Robust Private Web Maps with Open Tools and Open Data

Bas Lijnse
Netherlands Defence Academy
b.lijnse@mindef.nl
Radboud University Nijmegen
b.lijnse@cs.ru.nl

ABSTRACT

Crisis management information often has a geospatial dimension that allows it to be visualized on a map. As more and more systems are developed as web-based applications, maps have also become a common sight in such applications. The de-facto solution to add maps to web-based applications is to integrate a third-party service. For web-based crisis management information systems, this approach has two disadvantages. First, the third-party service must be available and reachable. Second, by using third-party services you implicitly share what you are viewing, with the risk of unintentionally exposing sensitive location information. In this *Tool Talks* paper, we show how to create a robust and private alternative for web-based maps using open source tools and open data.

Keywords

Web Maps, Web-Based Systems, Tools, Open Data, OpenStreetMap

INTRODUCTION

Many crisis coordination systems, such as incident response or command and control systems, deal with geospatial information. The stereotype image of a command centre typically features a map projected on a large screen. When designing such crisis management information systems, one of the important design decisions is to choose what tools to use for this geographic visualization.

A general long-time trend in the development of information systems is to move towards web-based systems. Modern web-browsers, protocols and standards are powerful enough to build every conceivable application. Even complex desktop applications such as office suites are now available as web-based versions. With more software development moving to web-development, it is safe to assume that a significant share of future crisis management systems will also be developed as web-based systems.

Integrating maps into web-based information systems is trivial. Search giants such as Google and Microsoft's Bing provide free application programming interfaces (API's) that let you integrate their maps easily. Additionally there are dedicated commercial providers such as Esri or Mapbox that have similar offerings but also offer more options, services and customizations. A well-known open-source alternative is OpenStreetMap (OpenStreetMap Contributors 2018b) that can be integrated just as easily.

Integrating these online third-party API's is a quick and easy solution for integrating a map into a web-based information system. It also has the added benefit of not having to manage your own map data. However, they also have some inherent properties that may be problematic for crisis management applications. The first and obvious property is that they create a dependency on an online third-party provider. Without an internet connection the maps do not work. This is not an issue for public online systems, which are not available in that case anyway. But for crisis management systems, that despite being web-based may run on a local network without a reliable connection to the internet, this can be a serious liability.

Another property that may be less obvious, is that using a third-party map provider is a potential privacy concern. Web-maps are typically not loaded all at once, but incrementally. By retrieving specific parts of a map, you implicitly share with the map-provider which geographic region you are looking at. Depending on the level of

network security, this information could be seen by additional third parties. How sensitive this information is depends on the specific context, but in crisis management applications one cannot simply assume that it is not sensitive at all. For example the OCHA policy paper "Humanitarianism in the Cyberwarfare Age" (Gilman and Baker 2014) states:

> Information about places and objects, such as pre-positioned stocks or medical facilities, is critical to a humanitarian response, and may be very sensitive.

Although this paper considers protection of personal information to be a more important concern, it still acknowledges the potential sensitivity of location information.

An alternative to using third-party map providers in your web applications is to serve your own map content. This approach enables several options to improve robustness. You can provide a map service inside a local area network, making it independent of internet connectivity. Moreovere, for situations where local network infrastrucure cannot be relied upon, it enables the possibility to bundle maps for offline use. When you use your own map service, accidental sharing of sensitive locations with a third-party is also not an issue, because no third-parties are involved.

Creating, and serving your own web maps is not as easy as integrating a Google Maps widget, but is not as difficult as you might expect. With many open sources of geographic information publicly available and a wealth of open-source tools for storing this information and rendering maps, this is a viable alternative to third-party map providers. You need a basic understanding of geographic information, but you do not need to be an expert in geographic information systems (GIS) expert to setup and use these tools.

For this *Tool Talks* paper we consider the situation in which you are only interested in a relatively small geographic area and do not need frequently updated maps. In this case the simplest tool-chain suffices: you prepare the maps once using a batch process, and integrate them in an application using a Javascript library and any basic web server.

The first time we applied this tool-chain was for the Incidone system (Lijnse et al. 2012). This was a prototype incident coordination system based on the operations of the Netherlands Coast Guard. This system was designed to be able to operate in a closed network without reliable internet connectivity. Online map services were therefore not an option.

Since this first prototype the tools that were used to create the such maps have evolved, but the basic process has stayed the same. In this paper we aim to explain the tool-chain and how to use it to create your own private maps. We aim for a high enough level of abstraction that it will stay applicable even as details of the specific tools continue to evolve. Additionally we provide a practical section that helps you set up the tool chain with the latest software versions (at the time of writing), and helps you understand the roles of various components.

We consider the main contributions of this paper to be:

- An explanation of the necessary tools and process to create your own private web-maps from open data.
- A practical guide to get started, with exercises to further understanding of the tool chain
- A convenient script that automates the process for a concrete example

Because this paper is a *Tool Talks* paper, it is organized as a practical guide: We start with an explanation of the necessary preliminary concepts of GIS and web-based maps in "Preliminary Concepts". This is followed by a detailed explanation of th tool chain in "The Tool-chain". We conclude with the practical exercises in "Exercises" and final remarks in "Conclusion".

PRELIMINARY CONCEPTS

To set up and use the tool chain we describe in this paper, you do not need to be a GIS expert. However, you need to have a basic understanding of how maps, and in particular web-based maps, are constructed. In this section we cover those basic concepts.

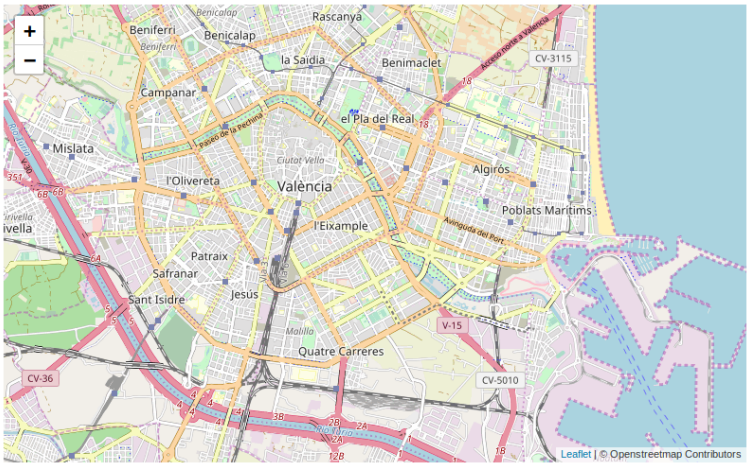

Figure 1. Example of web-based "Slippy map"

Map Basics: Layers, Points, Lines and Polygons

Essentially, a map is nothing more than the projection of *things* that have a location onto a flat two-dimensional surface. The basic data model that underlies any map is therefore fairly simple. The simplest thing you can put on a map is some point in space. Using points you can show the locations of objects. This representation is by definition an approximation, because a point is infinetely small and any object, even small ones take up some space. The next entity we can represent on a map is sequence of points, a line. Lines can be used to represent things like roads or waterways. Another interpretation of a sequence of points is a polygon. In a polygon the points define the outline of something. This can be used to represent areas: For example plots of land, or the outlines of buildings. Points, lines and polygons are the basic types of things you can put on a map. The final concept we need is a way to compose all these features. This is done using the concept of a layer. A layer is simply a collection of lines, points or polygons. A map is then defined as a stack of layers. Because the mapped objects are projected on a two-dimensional plane, a layer adds a third dimension. Layers define the order in which a map is drawn. Typically from bottom to top. This grouping makes it possible to specify that a set of lines that represent roads is drawn on top a set of polygons that define the underlying land areas. In interactive GIS applications the concept of layers also makes it easy to interactively show or hide collections of related features.

Storing Geographic Information

Because the data model is very simple you need little to store a geographic dataset. A small collection of points could easily be stored in a basic spreadsheet with three or four columns. One to hold a (unique) identifier and at least two more to represent the coordinates of the points. Depending on the type of data the coordinates can have two or three dimensions. The elevation of points is not always considered. Sometimes elevation has no meaning for that type of data, but for certain computations, such as determining an accurate distance between two points, the third dimension is necessary. One thing to note here is that there is more than one way to represent coordinates. There are different coordinate systems and to interpret a location correctly you need to know the coordinate system that was used. Luckily these coordinate reference systems are well standardized and documented, but you need to be careful to interpret the coordinates in the correct reference system.

A simple spreadsheet is enough to store small datasets, but for larger datasets dedicated file formats and relational databases with geographic extensions are commonly used. Such extensions add additional data types for storing coordinates in a normal relational database table and add support for querying based on geographic properties. In this paper we use two file formats and a relational database.

Because of the simple unified data model of layers, points, lines and polygons it is straightforward to create maps with multiple layers in which the data of the different layers comes from different data sources, even using different coordinate reference systems. As long as you can select coordinates and map them to a common reference systems you can combine layers in a single map. Many GIS tools allow you to combine information in this way.

Styling and Drawing maps

To draw a map from geographic data we need a program that takes all the geographical features as input and computes where and how they should be drawn on a two-dimensional canvas. To do so, a database with geographic

Figure 2. Zoom level 0: The whole earth on a single tile

features alone is not enough. Whether we use an interactive GIS or a batch program that renders a map to a file, we need to specify additional information. For lines, we have to decide what color, or shade of gray they have to be. We also have to know how thick they should be and what type of lines they should be. Dotted, dashed or solid for example. For points, we need to draw some kind of marker. This can be a simple dot, or a complicated image. Either way you need to specify what points will look like. For polygons, we also need to define how they will be drawn. This could again range from a simple solid fill color, to a complex shading or gradient. A final styling issue that we should mention is the styling of labels. Maps often contain textual labels on the map itself. Labels are used for things like street names, city names or other noteworthy features. A map rendering program needs to know how to draw them. It needs to know what fonts and colors to use, and where to align the labels in relation to the feature.

What is possible in terms of styling depends on the software you use, but to understand the map rendering toolchain it is important to know that map rendering typically needs these two orthogonal types of information. What features to draw on the map, and how to style them.

Slippy Maps

The previous sections have introduced concepts that apply to all types of maps. In this paper we are primarily interested in maps for web-based applications. The way these typically deal with maps is different from desktop GIS applications. Typical online maps you may be familiar with, such as Google maps or the `openstreetmap.org` home page, use a technique for creating maps called a slippy map. In these maps only a few layers of the map are dynamically rendered in the web browser. These layers contain the dynamic parts of the map, such as the locations of your search result, or a visualization of a proposed route. All other layers (landmasses, streets, buildings etc.) are rendered in aggregation by a server and downloaded by the browser as a collection of images. These images are called "tiles" and are typically 256 by 256 pixel images. The map application in the web browser only downloads the necessary tiles for the region of the map the user is looking at, and stitches them together to create a single bitmap layer. This so called "base layer" is an efficient way to offload the computationally expensive rendering to a centralized server. Another advantage of this approach is that the tiles do not need to be rendered every time a browser needs them, but can be cached as pre-rendered images. Tile images and the libraries and applications that use them use a standardized format. This format is based on a three-dimensional coordinate system. The first dimension is the zoom level. On the lowest zoom-level, level 0, the map of the entire earth fits on a single square tile (See figure 2). When zooming in a level, both the width and height of the map are doubled. This means that at zoom-level 1, the whole map fits on a 2x2 grid of 4 tiles. At zoom-level 2, a 4x4 grid of 16 tiles is used and so on. The remaining dimensions (x and y) are needed to indicate the column and row on the grid for the given zoom level. When requesting tiles from a server a standard pattern is used for URL's. Tile images are requested as URL's that end with `<zoom>/<x>/<y>.png`. For example the URL `http://tiles.example.com/4/3/2.png` refers to the tile at the second row in the third column at zoom level four. This standard pattern makes it possible to use standard web servers to serve the tiles as long as the images are organized in the right directory structure. When using the tiles, the Javascript slippy map libraries compute which tiles are needed to stitch together the piece of the map they want to display.

It is important to note that in this scheme, the number of tiles quadruples at every zoom level. This exponential growth means that at detailed zoom levels, the amount of images for the entire planet becomes unfeasible large to render in advance. For tile servers that provide such large numbers of tiles, such as the main OpenStreetMap tile

servers, specific web server software is used that renders tiles on demand, but caches tiles that have been requested before. When the area you want to cover is not that large, pre-rendering a collection of tiles is possible and is a simpler strategy.

A statically rendered base map usually serves as the lowest layer of a dynamic map. In an application that uses this map you typically can add dynamic layers that contain the the objects that are relevant to the application domain on top of it. All of the common slippy map libraries allow you to this.

THE TOOL-CHAIN

The goal of the tool-chain we explain in this paper is to create a pre-rendered tile-layer that can be used locally. We approach this by illustrating how to replicate a relevant subset of the collection of tiles that make up the OpenStreetMap "Standard Tile Layer" (OpenStreetMap Contributors 2018c). We largely use the same tools as are used by the main tile servers of `openstreetmap.org`. These official tiles are created using definitions from the openstreetmap-carto project (Allan et al. 2018). Because this is our starting point, we often refer to tools or resources found in the public repository of this project troughout this section.

In the remainder of this section, we explain the tasks that need to be completed to accomplish our goal without going into specifics. We do not show the exact details of the programs and scripts that need to be executed. These details can be found in "Appendix: Script",

Preparing Basic Infrastructure

To create the tile collection we do not use a single integrated tool. The tool-chain consists of different programs and scripts that tie them together. Most of those tools work on multiple operating systems and do not necessarily need to be deployed together on a single machine. For sake of simplicity we assume a single machine configuration running the Ubuntu Linux 18.04 operating system. On this Linux distribution most of the software we use is available in its standard package repositories. It is also the operating system on which we have tested and used the tool-chain.

Aquiring Geographic Data

The first thing we need is the geographic data that will be put on the map. There are various providers of open geographic data that publish free datasets. Because we are recreating the OpenStreetMap tiles, the primary data set we use is the OpenStreetMap database. The full database is available from `planet.openstreetmap.org`, but is fairly large (43GB at the time of writing). Depending on your application, you may not need data for the entire planet. In that case you can download an extract from a provider such as GeoFabrik (Geofabrik GmbH 2018) that offers many regional extracts.

To exactly recreate the standard OpenStreetMap tiles, just the OpenStreetMap data is not enough. We need several additional data sets from Natural Earth (*Natural Earth* 2018). These provide relatively static features such as as country borders and land masses. We also need a set of preprocessed data that is derived from the main OpenStreetMap data. These contain, for example, coast-line polygons that are simplified to create smoother maps at lower zoom levels. The openstreetmap-carto repository contains a script that automates downloading and indexing these additional data sets.

Aquiring Additional Style Resources

As explained in section "Preliminary Concepts", we also need styling definitions, marker images and fonts to be able to render the map. With the exception of the fonts, these are all included in the opentreetmap-carto repository. The fonts used in the style definitions should be installed on the operating system such that the map rendering tools can find them.

Storing and Preparing the Geographic Data

The tool that actually renders the map cannot work with the distributed form of the OpenStreetMap database directly. The planet dataset (or an extract) is published in a compressed format that is optimized for distribution. It is not structured to allow efficient querying of the data which is needed to create the map. To use the data it needs to be stored in a form that can be easily queried. The standard solution for doing this with openstreetmap data is to use the PostgreSQL database management system (The PostgreSQL Global Development Group 2018) with the PostGIS (PostGIS Contributors 2018) extension. Using a tool called "osm2pgsql" you can import an

openstreetmap dataset into a PostgreSQL database. When used on an empty database it will create a standardized database schema to import the data to. To create the standard tiles the osm2pgsql import tool is run with a customized script from openstreetmap-carto that filters and and transforms the data during import. Data from the data set that is not shown on the map is not even imported into the database. This is efficient, but also means that it limits your ability to customize the map later on.

The additional datasets are in "shape file" format and do not need additional processing. They can be used by the map rendering software directly.

Preparing the Map Style Configuration

The last piece of preparation that needs to be done is to create the configuration that will drive the map rendering process. Which layers will be drawn on the map, what data is needed for those layers and how the layer will look needs to be defined. The rendering tool, that is explained in the next section, relies on a single XML document as its configuration. For tiles as complex as the openstreetmap standard tiles, writing such a single document directly is not very user friendly. CartoCSS (MacWright et al. 2018) is a style language very similar to Cascading Style Sheets (CSS), the standard for styling web pages. The openstreetmap-carto project uses the CartoCSS language to define the style of the tiles in a modular collection of style files. A special compiler named "carto" is used to translate these CartoCSS styles into the XML document needed for rendering.

Rendering the map

Rendering the map, the creation of the collection of tile images, is done using a tool called "Mapnik" (Pavlenko 2018). Unlike the other tools in the chain, Mapnik is not a standalone program. Mapnik is a software library that you can use in your own programs. It provides interfaces for different programming languages, but we only need the interface for the Python programming language. More specifically, we customize a Python script from the OpenStreetMap project (OpenStreetMap Contributors 2018a) that uses Mapnik to render a collection of tiles. This script is exactly what we need to achieve our main goal. By default the script creates tiles for a number of hardcoded example regions. To use the script, the only additional step is to replace these regions by the regions that we want to create maps for.

Testing the Map

The last stage of the tool-chain is using the tiles in a web application. This requires two components. The first is a slippy map Javascript library that ties the dynamic data from your web application to a static tile-based layer. To test the maps we use Leaflet (Agafonkin 2018). This slippy map library is also used on the main `openstreetmap.org` website. Embedding a Leaflet map in a webpage is straightforward and the Leaflet website offers examples on how to create a map and add tile-based layers. The second component needed to test the map is a basic webserver to serve the tiles, the leaflet library and a web page that glues them together. Because the collection of tiles is just a directory structure of image files, it can be served by any web server software.

Using the Map

In the original Incidone use case, and in other applications where we used these tools, we used the iTask Framework. (iTasks Contributors 2018) This framework comes with a built-in web server that can serve the pre-rendered tiles. It also provides tight integration with the Leaflet library that enables the manipulation of the maps through an abstract datastructure without the need to program the low-level Javascript details. Similar integrations are available for other programming languages and web-application frameworks.

EXERCISES

The intend of this paper is to help you understand the tool-chain, and to enable you to customize it to the needs of your applications. In this section we provide some exercises to challenge you to explore and learn how to use the tools.

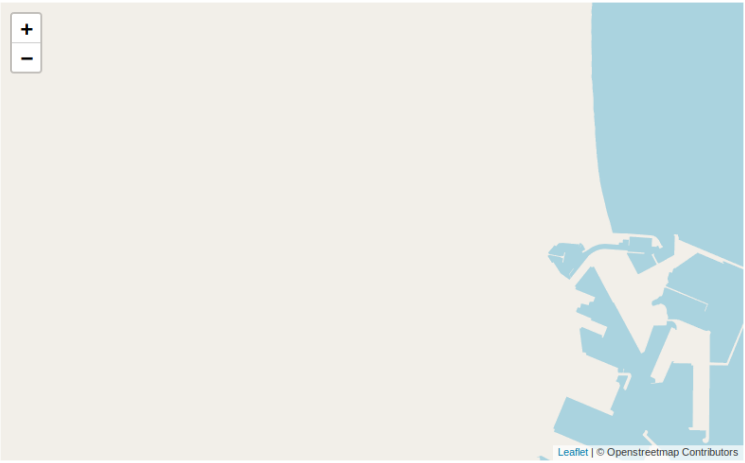

Figure 3. A mimimal rendering without the openstreetmap data

Set up the tools

The first exercise is to actually get the toolchain set up, to generate some tiles, and to view them in a simple web page. To help you get started we have created a basic shell script that executes all the necessary steps to set up the tools, to download the required data, and to render a first map. It is included as an appendix to this paper in "Appendix: Script".

With this script you should do the following:

- Prepare a machine running Ubuntu Linux 18.04,
- Download the script,
- Run the script, or to learn what's involved, execute all the steps in the script manually.

When succesful, you now have a basic set up for creating tile collections that you can change to your liking.

Simplify the Map

The standard OSM tiles contain many layers. Their definition may be overwhelming at first. In this exercise the goal is to reduce the map to the bare minimum and remove all the layers that use the OSM PostgreSQL database. To do this:

- Find the file that defines the layers of the openstreetmap-carto style,
- Figure out which layers are created from the shapefiles and which are created from the PostgreSQL database;
- Create a copy of the file with all the database-based layers removed;
- Create a Mapnik stylefile from this copy;
- Re-generate the tile collection with this alternative style.

When you test your map, you should now only see a basic map without any streets, buildings etc. as shown in figure 3.

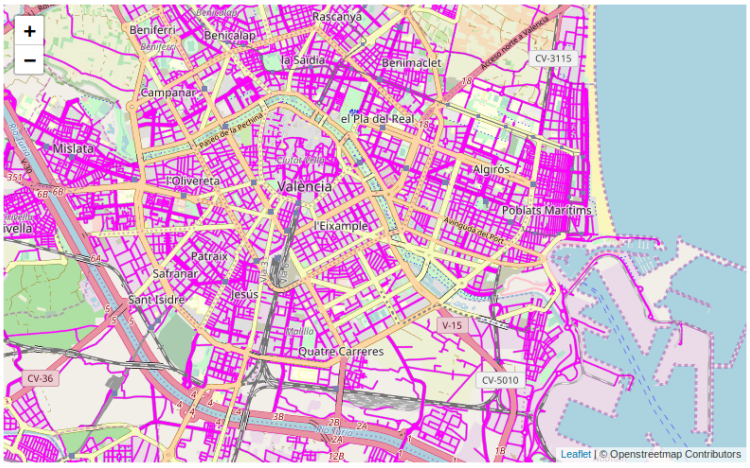

Figure 4. A brightly colored alternative map styling

Change the Map Style

The OpenStreetMap standard layer is a well designed map, but your own local maps can look any way you want. By using the openstreetmap-carto style as a starting point you can tune it to your liking. In this exercise the goal is to change the way a common feature is rendered.

- Find out where the color of city streets is defined;
- Change the color to something bright and recognizable;
- Re-create the Mapnik stylefile;
- Re-generate the tile collection with this alternative style.

Your map should now be rendered with your new favorite color, similar to figure 4.

Change the Map Data

When you have followed the script, your database will now contain (part of) an up-to-date OpenStreetMap data set. Having a local copy of this data allows you to modify it before you render your maps. In this exercise the goal is to explore the PostgreSQL database and modify or remove some of the data.

- When viewing your map, choose a point of interest. For example a streetname or building;
- Figure out how this feature is represented in the PostgreSQL database;
- Update or remove those records from the database;
- Re-generate the tile collection.

For example, in figure 5 you see a map of the city of Valencia that has been aptly renamed for the occasion.

RELATED WORK

The amount of open source tools for working with geographic information is very large. The tool chain presented in this paper is just one of many possible options for adding private maps to web applications. A large part of the specific setup of this toolchain is based on the installation instructions of openstreetmap-carto, because we chose to take the current version of the openstreetmap tiles as our starting point.

In this paper we explicitly focused on creating pre-rendered tile collections because of their simplicity, and because they are easy to deploy as bundled assets with a web application. If you are instead interested in creating a dedicated tile-server with on-demand rendering and caching of tiles, the “Manually building a tile server” guides from switch2osm.org (Fairhurst 2018) are a good starting point.

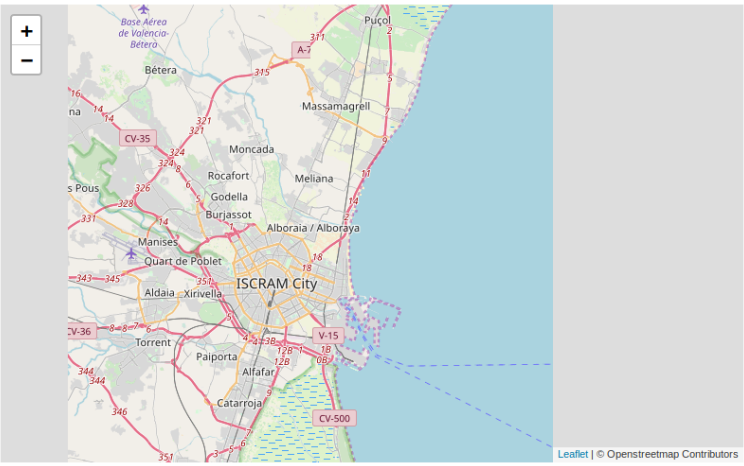

Figure 5. A custom rendering with a renamed city

The tool-chain we presented consists entirely of command line tools and scripts. This makes it very suitable for customization and automation on server machines. As an alternative, you could also use a desktop GIS application like QGIS (QGIS Contributors 2018) to define and style a map and use it to export tiles. Either through embedded scripting or by using a dedicated plugin.

This paper provides a pragmatic starting point for creating your own private maps from open data. We provide just enough background information to understand what the tools do, what their interfaces are, and how they depend on each other. It should be enough if all you need is a robust private replacement for popular third-party API's such as Google Maps. If you want to know more about cartography with open source tools, the website of the Open Source Geospatial Foundation (OSGeo 2018) is a good starting point.

CONCLUSION

In this paper we have shown the process of making your own collection of tile images from open data to use as a base layer in web-based maps. This approach enables the use of maps in web-based systems without relying on third-party online map service providers. This approach can therefore make the use of web-based maps both more private and more robust

The tool-chain to achieve this goal consists of all open source tools that are freely available. To use them together however, you need to have some background knowledge of GIS concepts thats is often implictly assumed. The aim of this paper is to explain the tool-chain, along with the necessary context, independent of specific details of the individual tools. By understanding the process and how the various pieces work together, it becomes possible to adapt to future changes in the specific tools.

Although we expect the tools to evolve, we have included a concrete example based on the tool chain at the time of writing. Additionally we have included several exercises to explore how the chain can be customized. Together with the explanation at the conceptual level, it should enable you to begin using robust private maps created from open data in your own web-based crisis management information systems.

ACKNOWLEDGMENTS

The authors of this paper are not affiliated with the authors of the tools and data sets that are being shown in this paper. We want to express our gratitude to all of these open source contributors for making their tools and data available publicly.

REFERENCES

Agafonkin, V. (2018). *Leaflet - a Javascript library for interactive maps*. URL: https://leafletjs.com/ (visited on 11/30/2018).

Allan, A., Melissen, M., Norman, P., Konieczny, M., Ko, D., Hormann, C., and Sommer, L. (2018). *OpenStreetMap Carto*. URL: https://github.com/gravitystorm/openstreetmap-carto (visited on 11/30/2018).

Fairhurst, R. (2018). *Manually building a tile server*. URL: https://switch2osm.org/manually-building-a-tile-server-18-04-lts/ (visited on 11/30/2018).

Geofabrik GmbH (2018). *OpenStreetMap Data Extracts*. URL: http://download.geofabrik.de/ (visited on 11/30/2018).

Gilman, D. and Baker, L. (Oct. 2014). "Humanitarianism in the Cyberwarfare Age - OCHA Policy Paper 11". In:

iTasks Contributors (2018). *iTask Framework*. URL: http://www.itasks.org/ (visited on 11/30/2018).

Lijnse, B., Jansen, J., and Plasmeijer, R. (Apr. 2012). "Incidone: A Task-Oriented Incident Coordination Tool". In: *Proceedings of the 9th International Conference on Information Systems for Crisis Response and Management, ISCRAM '12*. Ed. by L. Rothkrantz, J. Ristvej, and Z. Franco. Vancouver, Canada.

MacWright, T., Käfer, K., Ashton, A., Springmeyer, D., and Glanznig, M. (2018). *CartoCSS*. URL: https://github.com/mapbox/carto (visited on 11/30/2018).

Natural Earth (2018). URL: https://www.naturalearthdata.com/ (visited on 11/30/2018).

OpenStreetMap Contributors (2018a). *Generate tiles Python script*. URL: https://raw.githubusercontent.com/openstreetmap/mapnik-stylesheets/master/generate_tiles.py (visited on 11/30/2018).

OpenStreetMap Contributors (2018b). *OpenStreetMap Project*. URL: https://www.openstreetmap.org/ (visited on 11/30/2018).

OpenStreetMap Contributors (2018c). *Standard tile layer*. URL: https://wiki.openstreetmap.org/wiki/Standard_tile_layer (visited on 11/30/2018).

OSGeo (2018). *The Open Source Geospatial Foundation*. URL: https://www.osgeo.org/ (visited on 11/30/2018).

Pavlenko, A. (2018). *Mapnik - the core of geospatial visualization & processing*. URL: https://mapnik.org/ (visited on 11/30/2018).

PostGIS Contributors (2018). *PostGIS - Spatial and Geographic objects for PostgreSQL*. URL: https://postgis.net/ (visited on 11/30/2018).

QGIS Contributors (2018). *QGIS A Free and Open Source Geographic Information System*. URL: https://qgis.org/en/site/ (visited on 11/30/2018).

The PostgreSQL Global Development Group (2018). *PostgreSQL: The world's most advanced open source database*. URL: https://www.postgresql.org/ (visited on 11/30/2018).

APPENDIX: SCRIPT

This section contains the script used to setup and test the toolchain verbatim. It is publicly available online at https://gitlab.science.ru.nl/baslijns/web-maps-scripts.

```
#!/bin/bash
# This scripts sets up a toolchain for rendering tile collections for
# private web-maps.
# This script assumes execution as root user on an Ubuntu 18.04 system

### Aquire geodata ###

# Make sure basic utilities are installed
apt-get update
apt-get install -y sudo wget git python nodejs npm

# Clone the openstreetmap-carto git repository
# This will be the starting point for
git clone https://github.com/gravitystorm/openstreetmap-carto.git

# Download the map primary map data
wget http://download.geofabrik.de/europe/spain-latest.osm.pbf
mkdir -p openstreetmap-carto/data

# Download and index the additional map data as shapefiles
apt-get install -y mapnik-utils
cd openstreetmap-carto
python scripts/get-shapefiles.py
cd ..

# Install and configure a PostgreSQL RDBMS with GIS extensions
# Also create a database named 'gis' that will hold the OSM data
apt-get install -y postgis
/etc/init.d/postgresql start
sudo -u postgres createuser -s root
createdb gis
psql -d gis -c 'CREATE EXTENSION postgis; CREATE EXTENSION hstore;'

# Import the primary map data into the database
apt-get install -y osm2pgsql

cd openstreetmap-carto
osm2pgsql -G --hstore --style openstreetmap-carto.style --tag-transform-script openstreetmap-carto.lua -C 1600 -d gis data/spain-latest.osm.pbf
cd ..

### Creating the tiles ###

# Create the mapnik stylesheet using the carto compiler
npm -g install carto
apt-get install -y fonts-noto-cjk fonts-noto-hinted fonts-noto-unhinted ttf-unifont

cd openstreetmap-carto
carto project.mml > osm.xml
cd ..

# Install the mapnik library and the python bindings
apt-get install -y python-mapnik

# Download a script that renders batches of tiles
wget https://raw.githubusercontent.com/openstreetmap/mapnik-stylesheets/master/generate_tiles.py
# Remove the generation of tiles for the example cities in the script
# and disable strict loading of the osm.xml file
head -n 216 generate_tiles.py | sed "s/mapnik.load_map(self.m, mapfile, True)/mapnik.load_map(self.m, mapfile, False)/" > custom_generate_tiles.py
# Add a city of our own (Valencia) to be rendered in detail as an example
echo "    render_tiles((-0.49,39.37, -0.24, 39.56), mapfile, tile_dir, 1, 16, \"Valencia\")" >> custom_generate_tiles.py
# Run the script to render the tiles
mkdir -p tiles
MAPNIK_MAP_FILE=openstreetmap-carto/osm.xml MAPNIK_TILE_DIR=./tiles python custom_generate_tiles.py

### Testing the map ###

# Install and configure Nginx to serve a simple web-page with a Leaflet instance
apt-get install -y nginx
npm -g install leaflet

# Configure nginx
echo "server {" >> /etc/nginx/conf.d/privatemap.conf
echo "  listen 8000;" >> /etc/nginx/conf.d/privatemap.conf
echo "  location / { root $(pwd)/tiles; index index.html; }" >> /etc/nginx/conf.d/privatemap.conf
echo "  location /leaflet { alias /usr/local/lib/node_modules/leaflet/dist; }" >> /etc/nginx/conf.d/privatemap.conf
echo "}" >> /etc/nginx/conf.d/privatemap.conf

nginx; nginx -s reload

# Create a minimal html page that serves an instance of
# the leafletjs slippy map with our map added as tilelayer
echo "<html>" >> tiles/index.html
echo "<head>" >> tiles/index.html
echo "<link rel=\"stylesheet\" href=\"/leaflet/leaflet.css\">" >> tiles/index.html
echo "<script type=\"text/javascript\" src=\"/leaflet/leaflet.js\"></script>">> tiles/index.html
echo "</head>" >> tiles/index.html
echo "<body>" >> tiles/index.html
echo "<div id=\"map\" style=\"width: 100%; height: 100%;\"></div>" >> tiles/index.html
echo "<script type=\"text/javascript\">" >> tiles/index.html
echo "var m = L.map('map').setView( [39.4633, -0.3633], 13);" >> tiles/index.html
echo "L.tileLayer('http://localhost:8000/{z}/{x}/{y}.png', {attribution: '&copy; Openstreetmap Contributors'}).addTo(m);" >> tiles/index.html
echo "</script>" >> tiles/index.html
echo "</body>" >> tiles/index.html
echo "</html>" >> tiles/index.html

# Done
echo "You can view your map at http://localhost:8000/
```