pg_tileserv

Contents

Figure 1: Crunchy Spatial

pg_tileserv

pg_tileserv is a PostGIS-only tile server in Go. Strip away all the other requirements – it just has to take in HTTP tile requests and form and execute SQL. In a sincere act of flattery, the API and design look a lot like that of the Martin tile server.

This guide will walk you through how to install and use pg_tileserv for your spatial applications. The Usage section goes in-depth on how the service works.

We also include some basic examples of web maps that render tiles from the pg_tileserv application.

The purpose of pg_tileserv is to turn a set of spatial records into tiles, on the fly. The tile server reads two different layers of data:

- **Table layers** are what they sound like: tables and views in the database that have a spatial column with a spatial reference system defined on it.
- **Function layers** hide the source of data from the server, and allow the HTTP client to send in optional parameters to allow more complex SQL functionality. Any function of the form `function(z integer, x integer, y integer, ...)` that returns an MVT bytea result can serve as a function layer.

Dynamic Geometry Example

So far, all our examples have used simple SQL functions, but using the procedural PL/pgSQL language we can create much more interactive examples.

```
CREATE OR REPLACE
FUNCTION public.squares(z integer, x integer, y integer, depth integer defaul
RETURNS bytea
AS $$
DECLARE
    result bytea;
    sq_width float8;
    tile_xmin float8;
    tile_ymin float8;
    bounds geometry;
BEGIN
    -- Find the tile bounds
    SELECT ST_TileEnvelope(z, x, y) AS geom INTO bounds;
    -- Find the bottom corner of the bounds
    tile_xmin := ST_XMin(bounds);
    tile_ymin := ST_YMin(bounds);
    -- We want tile divided up into depth*depth squares per tile,
    -- so what is the width of a square?
    sq_width := (ST_XMax(bounds) - ST_XMin(bounds)) / depth;

    WITH mvtgeom AS (
        SELECT
            -- Fill in the tile with all the squares
            ST_AsMVTGeom(ST_MakeEnvelope(
                tile_xmin + sq_width * (a-1),
                tile_ymin + sq_width * (b-1),
                tile_xmin + sq_width * a,
                tile_ymin + sq_width * b), bounds),
            -- Each square gets a property that shows
            -- what tile it is a part of and what its sub-address
            -- in that tile is
            Format('(%s.%s,%s.%s)', x, a, y, b) AS tilecoord
        -- Drive the square generator with a two-dimensional
        -- generate_series setup
        FROM generate_series(1, depth) a, generate_series(1, depth) b
        )
    SELECT ST_AsMVT(mvtgeom.*, 'public.squares')
    -- Put the query result into the result variale.
    INTO result FROM mvtgeom;

    -- Return the answer
    RETURN result;
END;
$$
LANGUAGE 'plpgsql'
IMMUTABLE -- Same inputs always give same outputs
STRICT -- Null input gets null output
PARALLEL SAFE;

COMMENT ON FUNCTION public.squares IS 'For␣each␣tile␣requested,␣generate␣and␣
```

Dynamic Hexagons with Spatial join Example

Hexagonal tilings are popular with data visualization experts because they can be used to summarize point data without adding a visual bias to the output via different summary area sizes. They also have a nice "non-pointy" shape, while still providing a complete tiling of the plane.

When you want to provide a hexagonal summary of a data set at multiple scales, it presents an implementation problem: do you need to create a pile of hexagon tables, solely for the purpose of summary visualization?

The answer is no: you can generate your hexagons dynamically based on the scale of the requested map tiles.

Generate hexagons

The first challenge is that a hexagon tile set cannot be perfectly inscribed into a powers-of-two square tile set. That means that any given tile will contain some odd combination of full and partial hexagons. In order for the hexagons that straddle tile boundaries to match up, we need a hexagon tiling that is uniform over the whole plane.

So, our first function takes a "hexagon grid coordinate" and generates a hexagon for that coordinate. The size and location of that hexagon are controlled by the hexagon edge length for this particular tiling.

```
-- Given coordinates in the hexagon tiling that has this
-- edge size, return the built-out hexagon
CREATE OR REPLACE
FUNCTION hexagon(i integer, j integer, edge float8)
RETURNS geometry
AS $$
DECLARE
h float8 := edge*cos(pi()/6.0);
cx float8 := 1.5*i*edge;
cy float8 := h*(2*j+abs(i%2));
BEGIN
RETURN ST_MakePolygon(ST_MakeLine(ARRAY[
            ST_MakePoint(cx - 1.0*edge, cy + 0),
            ST_MakePoint(cx - 0.5*edge, cy + -1*h),
            ST_MakePoint(cx + 0.5*edge, cy + -1*h),
            ST_MakePoint(cx + 1.0*edge, cy + 0),
            ST_MakePoint(cx + 0.5*edge, cy + h),
            ST_MakePoint(cx - 0.5*edge, cy + h),
            ST_MakePoint(cx - 1.0*edge, cy + 0)
        ]));
END;
$$
LANGUAGE 'plpgsql'
IMMUTABLE
STRICT
PARALLEL SAFE;

SELECT ST_AsText(hexagon(2, 2, 10.0));

 POLYGON((20 34.6410161513775,25 25.9807621135332,
          35 25.9807621135332,40 34.6410161513775,
          35 43.3012701892219,25 43.3012701892219,
          20 34.6410161513775))
```

Find hexagon coordinates within the map tile

Now we need a function that, given a square input (a map tile), can figure out all the hexagon coordinates that fall within the tile. Again, the edge size of the hexagon tiling determines the overall geometry of the hex tiling. More than one hexagon will be required most times, so this is a set-returning function.

```
-- Given a square bounds, find all the hexagonal cells
-- of a hex tiling (determined by edge size)
-- that might cover that square (slightly over-determined)
CREATE OR REPLACE
FUNCTION hexagoncoordinates(bounds geometry, edge float8,
                            OUT i integer, OUT j integer)
RETURNS SETOF record
AS $$
    DECLARE
        h float8 := edge*cos(pi()/6);
        mini integer := floor(st_xmin(bounds) / (1.5*edge));
        minj integer := floor(st_ymin(bounds) / (2*h));
        maxi integer := ceil(st_xmax(bounds) / (1.5*edge));
        maxj integer := ceil(st_ymax(bounds) / (2*h));
    BEGIN
    FOR i, j IN
    SELECT a, b
    FROM generate_series(mini, maxi) a,
         generate_series(minj, maxj) b
    LOOP
        RETURN NEXT;
    END LOOP;
    END;
$$
LANGUAGE 'plpgsql'
IMMUTABLE
STRICT
PARALLEL SAFE;

SELECT * FROM hexagoncoordinates(ST_TileEnvelope(15, 1, 1), 1000.0);

    i    |   j
---------+-------
  -13358 | 11567
  -13358 | 11568
  -13357 | 11567
  -13357 | 11568
  -13356 | 11567
  -13356 | 11568
```

Generate hexagons that cover the map tile

Next, we need a function that puts the two parts together: with tile coordinates and edge size as input, generate the set of all the hexagons that cover the tile. The output here is basically a spatial table: a set of rows, each row containing a geometry (hexagon) and some properties (hexagon coordinates). This is the input we need for a spatial join.

```
-- Given an input ZXY tile coordinate, output a set of hexagons
-- (and hexagon coordinates) in web mercator that cover that tile
CREATE OR REPLACE
FUNCTION tilehexagons(z integer, x integer, y integer, step integer,
                      OUT geom geometry(Polygon, 3857), OUT i integer, OUT j
RETURNS SETOF record
AS $$
    DECLARE
        bounds geometry;
        maxbounds geometry := ST_TileEnvelope(0, 0, 0);
        edge float8;
    BEGIN
    bounds := ST_TileEnvelope(z, x, y);
    edge := (ST_XMax(bounds) - ST_XMin(bounds)) / pow(2, step);
    FOR geom, i, j IN
    SELECT ST_SetSRID(hexagon(h.i, h.j, edge), 3857), h.i, h.j
    FROM hexagoncoordinates(bounds, edge) h
    LOOP
        IF maxbounds ~ geom AND bounds && geom THEN
            RETURN NEXT;
        END IF;
    END LOOP;
    END;
$$
LANGUAGE 'plpgsql'
IMMUTABLE
STRICT
PARALLEL SAFE;
```

The function that the tile server actually calls looks like all other tile server functions: tile coordinates and optional parameter as input, bytea MVT as output.

```
-- Given an input tile, generate the covering hexagons,
-- spatially join to population table, summarize
-- population in each hexagon, and generate MVT
-- output of the result. Step parameter determines
-- how many hexagons to generate per tile.
CREATE OR REPLACE
FUNCTION public.hexpopulationsummary(z integer, x integer, y integer, step in
RETURNS bytea
AS $$
WITH
bounds AS (
    -- Convert tile coordinates to web mercator tile bounds
    SELECT ST_TileEnvelope(z, x, y) AS geom
),
rows AS (
    -- Summary of populated places grouped by hex
    SELECT Sum(pop_max) AS pop_max, Sum(pop_min) AS pop_min, h.i, h.j, h.geom
    -- All the hexes that interact with this tile
    FROM TileHexagons(z, x, y, step) h
    -- All the populated places
    JOIN ne_50m_populated_places n
    -- Transform the hex into the SRS (4326 in this case)
    -- of the table of interest
    ON ST_Intersects(n.geom, ST_Transform(h.geom, 4326))
    GROUP BY h.i, h.j, h.geom
),
mvt AS (
    -- Usual tile processing, ST_AsMVTGeom simplifies, quantizes,
    -- and clips to tile boundary
    SELECT ST_AsMVTGeom(rows.geom, bounds.geom) AS geom,
           rows.pop_max, rows.pop_min, rows.i, rows.j
    FROM rows, bounds
)
-- Generate MVT encoding of final input record
SELECT ST_AsMVT(mvt, 'public.hexpopulationsummary') FROM mvt
$$
LANGUAGE 'sql'
STABLE
STRICT
PARALLEL SAFE;

COMMENT ON FUNCTION public.hexpopulationsummary IS 'Hex␣summary␣of␣the␣ne_50m
```

Function Layer Detail JSON

In the detail JSON, each function declares information relevant to setting up a map interface for the layer.

Since functions generate tiles dynamically, the system cannot auto-discover properties such as extent, or center. However, the custom parameters as well as defaults can be read from the function definition and exposed in the detail JSON.

```
{
    "name" : "parcels_in_radius",
    "id" : "public.parcels_in_radius",
    "schema" : "public",
    "description" : "Given the click point (click_lon, click_lat) and radius,
    "minzoom" : 0,
    "arguments" : [
        {
            "default" : "-123.13",
            "name" : "click_lon",
            "type" : "double precision"
        },
        {
            "default" : "49.25",
            "name" : "click_lat",
            "type" : "double precision"
        },
        {
            "default" : "500.0",
            "type" : "double precision",
            "name" : "radius"
        }
    ],
    "maxzoom" : 22,
    "tileurl" : "http://localhost:7800/public.parcels_in_radius/{z}/{x}/{y}.pb
}
```

- description can be set using the COMMENT ON FUNCTION SQL command.
- id, schema, and name are the fully qualified name, schema, and function name, respectively.
- minzoom and maxzoom are the defaults as set in the configuration file.
- arguments is a list of argument names, with the data type and default value.

Function Layer Examples

Filtering example

This simple example returns a filtered subset of a table (ne_50m_admin_0_countries EPSG:4326). The filter in this case is the first letter of the name.

Note that the name_prefix parameter includes a **default value**: this is useful for clients (like the preview interface for this server) that read arbitrary function definitions and need a default value to fill into interface fields.

This example also uses ST_TileEnvelope(), a utility function only available in PostGIS 3.0 and higher. See the notes below for a workaround using custom functions.

```
CREATE OR REPLACE
FUNCTION public.countries_name(
            z integer, x integer, y integer,
            name_prefix text default 'B')
RETURNS bytea
AS $$
    WITH
    bounds AS (
      SELECT ST_TileEnvelope(z, x, y) AS geom
    ),
    mvtgeom AS (
      SELECT ST_AsMVTGeom(ST_Transform(t.geom, 3857), bounds.geom) AS geom,
        t.name
      FROM ne_50m_admin_0_countries t, bounds
      WHERE ST_Intersects(t.geom, ST_Transform(bounds.geom, 4326))
      AND upper(t.name) LIKE (upper(name_prefix) || '%')
    )
    SELECT ST_AsMVT(mvtgeom, 'public.countries_name') FROM mvtgeom;
$$
LANGUAGE 'sql'
STABLE
PARALLEL SAFE;

COMMENT ON FUNCTION public.countries_name IS 'Filters␣the␣countries␣table␣by␣
```

Some notes about this function:

- The ST_AsMVT() function uses the function name ("public.countries_name") as the MVT layer name. While this is not required, it allows clients that auto-configure to use the function name as the layer source name.
- In the filter portion of the query (i.e. in the WHERE clause), the bounds are transformed to the spatial reference of the table data (in this case, 4326) so that the spatial index on the table geometry can be used.
- In the ST_AsMVTGeom() portion of the query, the table geometry is transformed into Web Mercator (3857) to match the bounds and the *de facto* expectation that MVT tiles are delivered in Web Mercator projection.
- The LIMIT is hard-coded in this example. If you want a user-defined limit, you need to add another parameter to your function definition.
- The function "volatility" is declared as STABLE because within one transaction context, multiple runs with the same inputs will return the same outputs. It is not marked as IMMUTABLE because changes in the base table can change the outputs over time, even for the same inputs.
- The function is declared as PARALLEL SAFE because it doesn't depend on any global state that might get confused by running multiple copies of the function at once.

- For earlier versions of PostGIS, the following is an example of a custom function that emulates the behavior of ST_TileEnvelope(): sql CREATE OR REPLACE FUNCTION ST_TileEnvelope(z integer, x integer, y integer) RET

Spatial processing example

This example clips a layer of parcels (EPSG:26910) using a radius and center point, returning only the parcels in the radius, with the boundary parcels clipped to the center.

```
CREATE OR REPLACE
FUNCTION public.parcels_in_radius(
                    z integer, x integer, y integer,
                    click_lon float8 default -123.13,
                    click_lat float8 default 49.25,
                    radius float8 default 500.0)
RETURNS bytea
AS $$
    WITH
    args AS (
      SELECT
        ST_TileEnvelope(z, x, y) AS bounds,
        ST_Transform(ST_SetSRID(ST_MakePoint(click_lon, click_lat), 4326), 26
    ),
    mvtgeom AS (
      SELECT
        ST_AsMVTGeom(
            ST_Transform(
                ST_Intersection(
                    p.geom,
                    ST_Buffer(args.click, radius)),
                3857),
            args.bounds) AS geom,
        p.site_id
      FROM parcels p, args
      WHERE ST_Intersects(p.geom, ST_Transform(args.bounds, 26910))
      AND ST_DWithin(p.geom, args.click, radius)
      LIMIT 10000
    )
    SELECT ST_AsMVT(mvtgeom, 'public.parcels_in_radius') FROM mvtgeom
$$
LANGUAGE 'sql'
STABLE
PARALLEL SAFE;

COMMENT ON FUNCTION public.parcels_in_radius IS 'Given␣the␣click␣point␣(click_
```

Notes:

- The parcels are stored in a table with spatial reference system 3005, a planar projection.
- The click parameters are longitude/latitude, so in building a click geometry (ST_MakePoint()) to use for querying, we transform the geometry to the table spatial reference.
- To get the parcel boundaries clipped to the radius, we build a circle in the native spatial reference (26910) using the ST_Buffer() function on the click point, then intersect that circle with the parcels.

You can explore the contents of the tile server using:

- an HTML web interface for humans; and
- a JSON API for computers.

The JSON API is useful for clients that auto-configure based on the service metadata. In fact, the HTML web interface itself is an example of such an auto-configuring interface: it reads the JSON and uses that to set up the web map visualization and interface elements.

Web Interface

After start-up, you can connect to the server and explore the published tables and functions in the database via a web interface at:

- http://localhost:7800

Click the "preview" link of any of the layer entries to see a web map view of the layer. The "json" link provides a direct link to the JSON metadata for that layer.

Layers List

A top-level list of layers is available in JSON at:

- http://localhost:7800/index.json

The index JSON returns the minimum information about each layer.

```
{
    "public.ne_50m_admin_0_countries" : {
        "name" : "ne_50m_admin_0_countries",
        "schema" : "public",
        "type" : "table",
        "id" : "public.ne_50m_admin_0_countries",
        "description" : "Natural Earth country data",
        "detailurl" : "http://localhost:7800/public.ne_50m_admin_0_countries.
    }
}
```

- The detailurl provides more detailed metadata for table and function layers.
- The description field is read from the comment value of the table. To set a comment on a table, use the COMMENT command: sql COMMENT ON TABLE ne_50m_admin_0_countries IS 'This is my comment';

The basic principle of security is to connect your tile server to the database with a user that has just the access you want it to have, and no more.

Start with a new, blank user. A blank user has no select privileges on tables it does not own. It does have execute privileges on functions. However, the user has no select privileges on tables accessed by functions, so effectively the user will still have no access to data.

```
CREATE USER tileserver;
```

To support different access patterns, create different users with access to different tables/functions, and run multiple services, connecting with those different users.

Tables and Views

If your tables and views are in a schema other than public, you will have to also grant "usage" on that schema to your user.

```
GRANT USAGE ON SCHEMA myschema TO tileserver;
```

You can then grant access to the user one table at a time.

```
GRANT SELECT ON TABLE myschema.mytable TO tileserver;
```

Alternatively, you can grant access to all the tables at once.

```
GRANT SELECT ON ALL TABLES IN SCHEMA myschema TO tileserver;
```

Functions

As noted above, functions that access table data are effectively restricted by the access levels the user has to the tables the function reads. If you want to completely restrict access to the function, including visibility in the user interface, you can strip execution privileges from the function.

```
-- All functions grant execute to 'public' and all roles are
-- part of the 'public' group, so public has to be removed
-- from the executors of the function
REVOKE EXECUTE ON FUNCTION myschema.myfunction FROM public;
-- Just to be sure, also revoke execute from the user
REVOKE EXECUTE ON FUNCTION myschema.myfunction FROM tileserver;
```

By default, pg_tileserv will provide access to **only** those spatial tables and views that:

- your database connection has SELECT privileges for;
- include a geometry column;
- declare a geometry type; and,
- declare an SRID (spatial reference ID).

For example:

```
CREATE TABLE mytable (
    geom Geometry(Polygon, 4326),
    pid text,
    address text
);
GRANT SELECT ON mytable TO myuser;
```

To restrict access to a certain set of tables, use database security principles:

- Create a role with limited privileges
- Only grant SELECT to that role for tables you want to publish
- Only grant EXECUTE to that role for functions you want to publish
- Connect pg_tileserv to the database using that role

Table Layer Detail JSON

In the detail JSON, each layer declares information relevant to setting up a map interface for the layer.

```
{
    "id" : "public.ne_50m_admin_0_countries",
    "geometrytype" : "MultiPolygon",
    "name" : "ne_50m_admin_0_countries",
    "description" : "Natural Earth countries",
    "schema" : "public",
    "bounds" : [
        −180,
        −89.9989318847656,
        180,
        83.599609375
    ],
    "center" : [
        0,
        −3.19966125488281
    ],
    "tileurl" : "http://localhost:7800/public.ne_50m_admin_0_countries/{z}/{x}
    "properties" : [
        {
            "name" : "gid",
            "type" : "int4",
            "description" : ""
        },{
            "name" : "featurecla",
            "description" : "",
            "type" : "varchar"
        },{
            "description" : "",
            "type" : "varchar",
            "name" : "name"
        },{
            "type" : "varchar",
            "description" : "",
            "name" : "name_long"
        }
    ],
    "minzoom" : 0,
    "maxzoom" : 22
}
```

- id, name, and schema are the fully qualified, table, and schema name of the database table.
- bounds and center give the extent and middle of the data collection, in geographic coordinates. The order of coordinates in bounds is [minlon, minlat, maxlon, maxlat]. The order of coordinates in center is [lon, lat].
- tileurl is the standard substitution pattern URL consumed by map clients like Mapbox GL JS and OpenLayers.

- properties is a list of columns in the table, with their data types and descriptions. The description field can be set using the COMMENT SQL command, for example:

```
COMMENT ON COLUMN ne_50m_admin_0_countries.name_long IS 'This␣is␣the␣long␣nam
```

Table Tile Request Customization

Most developers will use the tileurl as is, but it's possible to add parameters to the URL to customize behaviour at run time:

- limit controls the number of features to write to a tile. The default is 50000.
- resolution controls the resolution of a tile. The default is 4096 units per side for a tile.
- buffer controls the size of the extra data buffer for a tile. The default is 256 units.
- properties is a comma-separated list of properties to include in the tile. For wide tables with large numbers of columns, this allows a slimmer tile to be composed.

For example:

```
http://localhost:7800/public.ne_50m_admin_0_countries/{z}/{x}/{y}.pbf?limit=1
```

We recommend avoiding commas in property names. If necessary, you can URL encode the comma in the name string before composing the comma-separated string of all names.

The tiles produced by PostGIS and published via pg_tileserv are "Mapbox vector tiles", a widely used de facto standard encoding of vector tiles.

The purpose of vector tiles is to efficiently transfer map features over the network, so they optimize for size, using a variety of techniques while retaining enough context to be useful to the client mapping environment.

Resolution

Coordinates in tiles are quantized to integer values, and the default resolution of vector tiles is 4096 by 4096. The default resolution can be altered using the DefaultResolution configuration parameter.

Tile Buffer

Tiles are rendered independently. For features with wide styles near borders, a copy of the feature needs to appear in both neighboring tiles, or a rendering failure will occur.

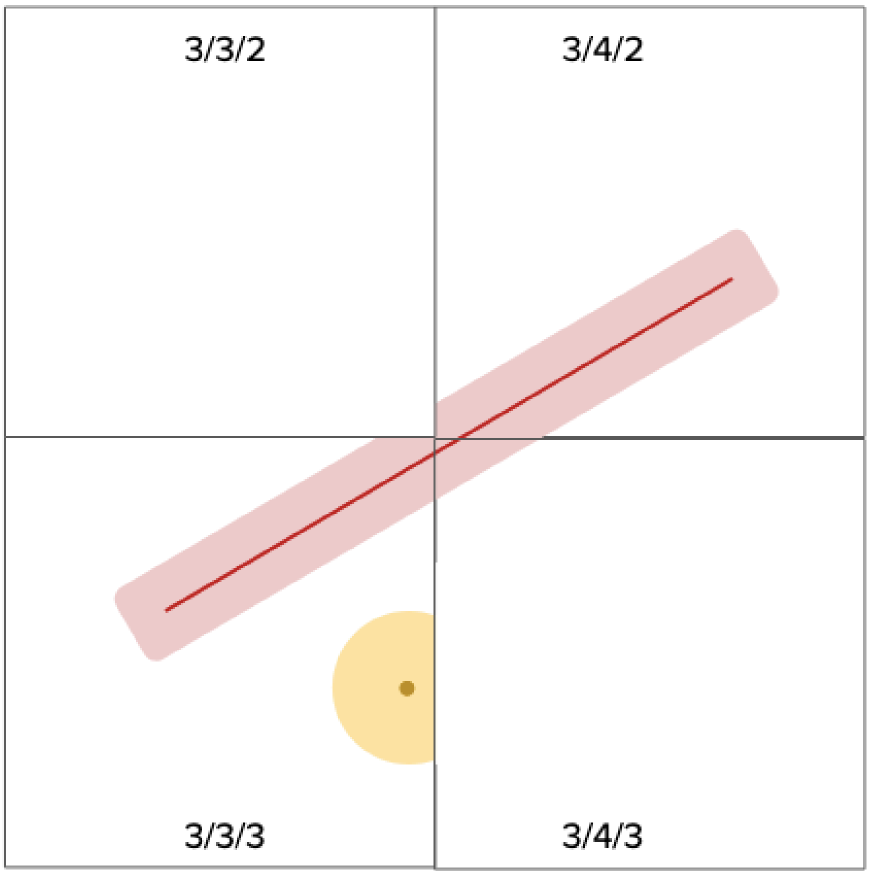


Figure 2: Tile rendering failure

The default tile buffer is 256 pixels, which is enough for most rendering cases. You can make your tiles smaller if you have narrow rendering styles, by reducing the DefaultBuffer configuration parameter.

Unique Identifier

The vector tile specification includes an optional "id" element that provides a unique feature identifier.

A single feature can end up in multiple tiles, and the unique identifier allows the client side renderer to do things like roll-overs and highlights on features that span tile boundaries.

The tile server can automatically populate the "id" element, but only in cases where:

- PostGIS version is >= 3.0, as the ST_AsMVT() function did not support feature id until then.
- The table being published has a integer primary key defined. This key will be used as the "id" automatically.

For function layers, the "id" can be populated, but that task is left to the function author, who will be calling the ST_AsMVT() function in their code, and must remember to populate the feature id name field. The column that is chosen to populate the "id" element **must** be unique per feature.

The web map examples in this section are set up to render a basemap layer from Wikimedia and vector tiles from pg_tileserver running on a local machine, using popular open source JavaScript web map components.

Load Natural Earth Data

Database preparation

The following terminal commands will create a database named naturalearth, assuming that your user account has create database privilege:

```
createdb naturalearth
```

Load the PostGIS extension as superuser (postgres):

```
psql -U postgres -d naturalearth -c 'CREATE EXTENSION postgis'
```

Import shapefile

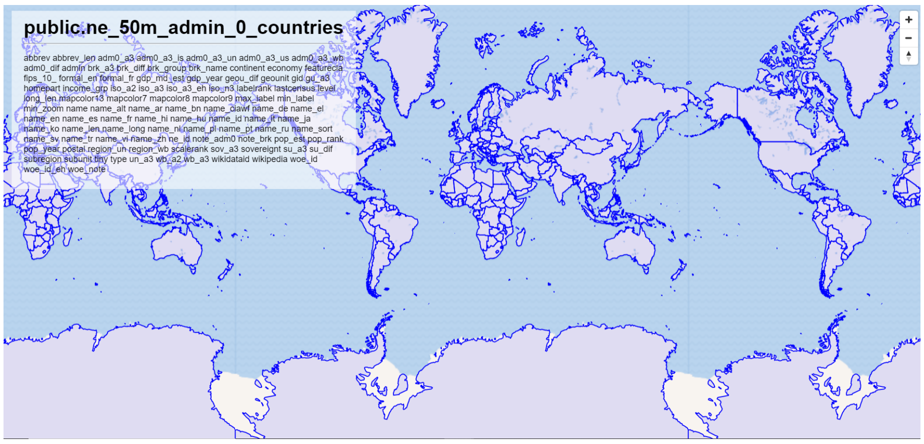
The data used in the examples are loaded from Natural Earth. Download the *Admin 0 - Countries* ZIP and extract to a location on your machine.

In that directory, run the following command in the terminal to load the shapefile data into the naturalearth database. This creates a new table ne_50m_admin_0_countries, with the application user as the owner – refer to Table Layers and Security for more information on access to spatial tables on pg_tileserv.

```
shp2pgsql -D -s 4326 ne_50m_admin_0_countries.shp | psql -U username -d natur
```

You should see the ne_50m_admin_0_countries table with the \dt SQL shell command.

Make sure that pg_tileserv connection specifies naturalearth, i.e.: DATABASE_URL=postgres://username
With the service running, you should also see the layer on the web preview, i.e.: http://localhost:7800/public.ne_50m_admin_0_countries.html



public.ne_50m_admin_0_countries

abbrev abbrev_len adm0_a3 adm0_a3_is adm0_a3_un adm0_a3_us adm0_a3_wb adm0_dif admin brk_a3 brk_diff brk_group brk_name continent economy featurecla fips_10_ formal_en formal_fr gdp_md_est gdp_year geou_dif geounit gid gu_a3 homepart income_grp iso_a2 iso_a3 iso_a3_eh iso_n3 labelrank lastcensus level long_len mapcolor13 mapcolor7 mapcolor8 mapcolor9 max_label min_label min_zoom name name_alt name_ar name_bn name_ciawf name_de name_el name_en name_es name_fr name_hi name_hu name_id name_it name_ja name_ko name_len name_long name_nl name_pl name_pt name_ru name_sort name_sv name_tr name_vi name_zh ne_id note_adm0 note_brk pop_est pop_rank pop_year postal region_un region_wb scalerank sov_a3 sovereignt su_a3 su_dif subregion subunit tiny type un_a3 wb_a2 wb_a3 wikidataid wikipedia woe_id woe_id_eh woe_note

Figure 3: pg_tileserv web interface preview

With the service running, open this HTML in your browser.

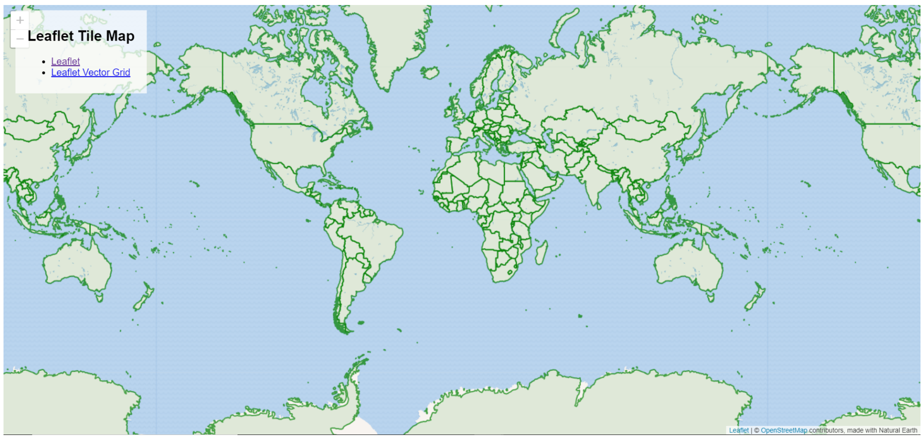

Leaflet Tile Map
Leaflet
Leaflet Vector Grid
Leaflet | © OpenStreetMap contributors, made with Natural Earth

Figure 4: Leaflet map preview

This example demonstrates interactivity where clicking within a boundary displays a popup that shows feature properties. The web preview for pg_tileserv also uses Mapbox GL JS.

With the service running, open the HTML in your browser.

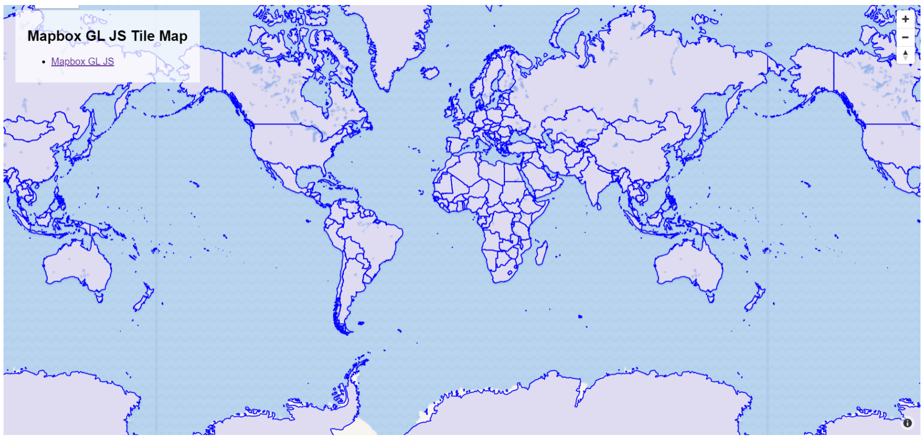


Figure 5: Mapbox GL JS map preview

With the service running, open this HTML in your browser.

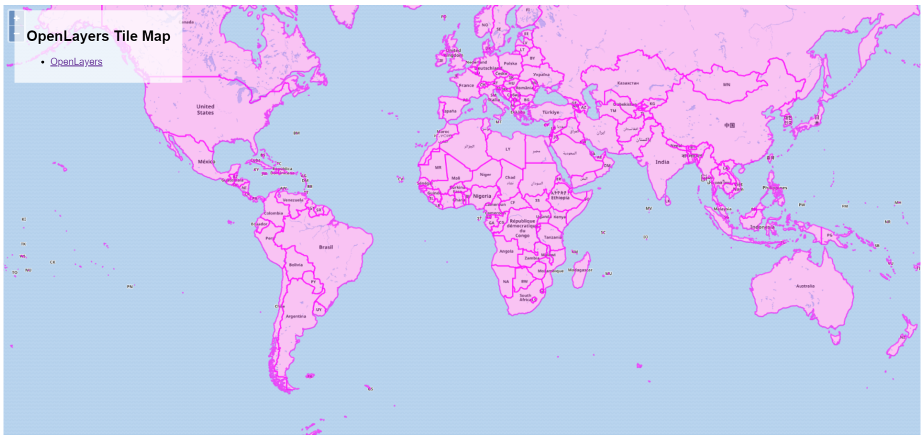


Figure 6: OpenLayers map preview

Requirements

- **PostgreSQL 9.5** or later
- **PostGIS 2.4** or later

The tile server depends on the ST_AsMVT() function, which is only available if PostGIS has been compiled with support for the **libprotobuf** library. See the output from PostGIS_Full_Version, for example:

```
SELECT postgis_full_version()
```

```
POSTGIS="3.0.1" [EXTENSION] PGSQL="121" GEOS="3.8.0-CAPI-1.13.1 "
PROJ="6.1.0" LIBXML="2.9.4" LIBJSON="0.13"
LIBPROTOBUF="1.3.2" WAGYU="0.4.3 (Internal)"
```

You don't need advanced knowledge in Postgres/PostGIS or web mapping to install pg_tileserv and set up the examples in this guide. If you are new to functions in Postgres, you might try this quick interactive course to better see how you might take advantage of pg_tileserv's capabilities.

We also link to further resources at the end of this guide, for your reference.

Installation

To install pg_tileserv, download the binary file. Alternatively, you may run a container. These first two options will suit most use cases; needing to build the executable from source is rare.

A. Download binaries

Builds of the latest code:

- Linux
- Windows
- OSX

Unzip the file, copy the pg_tileserv binary wherever you wish, or use it in place. If you move the binary, remember to move the assets/ directory to the same location, or start the server using the AssetsDir configuration option.

B. Run container

A Docker image is available on DockerHub:

- Docker

When you run the container, provide the database connection information in the DATABASE_URL environment variable and map the default service port (7800).

```
docker run -e DATABASE_URL=postgres://user:pass@host/dbname -p 7800:7800 pram
```

C. Build from source

If you must build from source, install the Go software development environment. Make sure that the GOPATH environment variable is also set.

```
SRC=$GOPATH/src/github.com/CrunchyData
mkdir -p $SRC
cd $SRC
git clone git@github.com:CrunchyData/pg_tileserv.git
cd pg_tileserv
go build
go install
```

To run the build, set the DATABASE_URL environment variable to the database you want to connect to, and run the binary.

```
export DATABASE_URL=postgres://user:pass@host/dbname
$GOPATH/bin/pg_tileserv
```

Deployment

Basic operation

```
export DATABASE_URL=postgresql://username:password@host/dbname
./pg_tileserv
```

```
SET DATABASE_URL=postgresql://username:password@host/dbname
pg_tileserv.exe
```

Configuration file

The configuration file will be automatically read from the following locations, if it exists:

- In the system configuration directory, at /etc/pg_tileserv.toml
- Relative to the directory from which the program is run, ./pg_tileserv.toml

If you want to pass a path directly to the configuration file, use the −−config command line parameter.

Configuration files in other locations will be ignored when using the −−config option.

```
./pg_tileserv −−config /opt/pg_tileserv/pg_tileserv.toml
```

The default settings will suit most uses, and the program autodetects values such as the server name.

```
# Database connection
DbConnection = "user=you host=localhost dbname=yourdb"
# Close pooled connections after this interval
DbPoolMaxConnLifeTime = "1h"
# Hold no more than this number of connections in the database pool
DbPoolMaxConns = 4
# Look to read html templates from this directory
AssetsPath = "./assets"
# Accept connections on this subnet (default accepts on all subnets)
HttpHost = "0.0.0.0"
# Accept connections on this port
HttpPort = 7800
# Advertise URLs relative to this server name
# default is to look this up from incoming request headers
# UrlBase = "http://yourserver.com/"
# Resolution to quantize vector tiles to
DefaultResolution = 4096
# Rendering buffer to add to vector tiles
DefaultBuffer = 256
# Limit number of features requested (−1 = no limit)
MaxFeaturesPerTile = 10000
# Advertise this minimum zoom level
DefaultMinZoom = 0
# Advertise this maximum zoom level
DefaultMaxZoom = 22
# Allow any page to consume these tiles
CORSOrigins = *
tra logging information?
Debug = false
```

Motivation

There are numerous tile generators available (such as Tegola, Geoserver, Mapserver) that read from multiple data sources and generate vector tiles.

pg_tileserv works exclusively with PostGIS data, but this also allows more flexibility of usage.

By restricting itself to only using PostGIS as a data source, pg_tileserv gains the following features:

- **Automatic configuration.** The server can discover and automatically publish as tiles sources all tables it has read access to: just point it at a PostgreSQL/PostGIS database.
- **Full SQL flexibility.** Using function layers, the server can run any SQL to generate tile outputs. Any data processing, feature filtering, or record aggregation that can be expressed in SQL, can be exposed as parameterized tile sources.
- **Database security model.** You can restrict access to tables and functions using standard database access control. This means you can also use advanced access control techniques, like row-level security to dynamically filter access based on the login role.

Architecture

pg_tileserv is one component in “PostGIS for the Web” (aka “PostGIS FTW”), a growing family of Go spatial microservices. Database-centric applications naturally have a central source of coordinating state, the database, which allows otherwise independent microservices to coordinate and provide HTTP-level access to the database with less middleware software complexity.

- pg_tileserv provides MVT tiles for interactive clients and smooth rendering
- pg_featureserv provides GeoJSON feature services for reading and writing vector and attribute data from tables

PostGIS for the Web makes it possible to stand up a spatial services architecture of stateless microservices surrounding a PostgreSQL/PostGIS database cluster, in a standard container environment, on any cloud platform or internal datacenter.

Definitions

- **Map tiles** are a way of representing a multi-scale, zoomable cartographic map by regularly subidividing the plane into independent tiles that can then be rendered on a server and retrieved by a map client in parallel.
- **Vector tiles** are a specific format of map tile that encode the features as vectors and delegate to the client web browser the rendering of the features into cartography. Client side vector rendering uses less bandwidth, which is good for mobile clients, and allow more options for client side dynamic data visualizations.
- A **spatial database** is a database that includes a “geometry” column type. The PostGIS extension to PostgreSQL adds a geometry column type, as well as hundreds of functions to operate on that type, including the ST_AsMVT() function that pg_tileserv depends upon.

GIS

- QGIS is a free and open source application for editing, visualizing, and analyzing spatial data. Get started with the QGIS Training Manual.
- The Introduction to PostGIS Workshop is a full tutorial on the PostGIS extension.
- Shorter interactive courses on PostGIS are also available on the Crunchy Data Learning Platform.
- Learn more about practical applications of PostGIS with PostGIS Day 2019 Talks.

Web Mapping

- OpenLayers
- Leaflet
- Mapbox GL JS

Source Code

- GitHub

Tile Server

To get more information about what's going on behind the scenes, run the server with the --debug command line parameter:

```
./pg_tileserv --debug
```

Or, turn on debugging in the configuration file.

Web Layer

Hitting your service end points with a command-line utility like curl can also yield useful information:

```
curl -I http://localhost:7800/index.json
```

Database Layer

The debug mode of the tile server returns the SQL that is being called on the database. If you want to delve more deeply into all the SQL that is being run on the database, you can turn on statement logging in PostgreSQL by editing the postgresql.conf file for your database and restarting.

Bug Reporting

If you find an issue with the tile server, bugs can be reported on GitHub at the issue tracker:

- https://github.com/crunchydata/pg_tileserv/issues