBTiles database generated by TileMill does not contain the tiles URL nor the UTFGrid URL, so unless the user modify it to add those values, he would need to create the TileJSON variable.

5. Test

5.1. Automatic test procedures

unittest is the Python unit testing framework, which is a version of JUnit written in Python. And it is the one I have used for automatic testing.
In our test class we would write all the test cases that would be executed all together at once. The individual tests are defined with methods whose names start with the word "test". This naming convention informs the test runner about which methods represent tests. There are several elements in our test class:

- runTest() method: it is the one that execute the test for our code. It should test only a fraction of code. Before every runTest() method setUp() would be called to create the necessary objects, and after every runTest() method tearDown() would be called for garbage collection.
- setUp() method: initialize the objects that will be used during the execution of the test case. If it raises an exception while is running, the framework will consider the test to have suffered an error, and the runTest() method will not be executed. It can be redefined by the user.
- tearDown() method: in general it is used to reclaim the memory. In this case, it would kill the objects used and close the connection with databases. If setUp() succeeded, the tearDown() method will be run whether runTest() succeeded or not. Because in any case, the object have been already created. It can be redefined by the user. doCleanups() is called unconditionally after tearDown() (or after setUp() if setUp() raises an exception), and it would call all the clean up functions defined.
- expectedFailure() decorator would have to be indicated when our test is a “negative test”, when we are in fact waiting for a fail to happened.

In our tests we want to assert that our function fulfill a property. Normally, that it is True or False, or that it is equal or different from another value. Mainly these are the ones used when testing.
To check if the zoom, the coordinate x and the coordinate y are the expected when accessing the MBTiles database; both for tiles and for UTFGrid. About the metadata we can check that the name, descriptions, bounds, center, minzoom, maxzoom, scheme kind, filesize, URL where the tiles are, URL where the UTFGrid is, template for the infobox and legend extracted have the corresponding value.

Assert first checks if the returned valued and the expected value are the exact same: list, tuple, dict, set, frozenset or Unicode. And we can also add any type and register it with addTypeEqualityFunc(). This function registers a type-specific method called by assertEqual() to check if two objects of exactly the same typeobj are the same.
That is really useful to check the UTFGrid in .json format. First, to verify that the output follows the appropriate pattern. And then that the UTFGrid was the one that we wanted to extract.

All the assert methods (except assertRaises() and assertRaisesRegexp(), whose purpose is to check that exceptions and warnings are raised) accept a message argument that, if specified, is used as the error message on failure. By default, message = None

The last part in our test class would be a way to run the tests. unittest.main() provides a command-line program that loads a set of tests from module and runs them; this is primarily for making test modules conveniently executable. The simplest use for this function is to include the following line at the end of a test script:

```
if __name__ == '__main__':
   unittest.main()
```

Tests have 3 possible outcomes:

ok	The test passes.
FAIL	The test does not pass, and raises an AssertionError exception.
ERROR	The test raises an exception other than AssertionError.

5.2. Manual test procedures

This section is dividing into two parts:

1. Testing with sqlite3.

Sqlite3 is a command-line utility included in the SQLite library that allows the user to manually enter and execute SQL commands in a SQLite database. Sqlite3 has been really useful to know the internal structure of the MBTiles database.

Command	Result	Comment
sqlite3 example.mbtiles		To connect to the database
.tables	grid_data grid_key grid_utfgrid grids images keymap map metadata tiles	To know what table are inside the database
.schema tiles	CREATE VIEW tiles AS SELECT map.zoom_level AS zoom_level, map.tile_column AS tile_column, map.tile_row AS tile_row, images.tile_data AS tile_data FROM map JOIN images ON images.tile_id = map.tile_id;	To display the SQL statement to create the table

Once I came to understand how the MBTiles database works, I started to create test to check that my MBTiles map follows the structure that it was supposed to. The tests were based on checking that the values were stored correctly.

- About zoom_level, tile_row, tile_column: the maximum and minimum value, the value in any case depending on the others parameters
  >sqlite3 example.mbtiles "select distinct zoom_level from tiles"
- About UTFGrid: how they were organized between the different tables, how to extract them to get appropriate format for the .grid.json files
  >sqlite3 example.mbtiles "select * from keymap order by key_name"

2. Direct access on the website.

Before accessing to the whole map, it was necessary to know that each particular tile and grid was in the position (regarding zoom level, tile row and tile column) that I was expecting. And these individual calls help to ascertain that.
That was especially important when dealing with different protocols: XYZ direct access, which origin is on left-bottom, like TMS, and the same that MapBox considers; and WMTS protocol, which origin is on left-top. That was easily to adapt with this formula y_wmts = (2**int(z) - 1) - int(y)

- To get one tile

/api/tile/example/5/10/16.png

Port-of-Spain
:ia
Ciudad Guayana
Georgetown
P
/acucho
Boa Vista

In the UTFGrid case, they are stored starting on top-left. And this does not need to be changed depending on the kind of protocol used.

- To get a UTFGrid file

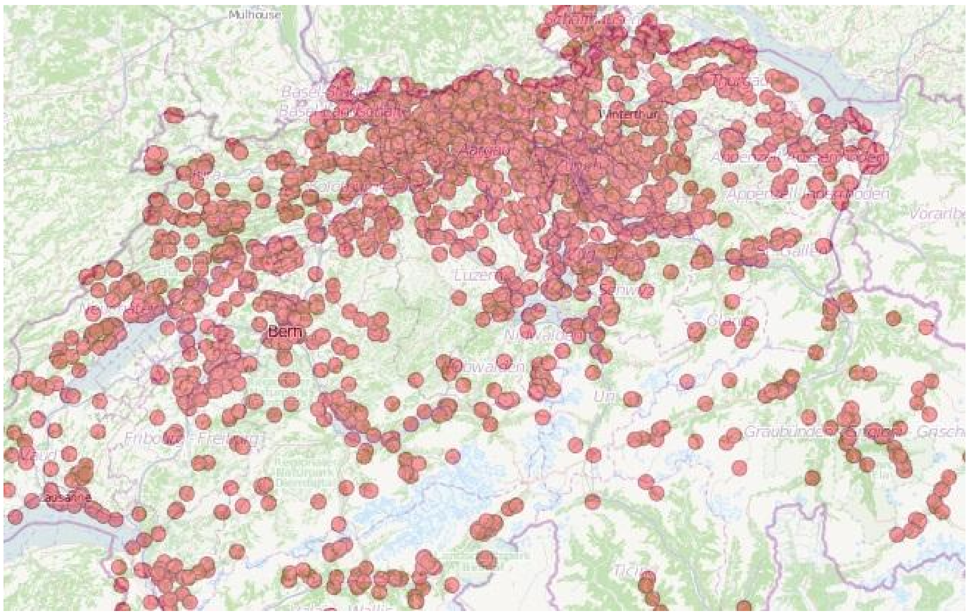
/api/grid/example/5/10/16.grid.json

```
grid({"keys": ["", "1142", "463", "1291", "1280"], "data": {"1142":
{"Name": "Negro"}, "1291": {"Name": "Madeira"}, "1280": {"Name":
"Guapor\u00e9"}, "463": {"Name": "Amazonas"}}, "grid": ["   !
", "   !!
", "     !!!    !!   !!   !
", "                       !
", "                         !
", "                          !
", "                            !
", "                              !
", "                                !!
", "                                  !
", "
", "                                   !
", "                                    !
", "
", "     #####                           !                  # ###
", "  # ##    #                           !
", "             #                         !              #
", "              #                         !            #
", "               #                           !  #    ###
", "                #                      ## #      #$
", "                 #                    #          $
", "                   #         #    #             $
", "                     ###   ##  ###             $
", "                         #                    $
", "                                              $
", "                                            $$
", "                                           $
", "                                           $
", "                                         $$
", "
", "                                       $
", "                                      $
", "                                    $$
", "                                  $$
", "                                 $
", "                              $ $
", "                              $
", "                             $
", "                             $
", "
", "                          $$
", "                          $
", "
", "                          $
", "                          $
", "                          $
", "                          $
", "                        $
", "                       $
", "                     $$
", "                    $
", "                   $
", "                 $$
", "                $
", "               $
", "            $
", "            $
", "            $
", "
", "            $
", "
", "            &
", "            &
", "            &
"]})
```

Once the unit test were finished, I did some integration test to be sure that all together was working correctly.

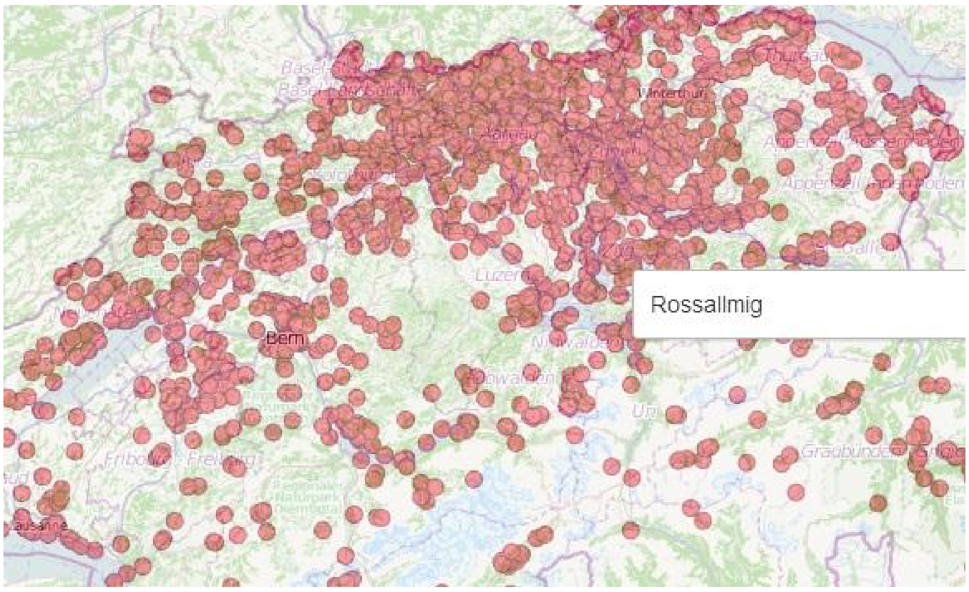
- To get all the tiles forming a part of the map

/switzerlandLLlocal



- Once the UTFGrid part was working, to add that information on the map

/switzerlandLLlocal



6. Further development

6.1. Results

Tiny Tile Server is a small server to extract tiles (map images smaller than screen for better performance), UTFGrid (textual information in JSON for each tile needed for infobox) and metadata (map description, indicating name, bounds, center, minzoom, maxzoom, scheme kind, filesize, tiles URL, UTFGrid URL, template for the infobox and legend) from a MBTiles database to display the corresponding map on the internet.

All the individual access to a tile, the UTFGrid for a tile or the metadata information works correctly. Also putting together the different tiles to display the part of the map the user wants to see works perfectly fine. But the UTFGrid part does not work fine in all the possible scenarios.

- When UTFGrid works normally and is displayed:
    1. At the beginning when the map is loaded for the first time.
    2. After zoom-in or zoom-out using the zoom slider.
    3. After doing double click in a point to zoom-in to that point.
    4. After doing shift + select a zone to show it.
    5. After zoom-in or zoom-out using the mouse wheel.
    6. After reloading the page.
    7. Writing in the URL the new location, like /exampleLLlocal#5/51.124/39.155

In all this cases, the UTFgrid works as it is supposed to: when the user hover over a pixel with UTFGrid information associated to it, this information appears in an infobox following the template defined by the user.

- When UTFGrid does not work:
    1. After doing single click and dragging the map to a new position.

In this case, the UTFGrid corresponding to the previous position keep being displayed, like if the map has not been moved. Internally, the connection with the database is done, and the UTFGrid is extracted, but not displayed. Or it could be that it is displayed, but the previous UTFGrid is displayed on top of it.

Because all the positives scenarios, in my opinion is clear that Tiny Tile Server is working correctly, and the problem is on the client side, when using the JavaScript library and Wax to display the map. It could be that the template is wrong and I need to add some more information to make that part works. The point is that it also does not work when using Wax with a map hosted in MapBox, so that makes clear that Tiny Tile Server is working fine. It could be that the client side with Leaflet or OpenLayers is not well done, but observing this example http://www.mapbox.com/wax/interaction-leaf.html, that also does not work like it is supposed to, makes me think that the problem could be inside Wax and not in my templates.

6.2. Possibilities of development

To serve our maps on the internet it is clear that we need more tools apart from a server. The first step is, of course, to create our MBTiles database, but there are already a lot of tools to create it or to generate it having the map in another format. And the last part is to form the whole map collecting the individual tiles; again in this case there enough good libraries that already do it perfectly well.

But there is something that has not been completely developed, and it is how to deal with interactivity. Here I have been using Wax: it is a good option because it is simple to use and easy to integrate, apart that it is available for different libraries; but it is mainly focused to be use with ModestMaps, so most of the examples and well documented tutorial are for that library. With Leaflet it works well enough, but with OpenLayers there are several compatibility problems, and even they recommended not to use it, at least for first projects.
Wax is not the only option regarding UTFGrid. Since recently OpenLayers allows to add a UTFGrid layer to the map, and the user just needs to specify the callback function. Also Leaflet count with a plugin that enables the interactivity.

The first interesting improvement that could be added to Tiny Tile Server would be to develop the part related with displaying the UTFGrid information on the map. This should be done in two phases.

1. To define a UTFGrid layer. Not needed in OpenLayers because it already exists. This part could be more complicated because it would be necessary to know well enough how the library is designed. But it could be similar to adding another kind of layer that are already added. So the part left would be to decode the UTFGrid and get the corresponding name.
2. To display the data obtained and already decoded in an infobox. This could be more or less straight forward if we are satisfied with a simple design like showing the data just in a div under the map. But there much more interesting possibilities. The infobox could appear near where the mouse is located. The infobox could be static and do not disappear until the user click in its closing button. The infobox could appear when hovering or when clicking, and also show different information.

Another possible development is to support more protocols to serve maps apart from direct access in XYZ and Web Map Tile Service. Two possibilities are Tile Map Service and Web Map Service.

- TMS request individual tiles, like WMTS, so it is faster. But its origin is on left-bottom, not like WMTS's which is on left-top. So it would need to be adapted.
- WMS request a map with a particular bounding box with specific information, so that makes it more powerful, but slower.

The TMS implementation should be similar to WMTS. The functions to get the metadata information and to transform the latitude longitude system is already done for WMTS and it is the same. It would be needed to generate the XML template. But again the operation mode is similar, the difference is just the XML structure.

TMS provides access to different resources, like rendered tiles. Each resource contains the descriptive information and links to further resources. We can get the different versions of the Tile Map Service, the metadata information, the tile maps that can be accessed, the whole map formed by all the tiles.
The WMS implementation would need a greater effort. Because it does not extract tiles, but a whole part of the map. In any case, the Capabilities in XML format template would only differ in the structure.

Another interesting improvement could be HTTP caching. Once we have access to the tiles in the server, we could provide a cache functionality, so the tiles would be cached and stored and there is no need to request them from the server the next time they are accessed.

Project management

1. Project management

I have used the Scrum agile project methodology and worked there for 16 weeks. The duration of a sprint I have determined to be 2 weeks.
Scrum is a way to work to develop a product. Product development, using Scrum, occurs in small pieces, with each piece building upon previously created pieces. Building products one small piece at a time encourages creativity and enables us to respond to feedback and change, to build exactly and only what is needed.
More specifically, Scrum is a simple framework for effective collaboration on complex projects. Scrum provides a small set of rules that create just enough structure for us to be able to focus our innovation on solving what might otherwise be an insurmountable challenge.
However, Scrum is much more than a simple framework. Scrum supports our need to be human at work: to belong, to learn, to do, to create and be creative, to grow, to improve, and to interact with other people. In other words, Scrum leverages the innate traits and characteristics in people to allow them to do great things.

Sprint 1

18th February 2013 to 3rd March 2013

Main tasks / focus in the sprint

- Setting up the infrastructure (repository, project management, development environment)
- Project setup
  - GitHub as repository
  - Website with show case and a gallery
  - Scrum methodology
- Familiarization with the subject
  - OpenStreetMap
  - OpenLayers
  - Leaflet
  - TileMill
  - MapTiler
  - MapBox
- Scenarios
  - Using TileMill
  - Using MapTiler

Goals

- Get to know OpenStreetMap (data structure, API)
- Use OpenLayers and Leaflet as clients to read MBTiles
- Learn how TileMill and MapTiler work from the users point of view

Delivery / Deliverables

- Project setup with GitHub
- Document containing exercises using TileMill and MapTiler
- Website with these exercises

Completed works

How to use TileMill: little example with layers and CartoCSS, hosted the map in MapBox, export the map as MBTiles and read it using Leaflet and OpenLayers. Another map with thematic data (Towns, Rivers, Areas… from Switzerland).

How to use MapTiler: example using a GeoTIFF image and publish the tiles using OpenLayers.

Sprint 2

4th March 2013 to 17th March 2013

Main tasks / focus in the sprint

- Familiarization with the subject
  - MapProxy
  - GDAL2Tiles
  - Python
- Scenarios
  - Using MapProxy

Goals

- Learn how to use MapProxy as a server.
- Use OpenLayers and Leaflet with Wax to show legends/tooltips in maps

Delivery / Deliverables

- Document containing exercises using MapProxy and GDAL2Tiles.
- Website with these exercises

Completed works

Read maps generated by TileMill using Leaflet and OpenLayers with Wax. Wax is needed to show the legends and the tooltips that we aggregated to our map.

How to use GDAL2Tiles: example using a GeoTIFF image and publish the tiles using OpenLayers.

Problems

Unfortunately I did not manage to learn enough about MapProxy to be able to finish these exercises.

Sprint 3

18th March 2013 to 31st March 2013

Main tasks / focus in the sprint

- Familiarization with the subject
  - MapProxy
  - Python
  - PostGIS
  - QuantumGIS
  - SQLite
- Scenarios
  - Using MapProxy
  - Using QuantumGIS and TileMill

Goals

- Learn how to create a SQLite database using Quantum
- Learn how to configure MapProxy

Delivery / Deliverables

- Documentation containing example exercise using SQLite file in TileMill
- Documentation about reprojecting, resolutions and seeds in MapProxy

Completed works

How to use QuantumGIS: creating a SQLite database with QuantumGIS using a shapefile

How to use TileMill: increasing the tutorials in TileMill with documentation about how to use a SQLite database as a source for a layer.

Configuration in MapProxy

Problems

The documentation about MapProxy is not finished yet.

Sprint 4

1st April 2013 to 14th April 2013

Main tasks / focus in the sprint

- Familiarization with the subject
  - MapProxy
  - Python
  - PostGIS
  - raster2mb
  - mbutil
- Scenarios
  - Python

Goals

- Program .php files in Python
- Using MapProxy to reproject a local TMS directory by doing MapProxy (TMS in/WMS out) > MapProxy(WMS in/reproject/cache/WMTS out)

Delivery / Deliverables

- Documentaion about raster2mb and mbutil

Completed works

How to use raster2mb and mbutil to generate .mbtiles

Problems

I was not able to do the reprojection using MapProxy

Sprint 5

15th April 2013 to 28th April 2013

Main tasks / focus in the sprint

- Familiarization with the subject
  - Python
  - PostGIS
  - MapTiler Pro
  - MapProxy
- Scenarios
  - Python

Goals

- Program Python code
- Learn how MapTiler Pro works from the users point of view

Delivery / Deliverables

- Tests and documentation using MapTiler Pro
- Tutorial about MapProxy: how to use it.

Completed works

Changes in the website.

MapTiler Pro tests.

MapProxy tutorial.

Problems

Trouble with Python code: how to read the info data.

Sprint 6

29th April 2013 to 12th May 2013

Main tasks / focus in the sprint

- Familiarization with the subject
  - Python
- Scenarios
  - Python

Goals

- Program Python code

Delivery / Deliverables

- Python code manual tests

Completed works

Reading images from MBTiles using Python code.

Reading UTFGrid from MBTiles using Python code.

Testing Python code for getting tiles and UTFGrid.

Problems

How to show the UTFGrid on the map; I can read it using the Python code and I can see it on the browser, but I cannot show it on top of the map when accessing it.

Sprint 7

13th May 2013 to 26th May 2013

Main tasks / focus in the sprint

- Familiarization with the subject
  - Python
- Scenarios
  - Python

Goals

- Program Python code
- Finish Web Map Tile Service protocol: GetCapabilities

Delivery / Deliverables

- Python code manual tests documentation

Completed works

Changes in the website.

Reading the ServiceMetadata document from MBTiles using Python code.

Testing Python code for displaying the XML document following the Web Map Tile Service protocol.

Problems

It is only working for Leaflet and not for OpenLayers.

Sprint 8

27th May 2013 to 9th June 2013

Main tasks / focus in the sprint

- Familiarization with the subject
  - Python
- Scenarios
  - Python

Goals

- Program Python code
- Finish displaying UTFGrid information on the map

Delivery / Deliverables

- Python code automatic tests documentation
- Results and further development documentation

Completed works

Web Map Tile Service protocol working completely for Leaflet and Openalyers. Also including GetCapabilities request and response.

Extracting UTFGrid working fine with Leaflet.

Problems

UTFGrid does not work with OpenLayers. And there are also problems with Wax when using it with a hosted file in MapBox.

2. Project monitoring

2.1. Target-time comparison

I consider that I did have not enough time to develop the whole project and to write the thesis. The bachelor thesis was supposed to be realized in 360 hours. I have invested 415 hours and still I am not completely satisfied with the project.

I guess I should have previously known more about the technologies I was going to employ. Like Python, that I have to learn from the basics. I also did not know enough about map related topics, like Geographic Information System or projections, and that was a complicate part to research.
Looking backward, I think I have dedicated too much time to make some research that were not so important, some of them I did not even use. But at that time I did not know what would be useful or necessary.

Regarding the software product, Tiny Tile Server and the website in general, I think I have dedicated to it all the time possible. And I am satisfied with it because Tiny Tile Server works perfectly fine. So the extra time would have been only to make necessary changes in the website.

I wish I had had more time to write the thesis, maybe to organize some matters differently and to explain some topics better.

2.2. Code statistics

Tiny Tile Server

File	Number of lines	Number of comments
app.py	35	4
webapp/web.py	118	4
webapp/api.py	42	6
tiny_tile_server/base.py	133	25
tiny_tile_server/python_server.py	176	29
tiny_tile_server/python_wmts.py	470	3
Total	974	71

Website

File	Number of lines	Number of comments
templates/layout.html	70	8
templates/main.html	42	0
templates/tutorials.html	17	0
templates/protocol.html	35	0
templates/planning.html	27	0
templates/tileMillExample.html	136	0
templates/tileMillSwitzerland.html	46	0
templates/{Leaflet}.html	27	3
templates/{OpenLayers}.html	35	3
assets/js/switzerlandLL.js	75	23
assets/js/exampleLLlocal.js	86	22
assets/js/switzerlandLLlocal.js	82	24
assets/js/exampleOLlocal.js	86	20
assets/js/switzerlandOLlocal.js	80	21
assets/js/exampleOLwmts.js	80	17
assets/js/switzerlandLLwmts.js	78	21

Appendix

1. Contents of the CD

Path	Description
TinyTileServer.pdf	Thesis documentation in PDF format
software_project/	Software project web application
software_project/readme.txt	How to install and run the application
software_project/app.py	Application init file
software_project/assets/ bootstrap	Bootstrap files
software_project/assets/css	Application CSS files
software_project/assets/img	Application images
software_project/assets/js	Application JS files. In particular the JavaScript files for creating the maps
software_project/assets/leaflet _hash	Plugin for Leaflet to add the actual location in the URL
software_project/assets/leaflet _slider	Puglin for Leaflet to add a zoom with slider
software_project/assets/openla yers	OpenLayers files
software_project/assets/wax	Wax files
software_project/data	Maps in MBTiles format
software_project/packages	Bottle file
software_project/templates	Application HTML files
software_project/tiny_tile_serv er	Python scripts to extract a map from a MBTiles database
software_project/tiny_tile_serv er/python_server.py	To indicate the URL and title application
software_project/webapp	Files to handled routing, variables requesting and response content type

2. Glossary and abbreviations

Amazon CloudFront is a web service for content delivery. It integrates with other Amazon Web Services to give developers and businesses an easy way to distribute content to end users with low latency, high data transfer speeds, and no commitments.

Amazon S3 is storage for the Internet. It is designed to make web-scale computing easier for developers. Amazon S3 provides a simple web services interface that can be used to store and retrieve any amount of data, at any time, from anywhere on the web. It gives any developer access to the same highly scalable, reliable, secure, fast, inexpensive infrastructure that Amazon uses to run its own global network of web sites. The service aims to maximize benefits of scale and to pass those benefits on to developers.

ArcGIS is a geographic information system (GIS) for working with maps and geographic information. It is used for: creating and using maps; compiling geographic data; analyzing mapped information; sharing and discovering geographic information; using maps and geographic information in a range of applications; and managing geographic information in a database. The system provides an infrastructure for making maps and geographic information available throughout an organization, across a community, and openly on the Web.

CartoCSS is a stylesheet renderer for Mapnik. It's an evolution of the Cascadenik idea and language, with an emphasis on speed and flexibility. You can find an online reference here http://www.mapbox.com/carto/api/2.1.0/.

Bottle is a fast, simple and lightweight WSGI micro web-framework for Python. It is distributed as a single file module and has no dependencies other than the Python Standard Library. It can be found here http://bottlepy.org/docs/dev/.

GDAL2Tiles is a command line tool that allows easy publishing of raster maps on the Internet. The raster image is converted into a directory structure of small tiles which you can copy to your webserver.

Geographic data describe our world allows for city planning, flood prediction and relief, emergency service routing, environmental assessments, wind pattern monitoring and many other applications. Geographic data is processed with geographic information system (GIS) software which can, as one aspect of its functioning, produce maps.

GeoJSON is a format for encoding a variety of geographic data structures. A GeoJSON object may represent a geometry, a feature, or a collection of features. GeoJSON supports the following geometry types: Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon, and GeometryCollection. Features in GeoJSON contain a geometry object and additional properties, and a feature collection represents a list of features.

GeoTIFF is a public domain metadata standard which allows georeferencing information to be embedded within a TIFF file. The potential additional information includes map projection, coordinate systems, ellipsoids, datums, and everything else necessary to establish the exact spatial reference for the file.

Geographic information system, GIS, is a system designed to capture, store, manipulate, analyze, manage, and present all types of geographical data.

JSON or JavaScript Object Notation, is a text-based open standard designed for human-readable data interchange. It is derived from the JavaScript scripting language for representing simple data structures and associative arrays (objects). Despite its relationship to JavaScript, it is language-independent, with parsers available for many languages. The JSON format is often used for serializing and transmitting structured data over a network connection. It is used primarily to transmit data between a server and web application, serving as an alternative to XML.

JSONP or "JSON with padding" is a communication technique used in JavaScript programs which run in Web browsers. It provides a method to request data from a server in a different domain.

Latitude is a geographic coordinate that specifies the north-south position of a point on the Earth's surface . It is an angular measurement, usually expressed in degrees and denoted by ϕ.

Layer is an object on the map that consist of one or more separate items, but are manipulated as a single unit. Layers generally reflect collections of objects that you add on top of the map to designate a common association. The Maps API manages the presentation of objects within layers by rendering their constituent items into one object (typically a tile overlay) and displaying them as the map's viewport changes.

Leaflet is a modern open-source JavaScript library for mobile-friendly interactive maps. Leaflet is designed with simplicity, performance and usability in mind. It works efficiently across all major desktop and mobile platforms out of the box, taking advantage of HTML5 and CSS3 on modern browsers while still being accessible on older ones. It can be extended with many plugins, has a beautiful, easy to use and well documented API and a simple, readable source code that is a joy to contribute to. You can find it here http://leafletjs.com/reference.html

Longitude is a geographic coordinate that specifies the east-west position of a point on the Earth's surface. It is an angular measurement, usually expressed in degrees and denoted by λ.

MapBox is a company that creates different tools for styling and deploying maps as well as providing services for hosting maps. Many of these tools use OpenStreetMap data and involve large open-source efforts and MapBox is also a main contributor to the Mapnik renderer project.

Mapnik is an open source toolkit for rendering maps. Among other things, it is used to render the four main Slippy Map layers on the OpenStreetMap website. It supports a variety of geospatial data formats and provides flexible styling options for designing many different kinds of maps. It can read ESRI shapefiles, PostGIS, TIFF rasters, .osm files, and any GDAL or OGR supported formats.

MapProxy is an open source geospatial tile proxy that supports reprojection. Mapproxy is a Python proxy server for geospatial images. It can read data from WMS, tiles, mapserver and mapnik, and cache and serve that data as WMS, WMTS, TMS and KML. It can also do reprojections between different coordinate reference systems.

MapTiler is graphical application for online map publishing. Your map can create overlay of standard maps like Google Maps, Yahoo Maps, Microsoft VirtualEarth or OpenStreetMap and can be also visualized in 3D form by Google Earth. Only thing you have to do for publishing the map is to upload the automatically generated directory with tiles into your webserver.

MBTiles provides a way of storing millions of tiles in a single SQLite database making it possible to store and transfer web maps in a single file. And because SQLite is available on so many platforms, MBTiles is an ideal format for reading tiles directly for serving on the web or displaying on mobile devices.

Metadata is the description of the map regarding its characteristics. Normally: name, bounds, center, minzoom, maxzoom, scheme kind, filesize, tiles URL, UTFGrid URL, template for the infobox and legend

OpenLayers is an open source JavaScript library for displaying map data in web browsers. It provides an API for building rich web-based geographic applications similar to Google Maps and Bing Maps. The library was originally based on the Prototype JavaScript Framework. OpenLayers is used by the OpenStreeMap project for its "Slippy Map" map interface. You can find the documentation here http://docs.openlayers.org/

Open Source Software (OSS) is computer software with its source code made available and licensed with an open source license in which the copyright holder provides the rights to study, change and distribute the software for free to anyone and for any purpose. Open source software is very often developed in a public, collaborative manner. Open source software is the most prominent example of open source development and often compared to (technically defined) user-generated content or (legally defined) open content movements.

OpenStreetMap, OSM, is a collaborative project to create a free editable map of the world.

PostGIS is a spatial database extender for PostgreSQL object-relational database. It adds support for geographic objects allowing location queries to be run in SQL.

Python is a general-purpose, high-level programming language whose design philosophy emphasizes code readability. Python's syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C, and the language provides constructs intended to enable clear programs on both a small and large scale. Python supports multiple programming paradigms, including object-oriented, imperative and functional programming styles. It features a fully dynamic type system and automatic memory management.

Quantum GIS, QGIS, is a user friendly Open Source Geographic Information System (GIS) licensed under the GNU General Public License. QGIS is an official project of the Open Source Geospatial Foundation (OSGeo). Provides data viewing, editing, and analysis capabilities, and supports numerous vector, raster, and database formats and functionalities.

Rackspace is a global web host known for their high end managed hosting and dedicated services. The company delivers enterprise-level managed services to businesses of all sizes and kinds around the world.

Shapefile, is a popular geospatial vector data format for geographic information system software. It is developed and regulated by Esri as a (mostly) open specification for data interoperability among Esri and other software products. Shapefiles spatially describe features: points, lines, and polygons, representing, for example, water wells, rivers, and lakes. Each item usually has attributes that describe it, such as name or temperature.

SpatiaLite is an open source library intended to extend the SQLite core to support fully fledged Spatial SQL capabilities.

SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. SQLite is the most widely deployed SQL database engine in the world. The source code for SQLite is in the public domain.

TIFF (originally standing for Tagged Image File Format) is a file format for storing images, popular among graphic artists, the publishing industry, and both amateur and professional photographers in general.

Tile is a small image, and contains the topographic information of a rectangular map area as pixel-based graphic. Each tile normally consists of 256 × 256 pixels. This tile-based approach is done to manage maps at higher zoom levels.

Tiled rendering is a technique for subdividing (or tilling) large images in pieces (tiles).

TileJSON is a consistent way of describing a map: zoom levels, center point, legend contents, and more, and in turn provides an easy way to load and display a map the way it is meant to be seen.

TileMill is a design environment developed by MapBox for cartography, constituting Mapnik as a renderer, Carto as a stylesheet language, and a locally-served web interface with node.js as a server and based on Backbone.js for the client. TileMill can load shapefiles or connect to a PostGIS database for rendering. Raster data such as GeoTIFFs can also be rendered. By default TileMill renders to an MBTiles file, an SQLite bundle of tile images that enables compression and faster transfers. PNG and PDF output is also supported for static maps.

UTFGrid is a standard, scalable way of encoding data for hundreds or thousands of features alongside your map tiles. The UTFGrids are invisible “ASCII Art” and attribute data embedded in json. They sit “behind” your map tiles (they are not rendered visually) and allows quick attribute lookups without going back to the server. This allows a high degree of real-time map interactivity in an HTML web map - something that has typically been the strong point of plugin-based maps.

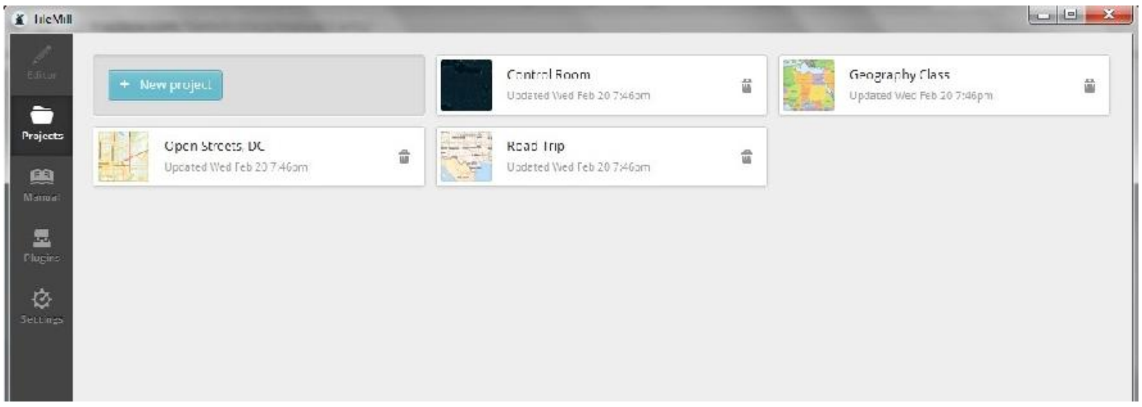
Web Map Tile Service, WMTS, is a standard protocol for serving pre-rendered georeferenced map tiles over the Internet.

Wax is a library that adds common utilities to minimal mapping libraries and aims to give developers and designers ultimate control and flexibility. And It makes it easier to use APIs like Modest Maps and Leaflet. It provides zoom controls and other stuff that people already expect, functionalities that can be found in some portals, but that our map does not show by default when we use our own server.

3. TileMill tutorials

3.1 TileMill: initial example

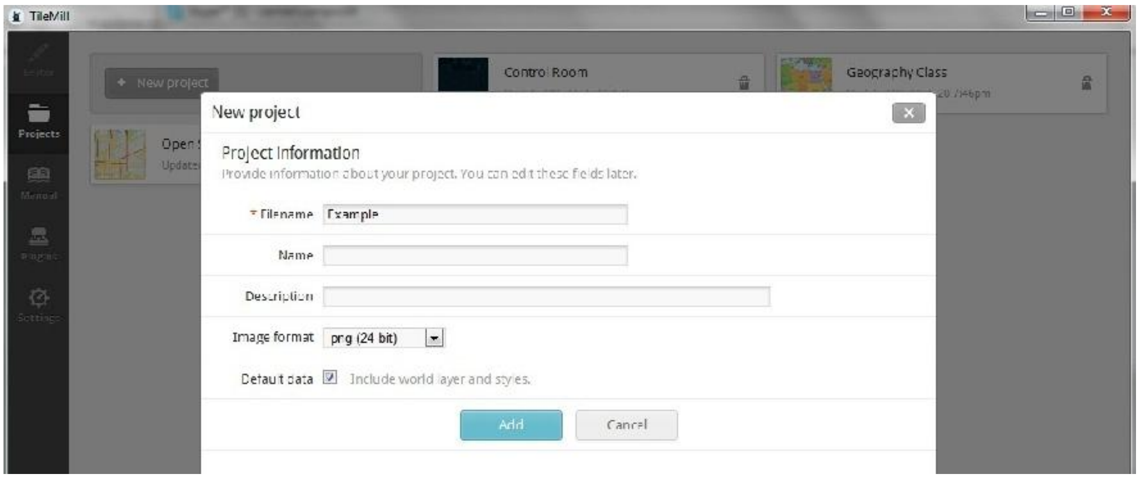
Using TileMill is easy once you have done your first project because everything is intuitive. Once we have open the program, to start creating a map, we have to press the button “New project”.



We need to provide some information about the project. The only require data is the name of the file. But we can also add a name for the project, a description and the image format.
There is an option “Default data” to indicate that we want an initial world layer to appear in our project with an style:

- background-color: #b8dee6; background color.
- line-color: #85c5d3; color of the line that separates the countries.
- line-width: 2; width of the line that separate the countries.
- line-join: round; kind of line that separate the countries.
- polygon-fill: #fff; color to fill the countries.

Then we press “Add”.



New project

Project Information

Provide information about your project. You can edit these fields later.

* Filename: Example

Name:

Description:

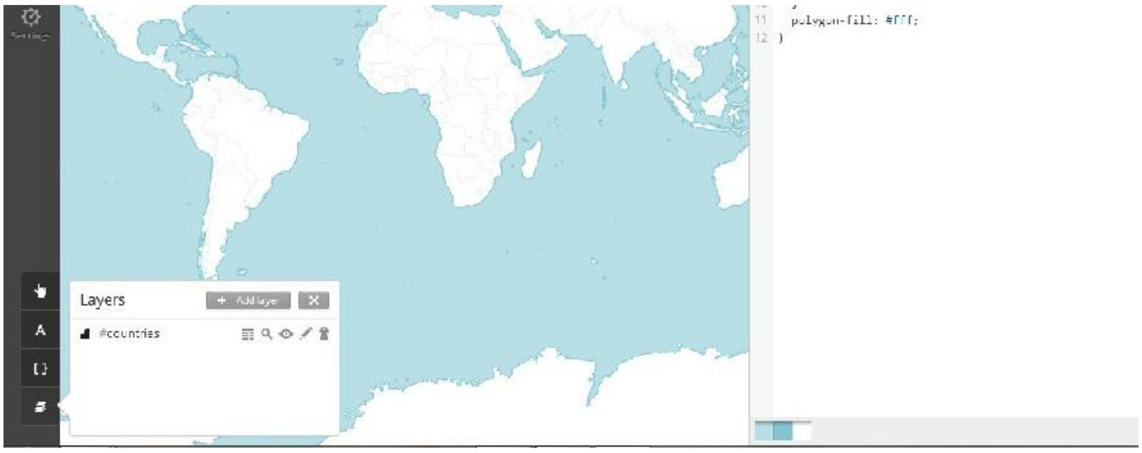
Image format: png (24 bit)

Default data: ☑ Include world layer and styles.

It returns to the main page in TileMill, and we can observe that our project has been created there. To open it, we have to click in its image.

Layers

To begin with, we should add a layer. To do that, we have to select the appropriate image down on the left and press the button "Add layer".

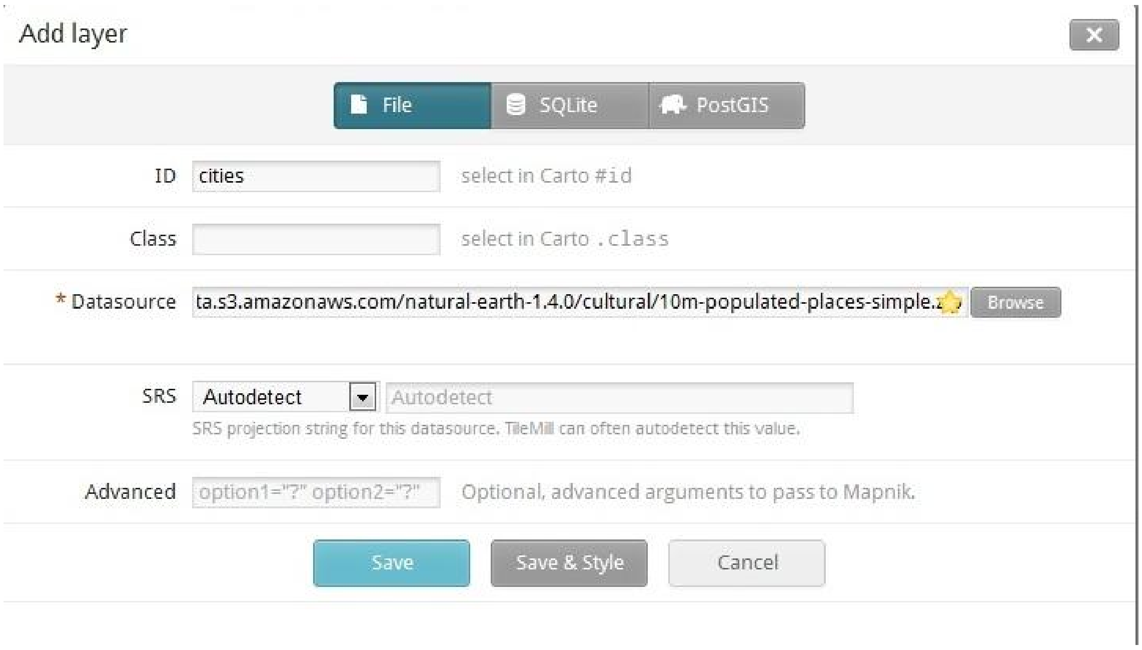


As we can see, there are other options for every layer:

1. To see the features of the layer. This features can be used, for example in this case, to consider only a specific country, or a group of them which satisfy a concrete property; to show the value of a concrete feature in the map.
2. To zoom in the map.
3. To not consider the layer. It is like if we delete the layer. But it is still there, in case we need to consider it again, we do not have to import it.
4. To edit the layer.
5. To delete the layer.

When we add a layer, we have to indicate the datasource address where the file is located; it could be a file in our computer or be located in MapBox. When can also select the file as "favorite", so when we do another project, we do not need to look for the file.

We could assign an ID and a class for the layer, so we can apply a css style to the layer. The ID is necessary, so in the we do not choose any, one will be automatically assigned.


Add layer

File | SQLite | PostGIS

ID: cities — select in Carto #id

Class: — select in Carto .class

* Datasource: ta.s3.amazonaws.com/natural-earth-1.4.0/cultural/10m-populated-places-simple.z — Browse

SRS: Autodetect — Autodetect
SRS projection string for this datasource. TileMill can often autodetect this value.

Advanced: option1="?" option2="?" — Optional, advanced arguments to pass to Mapnik.

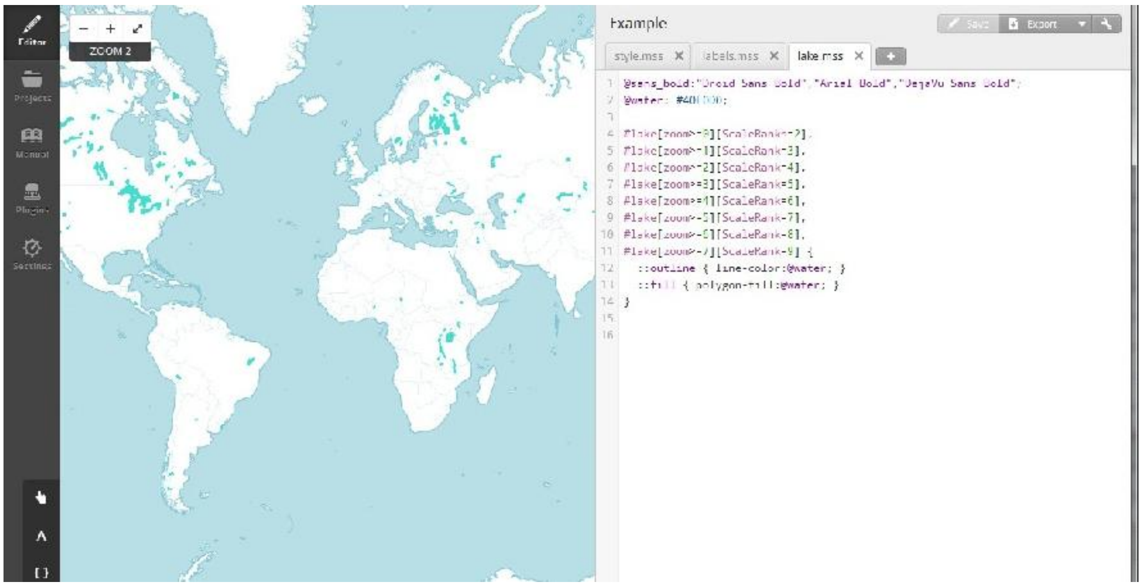
Save | Save & Style | Cancel

We just have to look for the different layers we want to add. Some options are:

- http://mapbox-geodata.s3.amazonaws.com/natural-earth-1.4.0/cultural/10m-populated-places-simple.zip   show the different cities.
- http://mapbox-geodata.s3.amazonaws.com/natural-earth-1.4.0/cultural/10m-admin-0-country-points.zip   show the names of the countries.

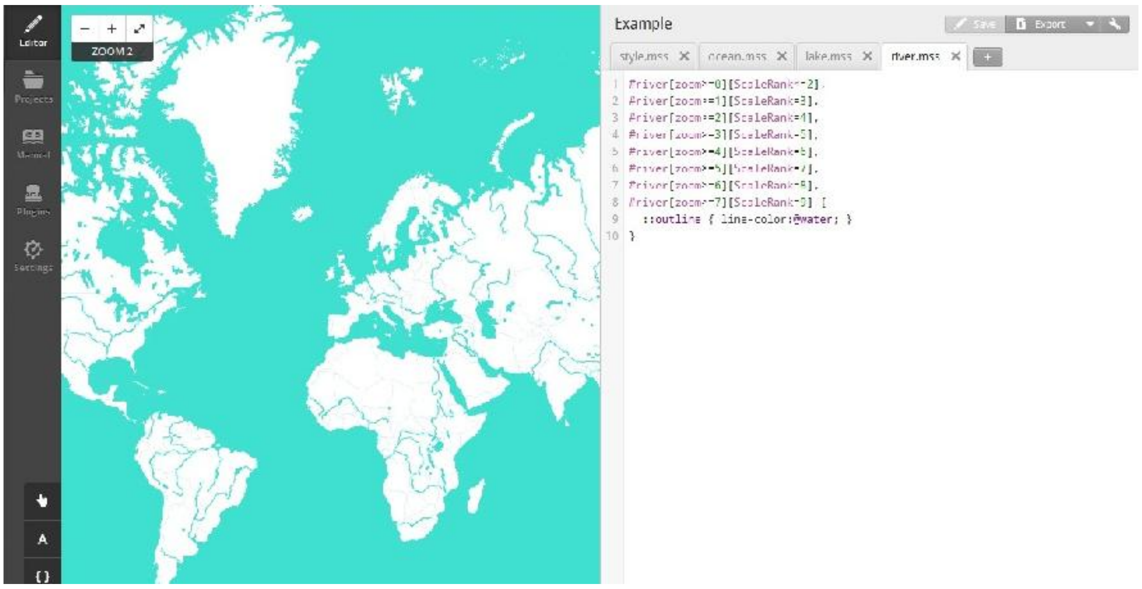
We could add the lakes in the world, which datasource is http://mapbox-geodata.s3.amazonaws.com/natural-earth-1.3.0/physical/10m-lakes.zip, with the ID lake and apply a style:

- @sans_bold:"Droid Sans Bold","Arial Bold","DejaVu Sans Bold";   select the kind of letter, we can only put a name, or several, in case one of the options is not available.
- @water: #40E0D0;   we can define a name for an specific color, in case we want to use the same several times. Of course, we could also write the code of the color every time
- #lake[zoom>=0][ScaleRank<=2],   depending on the zoom we are applying, we will only show determinate lakes. We can see the ScaleRank of every element in the layers image.
- line-color: @water;   line of the lake color
- polygon-fill: @water;   color inside the line



```
@sans_bold:"Droid Sans Bold","Arial Bold","DejaVu Sans Bold";
@water: #40E0D0;

#lake[zoom>=0][ScaleRank<=2],
#lake[zoom>=1][ScaleRank=3],
#lake[zoom>=2][ScaleRank=4],
#lake[zoom>=3][ScaleRank=5],
#lake[zoom>=4][ScaleRank=6],
#lake[zoom>=5][ScaleRank=7],
#lake[zoom>=6][ScaleRank=8],
#lake[zoom>=7][ScaleRank=9] {
  ::outline { line-color:@water; }
  ::fill { polygon-fill:@water; }
}
```

We could also add the oceans and the rivers, which are respectively in:
http://mapbox-geodata.s3.amazonaws.com/natural-earth-1.3.0/physical/10m-ocean.zip
http://mapbox-geodata.s3.amazonaws.com/natural-earth-1.3.0/physical/10m-rivers-lake-centerlines.zip



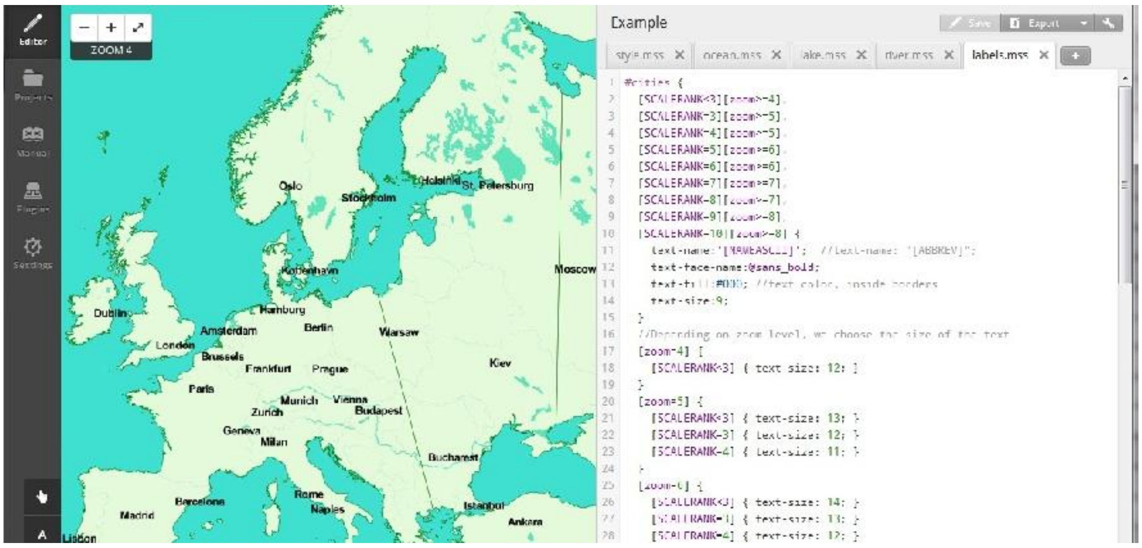
```
#river[zoom>=0][ScaleRank=2],
#river[zoom>=1][ScaleRank=3],
#river[zoom>=2][ScaleRank=4],
#river[zoom>=3][ScaleRank=5],
#river[zoom>=4][ScaleRank=6],
#river[zoom>=5][ScaleRank=7],
#river[zoom>=6][ScaleRank=8],
#river[zoom>=7][ScaleRank=9] {
  ::outline { line-color:@water; }
}
```

In order to show the names of the cities, we need to add this layer http://mapbox-geodata.s3.amazonaws.com/natural-earth-1.4.0/cultural/10m-admin-0-country-points.zip

- text-name:'[NAMEASCII]';   the name of the city, we could have used NAME, NAME_ALT, LS_NAME… In every particular case, we can choose one or another, and we can consult what we are going to show in the features of the layer.
- text-face-name:@sans_bold;   type of letter.
- text-fill:#000;   text color inside borders.
- text-size:9;   size of the letter.
- [zoom=10] {

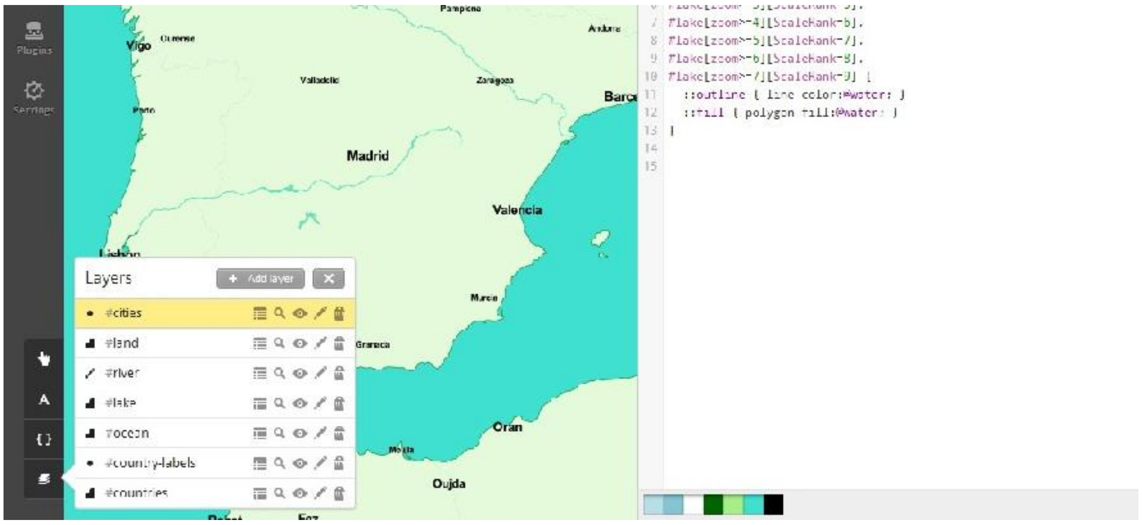
  [SCALERANK<3] { text-size: 16; text-character-spacing:2; }
  [SCALERANK=3] { text-size: 16; text-character-spacing:2; }
  [SCALERANK=4] { text-size: 15; text-character-spacing:1; }
  [SCALERANK=5] { text-size: 15; text-character-spacing:1; }
  [SCALERANK=6] { text-size: 15; text-character-spacing:1; }
  [SCALERANK=7] { text-size: 14; }
  [SCALERANK=8] { text-size: 14; }
  [SCALERANK=9] { text-size: 13; }
  [SCALERANK=10] { text-size: 13; }

 }   depending on zoom level, we choose the size of the text and the space between characters



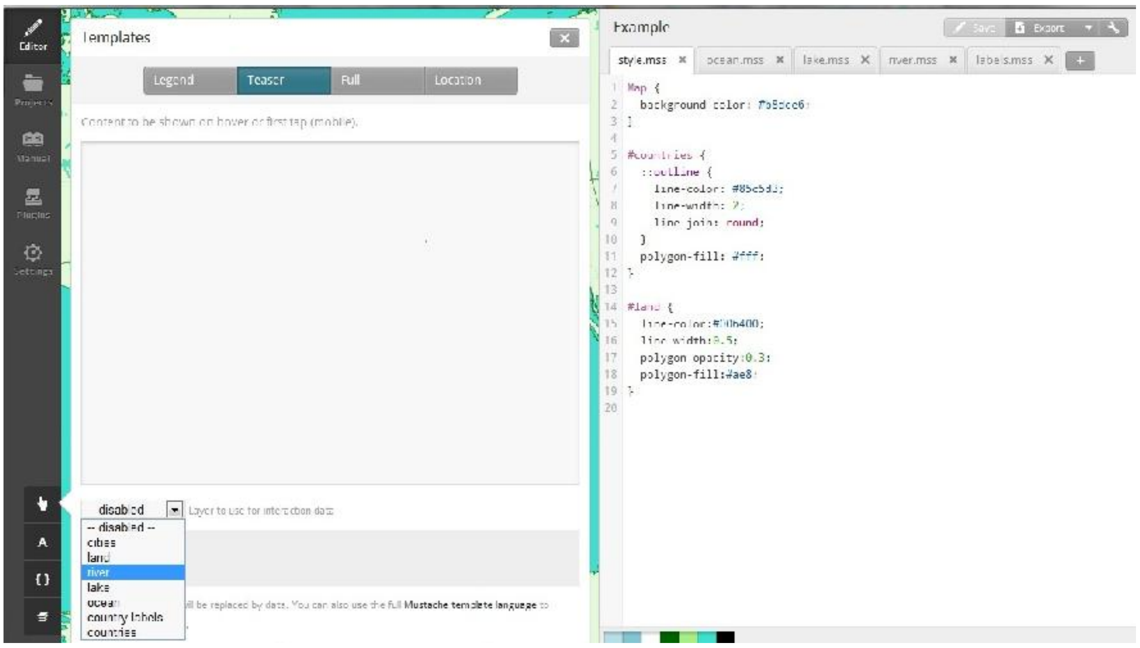
```
#cities {
  [SCALERANK<3][zoom>=4],
  [SCALERANK=3][zoom>=5],
  [SCALERANK=4][zoom>=5],
  [SCALERANK=5][zoom>=6],
  [SCALERANK=6][zoom>=6],
  [SCALERANK=7][zoom>=7],
  [SCALERANK=8][zoom>=7],
  [SCALERANK=9][zoom>=8],
  [SCALERANK=10][zoom>=8] {
    text-name:'[NAMEASCII]';  //text-name: '[ABBREV]';
    text-face-name:@sans_bold;
    text-fill:#000; //text color, inside borders
    text-size:9;
  }
  //Depending on zoom level, we choose the size of the text
  [zoom=4] {
    [SCALERANK<3] { text-size: 12; }
  }
  [zoom=5] {
    [SCALERANK<3] { text-size: 13; }
    [SCALERANK=3] { text-size: 12; }
    [SCALERANK=4] { text-size: 11; }
  }
  [zoom=6] {
    [SCALERANK<3] { text-size: 14; }
    [SCALERANK=3] { text-size: 13; }
    [SCALERANK=4] { text-size: 12; }
```

If we want a layer to be on top of another, for example the names, so they are not half cover, we just have to select it and move it to the position where we want it.



```
#lake[zoom>=4][ScaleRank=6],
#lake[zoom>=5][ScaleRank=7],
#lake[zoom>=6][ScaleRank=8],
#lake[zoom>=7][ScaleRank=9] {
  ::outline { line-color:@water; }
  ::fill { polygon-fill:@water; }
}
```

To add a teaser, we press the image and we choose “teaser” in the templates. In the combox, we have to choose for what layer we want to do the teaser.



Content to be shown on hover or first tap (mobile).

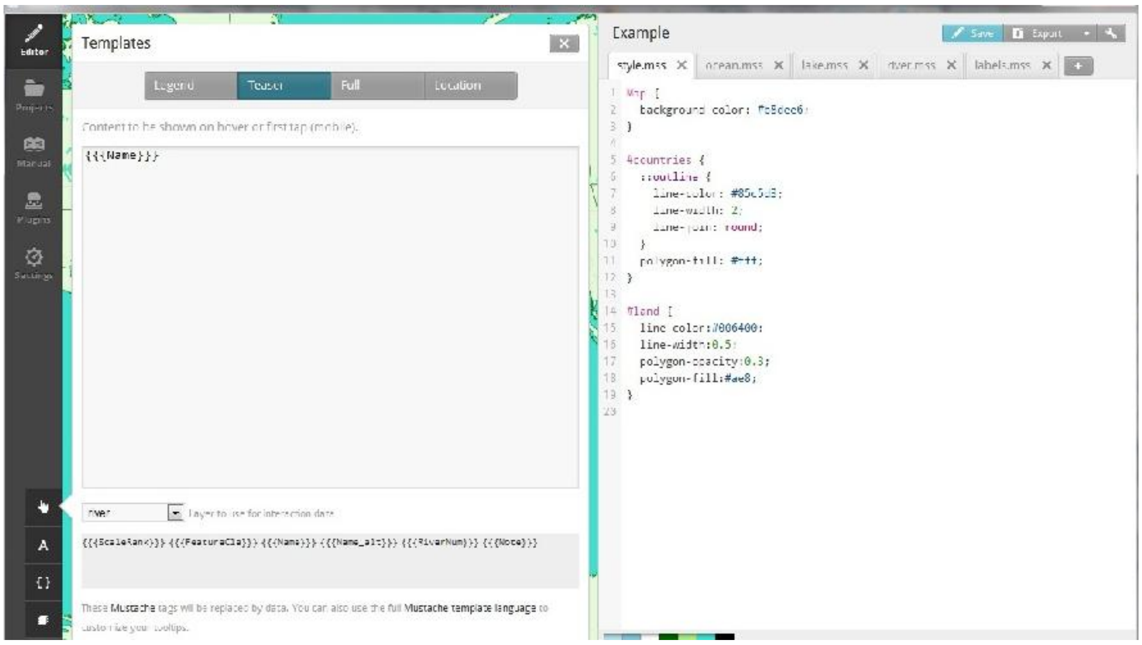
disabled ▾ Layer to use for interaction data

-- disabled --
cities
land
river
lake
ocean
country labels
countries

…ill be replaced by data. You can also use the full **Mustache template language** to

```
Map {
  background-color: #b8dee6;
}

#countries {
  ::outline {
    line-color: #85c5d3;
    line-width: 2;
    line-join: round;
  }
  polygon-fill: #fff;
}

#land {
  line-color:#00b400;
  line-width:0.5;
  polygon-opacity:0.3;
  polygon-fill:#ae8;
}
```

Once we select one, the different attributes we can use appear in the second text area. And in the first one, we can write the text we want to show in the map when the mouse rolls a concrete element. We can show one or more element, and a normal sentence.



Content to be shown on hover or first tap (mobile).

```
{{{Name}}}
```

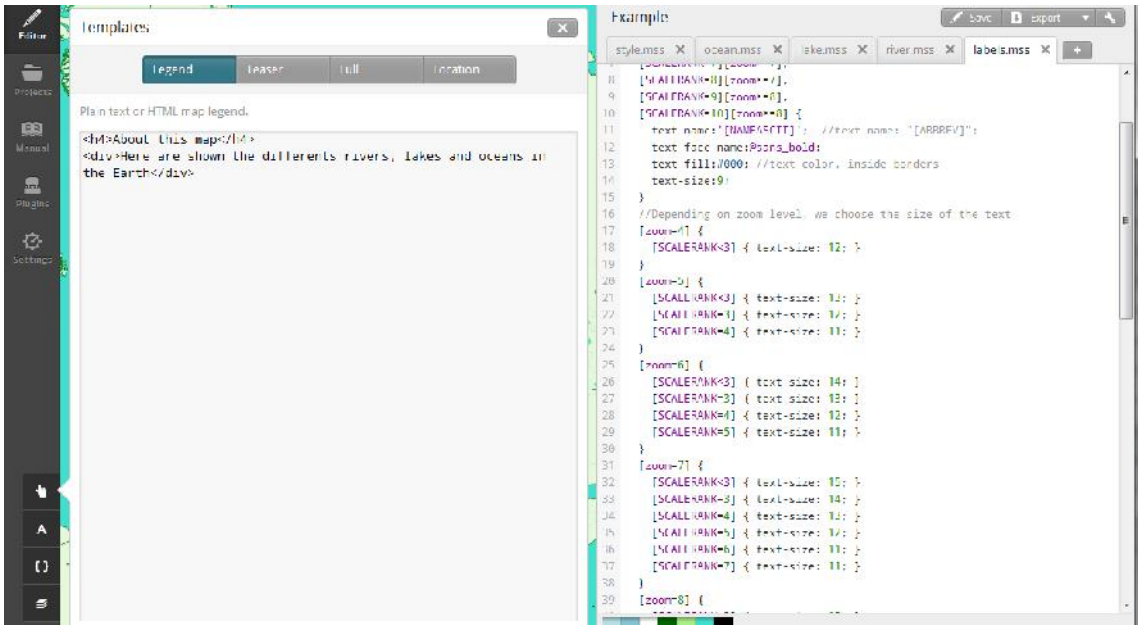
river ▾ Layer to use for interaction data

```
{{{ScaleRank}}} {{{FeatureCla}}} {{{Name}}} {{{Name_alt}}} {{{RiverNum}}} {{{Note}}}
```

These **Mustache** tags will be replaced by data. You can also use the full **Mustache template language** to customize your tooltips.

```
Map {
  background-color: #b8dee6;
}

#countries {
  ::outline {
    line-color: #85c5d3;
    line-width: 2;
    line-join: round;
  }
  polygon-fill: #fff;
}

#land {
  line-color:#806400;
  line-width:0.5;
  polygon-opacity:0.3;
  polygon-fill:#ae8;
}
```

To create a legend for the map, we press the image and we choose “legend” in the templates. We can add normal text or html formatted text. Te legend will appear down in the right.



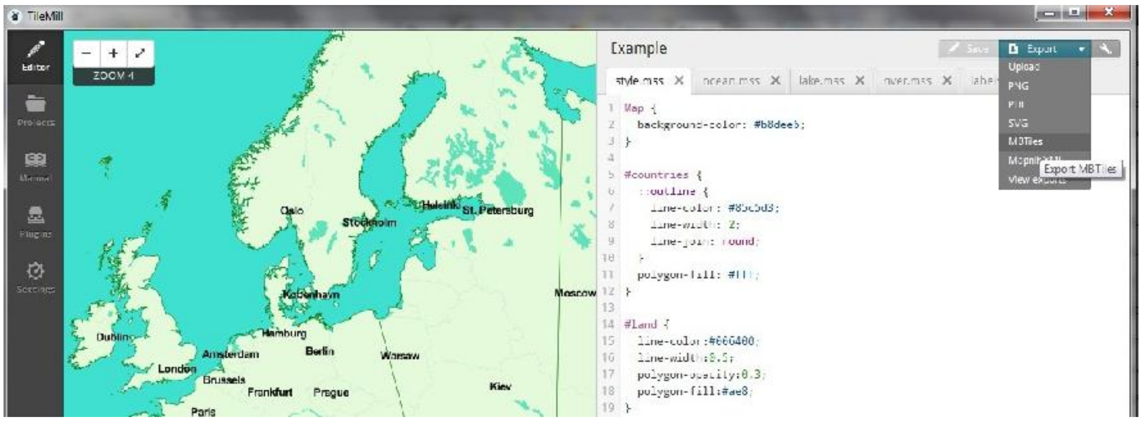
Plain text or HTML map legend.

```
<h4>About this map</h4>
<div>Here are shown the differents rivers, lakes and oceans in
the Earth</div>
```

```
[SCALERANK=8][zoom>=7],
[SCALERANK=9][zoom>=8],
[SCALERANK=10][zoom>=8] {
  text-name:'[NAMEASCII]';  //text-name: '[ABBREV]';
  text-face-name:@sans_bold;
  text-fill:#000; //text color, inside borders
  text-size:9;
}
//Depending on zoom level, we choose the size of the text
[zoom=4] {
  [SCALERANK<3] { text-size: 12; }
}
[zoom=5] {
  [SCALERANK<3] { text-size: 13; }
  [SCALERANK=3] { text-size: 12; }
  [SCALERANK=4] { text-size: 11; }
}
[zoom=6] {
  [SCALERANK<3] { text-size: 14; }
  [SCALERANK=3] { text-size: 13; }
  [SCALERANK=4] { text-size: 12; }
  [SCALERANK=5] { text-size: 11; }
}
[zoom=7] {
  [SCALERANK<3] { text-size: 15; }
  [SCALERANK=3] { text-size: 14; }
  [SCALERANK=4] { text-size: 13; }
  [SCALERANK=5] { text-size: 12; }
  [SCALERANK=6] { text-size: 11; }
  [SCALERANK=7] { text-size: 11; }
}
[zoom=8] {
```

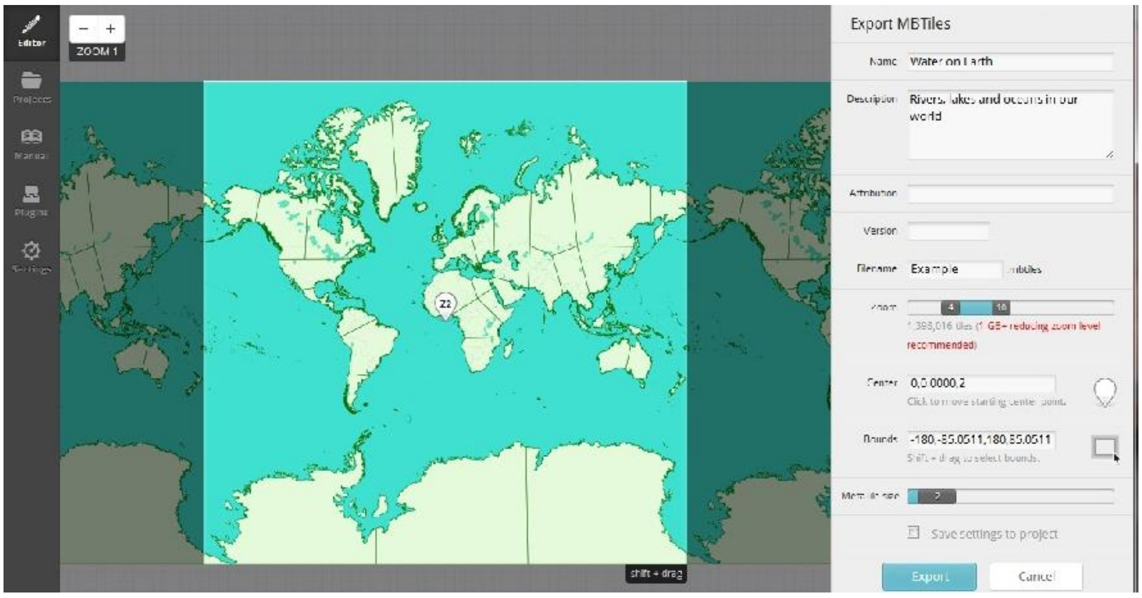
Export as MBTiles

To export the map, we press the button and select the kind of file we want to get. In this case, MBTiles.



```
Map {
  background-color: #b8dee6;
}

#countries {
  ::outline {
    line-color: #85c5d3;
    line-width: 2;
    line-join: round;
  }
  polygon-fill: #fff;
}

#land {
  line-color:#666400;
  line-width:0.5;
  polygon-opacity:0.3;
  polygon-fill:#ae8;
}
```

We can choose a name, a description, the attribution and the version. And the filename is the one we set at the beginning. An important thing here is the zoom. By default, the zoom is the complete range of values, which could take years in generate the map. We can change the minimum and the maximum, and we can observe how many tiles will be generated and how much space will be needed. We can also change the part of the map we want to generate, maybe we are only interested in a country or a city. Of course, this would also minimize the number of tiles that will be generated. And we can select which point we want to see when we open the map. Then we press "Export".



Export MBTiles

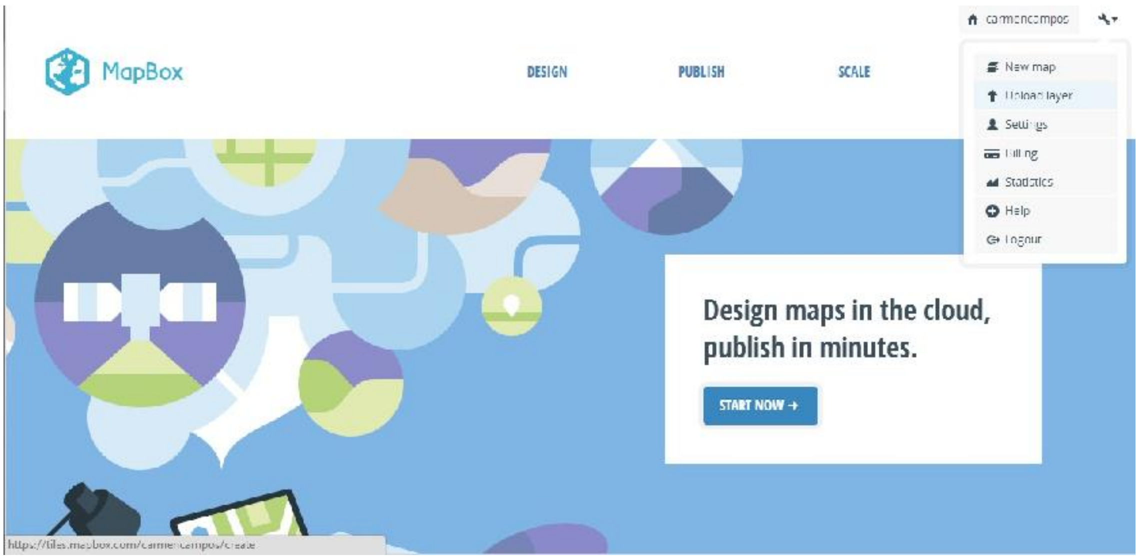
Name: Water on Earth
Description: Rivers, lakes and oceans in our world
Attribution:
Version:
Filename: Example .mbtiles
Zoom: 4 – 16
1,398,016 tiles (1 GB+ reducing zoom level recommended)
Center: 0,0,0000,2
Click to move starting center point.
Bounds: -180,-85.0511,180,85.0511
Shift + drag to select bounds.
Metatile size: 2
Save settings to project

Once it is finish, we press the button “Save MBTiles”.

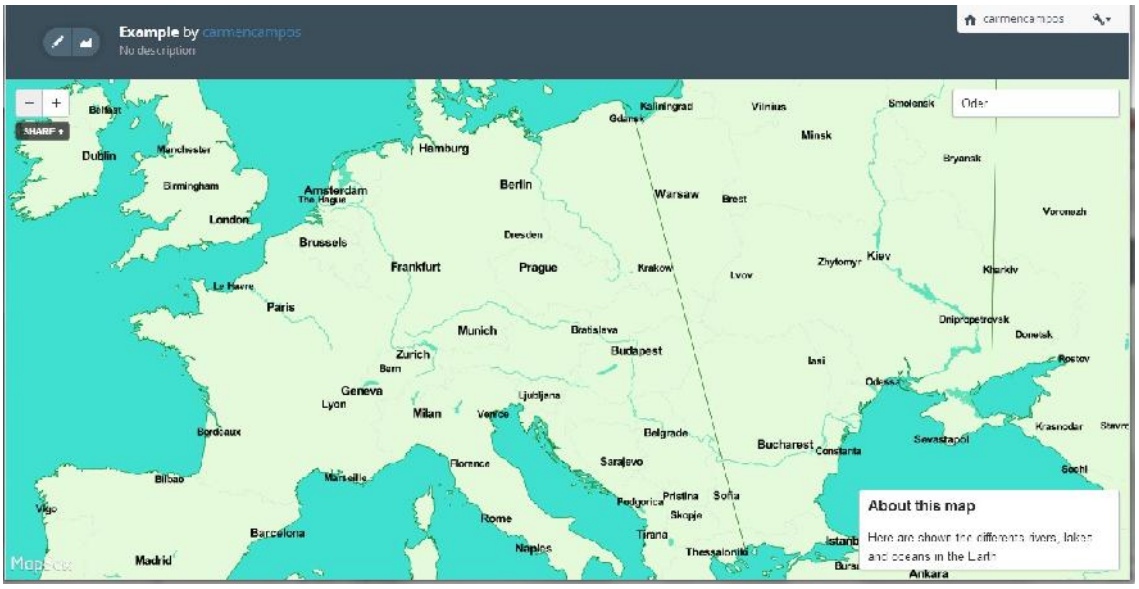
Publish to Mapbox

To publish our map in Mapbox (http://mapbox.com/) we need to create an account and to log in.

We have to press the tool image and select “Update layer” option.



Then we just select a file in our computer and press “Upload file” button. The file size must be less than 5GB, but also less than the capacity we have available in our account.



Publish map using leaflet

1. We need to have a server installed in our system.
2. We create a folder to work with leaflet in the appropriate folder in the server, the one where the server accesses to show the files.
3. We add the server code to read MBTiles to this folder.
4. We add our index.html page to this folder.
5. We add our {name-of-the-map}.mbtiles to this foldel.
6. We run the server.
7. We go to localhost/{name-of-the-folder}

In our index.html page we have to write something like this:

To show the layer using the map hosted in mapbox.
In general would be http://{s}.tiles.mapbox.com/v3/{user}.{name-of-the-map}/{z}/{x}/{y}.png
var hostedTiles = new
L.tileLayer('http://{s}.tiles.mapbox.com/v3/carmencampos.example/{z}/{x}/{y}.png'

To show the layer using the local map.
In general would be {name-of-the-file-to-extract-the-map}.php?db={name-of-the-map}.mbtiles&z={z}&x={x}&y={y}
var mbTiles = new L.tileLayer('mbtiles.php?db=example.mbtiles&z={z}&x={x}&y={y}'

Publish map using openlayers

1. We need to have a server installed in our system.
2. We create a folder to work with leaflet in the appropriate folder in the server, the one where the server accesses to show the files.
3. We add the server code to read MBTiles to this folder.
4. We add our index.html page to this folder.
5. We add our {name-of-the-map}.mbtiles to this folder.
6. We run the server.
7. We go to localhost/{name-of-the-folder}

In our index.html page we write something like this:

To show the layer using the map hosted in mapbox
In general would behttp://a.tiles.mapbox.com/v3/{user}.{name-of-the-map}/${z}/${x}/${y}.png
var hostedTiles = new OpenLayers.Layer.XYZ("Hosted Tiles",
["http://a.tiles.mapbox.com/v3/carmencampos.example/${z}/${x}/${y}.png",
"http://b.tiles.mapbox.com/v3/carmencampos.example/${z}/${x}/${y}.png",
"http://c.tiles.mapbox.com/v3/carmencampos.example/${z}/${x}/${y}.png",
"http://d.tiles.mapbox.com/v3/carmencampos.example/${z}/${x}/${y}.png"],

Create TMS layer using MBTiles sqlite database
To show the layer using the local map
var mbTiles = new OpenLayers.Layer.TMS("Local MBTiles File", "mbtiles.php",

example.mbtiles name of the file where our map is stored

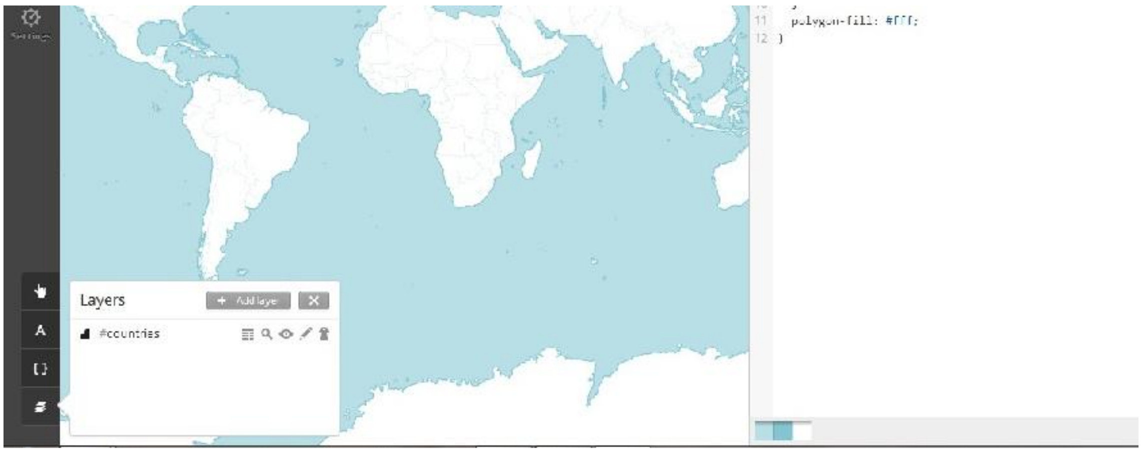
```
function mbtilesURL (bounds) {
        var db = "example.mbtiles";
        var res = this.map.getResolution();
        var x = Math.round ((bounds.left - this.maxExtent.left) / (res * this.tileSize.w));
        var y = Math.round ((this.maxExtent.top - bounds.top) / (res * this.tileSize.h));
        var z = this.map.getZoom();
        return this.url+"?db="+db+"&z="+z+"&x="+x+"&y="+((1 << z) - y - 1);
    }
```

3.2 TileMill: Points of Interest in Switzerland

In this document is showed how to use TileMill to create a base layer with real information about Switzerland. The first steps about how to create the project and start the map are the same as in the initial example.

It will be useful to take a look to TileMill CartoCSS reference. It is accessible by clicking the {} button, and there is an online CartoCSS reference by MapBox.

Once we have the map, we add the layers. As it is explained in the initial example, we click in the "Add Layer" button and we select the datasource path. We could change the id that appears by default and we can add a class, to write the css.



```
  polygon-fill: #fff;
}
```

Once we have added the datasource of the layer, we can choose if we want a style file to be generated by default. In this case, the style will be the necessary to show the data of the layer in the map. To do that, we press the "Save & Style" button.

If we prefer to generate our own style, we just press the "Save" button.

There are different kinds of layers, which will we treated differently when we write the necessary css. Once we have added the layers, we can know the type of every layer. In this case, #pois will be an example of point.

A point is a single 'spot' in space. It has no dimension, i.e. no length, width, or height. Points are typically defined by a set of coordinates, also known as a coordinate tuple.

A line is built of points. A sequence of points will form a line, or linestring.

A polygon is similar to a line, except that its start and end points are the same. In effect, polygons are closed loops. Polygons form an area, and can have loops cut out of them.

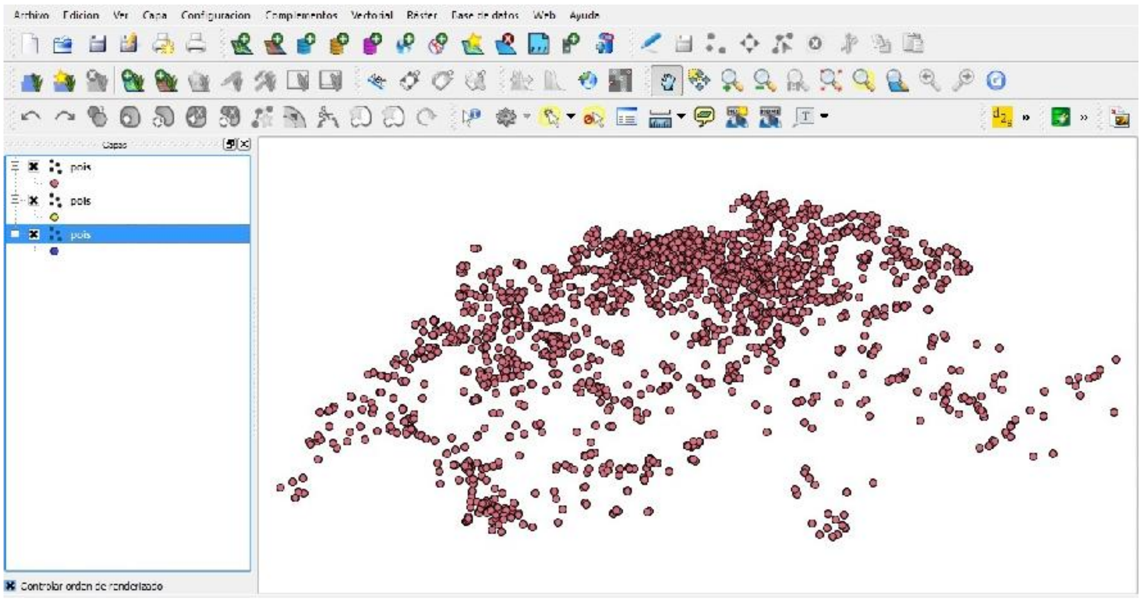
- linestring: the only css code we need is the color in which we want to show it
    - ::outline { line-color: #800080; }
- polygon: the css code we need is the color in which we want to show the border and the inside
    - ::outline { line-color: #00CED1; }
    - ::fill { polygon-fill: #00CED1; }

- point: the css code we need is the color in which we want to show
    - marker-fill: #FFFF00;

  point: and we can change the size of the point
    - marker-width: 3;

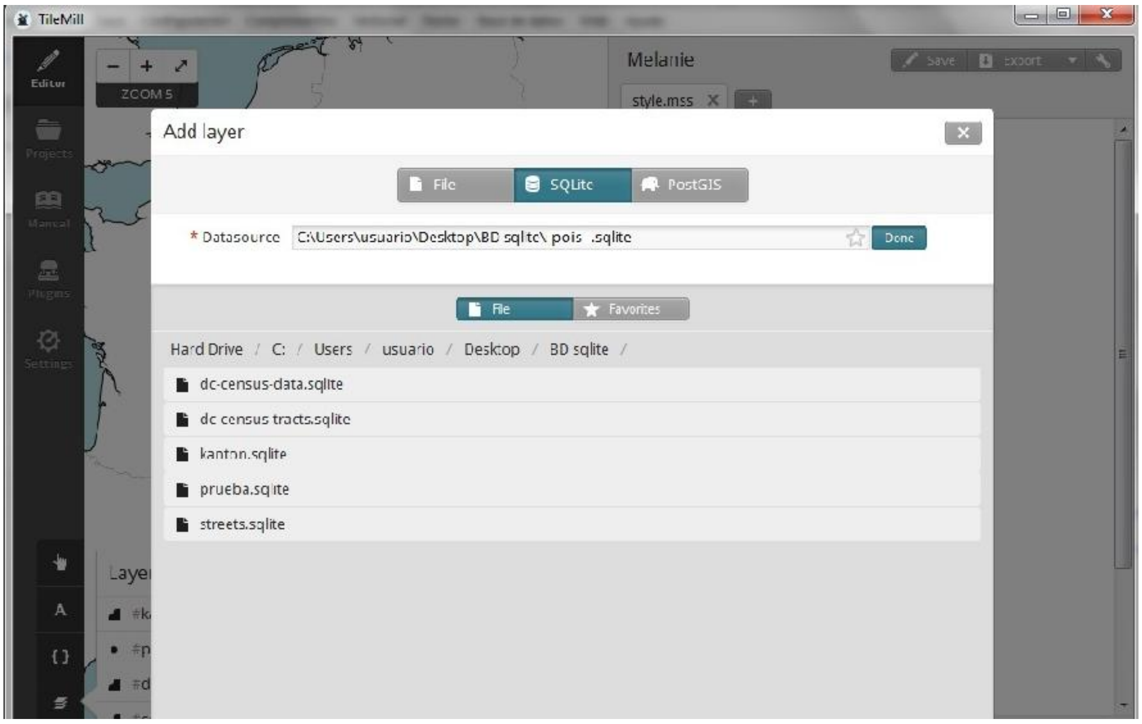
We can also add a layer using a SQLite databases as geographic datasource. SQLite files can be edited with any SQLite client, including free GIS tools like Quantum GIS.
SQLite files are an ideal alternative to shapefiles because they consist of just one file, making them very easy to share, and just like shapefiles, SQLite databases can store geographic features along with non-geographic attributes about those features.

To create the SQLite database using Quantum GIS we need a shapefile with our data. For example, to create a database with the pois in Switzerland we need a folder containing the .dbf, .prj, .shp, .xml and .shx files about the pois in Switzerland. Once we have our shapefile, we just need to add the folder in Quantum GIS, just drag it to the Layers window.


Archivo Edicion Ver Capa Configuracion Complementos Vectorial Ráster Base de datos Web Ayuda
Capas
pois
pois
pois
Controlar orden de renderizado

Right-click on our layer named pois within the Layers window and click “Save As…” . Select SQLite for the Format, and enter the name. You can browse to select the directory to save the file.

Once in TileMill, we click “Add layer” and we change the add layer type to SQLite. In datasource we have to add the .sqlite file we have just created.



For the Table or subquery field, enter (SELECT * from pois). This is a query to select the data from table pois within your SQLite database. This field acts as a subquery so the information must be entered in a subquery fashion.

Select the Spatial Reference System (SRS) for your feature. This will be the projection your data is. TileMill can often autodetect this value. But it can also be obtained from here.

At this point, you can already click the "Save & Style" to add your layer with the default CartoCSS settings and preview the result.

The real power of having SQLite in TileMill is the ability to join two or more SQLite databases together. This feature allows you supplement your geographic data with data from other sources and use it in your stylesheets and tooltips.

When joining multiple SQLite files, you will have one database that contains feature geometries and additional database can add more attributes about those features. To complete the join, the databases need to share a common key or ID.

We can edit the previous layer to add a join query to another database. In order to do that, in the "Attach DB" field we write where our second .sqlite database is stored. And into the "Table or subquery" field we could put something like this:

(SELECT * FROM pois JOIN interesting_places on pois.city = interesting_places.city && interesting_places.city=Bern)

When we have finished writing the css code for every layer, we could add a legend to show some extra information about the layer.

We could also add a Teaser: a piece of information that will appear when we hover with the mouse in a point. Or a Full: a piece of information that will appear when we click with the mouse in a point. Here we can show normal text, or specific information about the particular point.

The final step will be to export as mbtiles. Remember not to choose to many zoom levels or the size of the map will be too big. You can also restrict the part of the world that is going to be shown to Switzerland and a little bit around it.

4. Bibliography

[1] Geospatial Data Abstraction Library, translator for geospatial formats,
URL http://www.gdal.org/

[2] ArcGIS, to work with maps and geography information,
URL http://resources.arcgis.com/en/help/getting-started/

[3] CartoDB, for creating dynamic maps,
URL http://developers.cartodb.com/

[4] TileMill, how to create interactive maps,
URL http://www.mapbox.com/tilemill/docs/manual/

[5] Quantum GIS, how to use raster data in Geographical Information Systems,
URL http://www.qgis.org/en/documentation/manuals.html

[6] Geographic Information Systems,
URL http://www.esri.com/what-is-gis/overview#overview_panel

[7] GDAL2Tiles to generate tiles from a raster image,
URL https://developers.google.com/kml/articles/raster

[8] Web Map Tile Service specification,
URL http://www.opengeospatial.org/standards/wmts

[9] GeoTIFF file format specification,
URL http://www.remotesensing.org/geotiff/spec/contents.html

[10] Map projections in different coordinate systems,
URL http://kartoweb.itc.nl/geometrics/Introduction/introduction.html

[11] MapTiler, graphical interface for GDAL2Tiles,
URL http://www.maptiler.org/

[12] Leaflet, JavaScript library to create maps,
URL http://leafletjs.com/

[13] Video tutorial: how to use TileMill,
URL http://fuzzytolerance.info/blog/screencast-11-a-quick-run-through-tilemill/

[14] Creating and serving thematic maps,
URL http://blog.thematicmapping.org/

[15] MapProxy, the part related with caching and MBTiles
URL http://mapproxy.org/docs/1.5.0/tutorial.html

[16] MapProxy configuration examples,
https://github.com/mapproxy/mapproxy/blob/master/doc/configuration_examples.rst

[17] MapProxy tutorial,
URL http://es.scribd.com/doc/102823641/Mapproxy-Tutorial

[18] Open Street Map wiki
URL http://wiki.openstreetmap.org/wiki/Main_Page

[19] Geography information about maps,
URL http://www.mapsofworld.com/

[20] MapProxy command line tool,
URL https://github.com/mapproxy/mapproxy/blob/master/doc/mapproxy_util.rst

[21] PHP server to extract tiles from MBTiles database,
URL https://github.com/bmcbride/PHP-MBTiles-Server

[22] OpenLayers, JavaScript library to create maps,
URLhttp://trac.osgeo.org/openlayers/wiki/Documentation

[23] Spherical Mercator projection,
URL http://docs.openlayers.org/library/spherical_mercator.html

[24] Map coordinate reference systems,
URL http://help.maptiler.org/coordinates/

[25] Python, programming language,
http://docs.python.org/2/tutorial/

[26] Python tutorial,
URL http://www.tutorialspoint.com/python/index.htm

[27] Differences between projections,
URL http://www.sharpgis.net/post/2007/05/05/Spatial-references2c-coordinate-systems2c-projections2c-datums2c-ellipsoids-e28093-confusing.aspx

[28] SQLite database,
URL http://www.sqlite.org/

[29] Wax, to add interactivity to the map,
URL http://www.mapbox.com/wax/

[30] Spherical Mercator projection,
URL http://alastaira.wordpress.com/2011/01/23/the-google-maps-bing-maps-spherical-mercator-projection/

[31] OpenLayers tutorial,
URL http://vasir.net/blog/openlayers/

[32] Zoom slider plugin for Leaflet,
URL https://github.com/kartena/Leaflet.zoomslider

[33] MBTiles specification,
URL https://github.com/mapbox/mbtiles-spec

[34] Using SQLite databases in Python,
URL http://zetcode.com/db/sqlitepythontutorial/

[35] Importing and exporting MBTiles format,
URL https://github.com/mapbox/mbutil

[36] Flask, a Python microframework,
URL http://flask.pocoo.org/

[37] TileStache, a Python server for map tiles,
URL http://tilestache.org/

[38] Quantum GIS, to work with map databases,
URL http://www.qgis.org/

[39] Tile Server in PHP,
URL https://github.com/klokantech/tileserver-php

[40] Extra features in OpenLayers,
URL https://github.com/developmentseed/openlayers_plus

[41] How to use Python in the web,
URL http://docs.python.org/2/howto/webservers.html

[42] Getting started with PostGIS
URL http://www.bostongis.com/PrinterFriendly.aspx?content_name=postgis_tut01

[43] Working with PostGIS layers in Quantum GIS
URL http://www.gistutor.com/quantum-gis/20-intermediate-quantum-gis-tutorials/34-working-with-your-postgis-layers-using-quantum-gis-qgis.html

[44] Bottle: Python web framework,
URL http://bottlepy.org/docs/dev/index.html

[45] UTFGrid specification,
URL https://github.com/mapbox/utfgrid-spec/

[45] JavaScript Object Notation, UTFGrid format,
URL http://en.wikipedia.org/wiki/JSON

[46] How to use JSON in Python,
URL http://docs.python.org/2/library/json.html

[47] UTGrid process,
URL http://www.mapbox.com/demo/visiblemap/

[48] Interactivity using GeoJSON or UTFGrid,
URL http://milkator.wordpress.com/2013/03/05/interactivity-using-geojson-vs-utfgrid-vs-mapbox/

[49] GeoJSON specification,
URL http://www.geojson.org/geojson-spec.html

[50] Shapefile format,
URL http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf

[51] PostGIS: spatial database extender for PostgreSQL
URL http://postgis.net/

[52] Serve MBTiles features with PHP script,
URL http://blog.carte-libre.fr/index.php?post/2012/02/12/Serve-all-MBTtile-features-with-PHP-script

[53] Vector tile tutorial,
URL https://github.com/NelsonMinar/vector-river-map

[54] HTML inside Python,
URL http://karrigell.sourceforge.net/en/htmlinsidepython.html

[55] mod_wsgi: Apache module for hosting Python applications,
URL https://code.google.com/p/modwsgi/

[56] unittest, Unite testing framework in Python,
URL http://docs.python.org/2/library/unittest.html

[57] How to test your code in Python,
URL http://docs.python-guide.org/en/latest/writing/tests.html

[58] Tile Map Service specification,
URL http://wiki.osgeo.org/wiki/Tile_Map_Service_Specification

[59] JSON with padding,
URL http://en.wikipedia.org/wiki/JSONP

[60] JSONP calls examples,
URL http://www.onlinesolutionsdevelopment.com/blog/web-development/javascript/jsonp-example/

[61] JSONP explained with a example,
URL http://www.jquery4u.com/json/jsonp-examples/

[62] Web Map Tile Service services,
http://resources.arcgis.com/en/help/main/10.1/index.html#//0154000003r6000000

[63] Coordinates, tile bounds and projections,
URL http://www.maptiler.org/google-maps-coordinates-tile-bounds-projection/

[64] Web Map Service specification,
URL http://www.opengeospatial.org/standards/wms

[65] Representational state transfer,
URL http://www.peej.co.uk/articles/rest.html

[66] Tiled vectors,
URLhttp://research.microsoft.com/enus/um/people/jelson/mapcruncher/tiledvectors/tiledvectorsdemo-1.1/description.html