pg_featureserv

Contents

pg_featureserv **1**

Figure 1: Crunchy Spatial

pg_featureserv

This is the documentation for pg_featureserv version 1.1.

pg_featureserv is a PostGIS-based feature server written in Go. It is a lightweight, low-configuration RESTful web service that provides access to spatial data stored in PostGIS tables, as well as spatial processing capability based on PostGIS spatial functions.

pg_featureserv supports a wide variety of situations where web access to spatial data enables richer functionality. Use cases include:

- Display features at a point or in an area of interest
- Query features using spatial and/or attribute filters
- Retrieve features for use in a web application (e.g. for tabular or map display)
- Download spatial data for use in applications

This guide walks you through how to install and use pg_featureserv for your spatial applications. See Quick Start to learn how to get the service up and running with a spatial database. The Usage section goes in-depth on how the service works. We're continuing to add basic examples of working with feature data from pg_featureserv.

Installation

To install pg_featureserv, 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_featureserv 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 (9000).

```
docker run -e DATABASE_URL=postgres://username:password@host/dbname -p 9000:9
```

C. Build from source

If not already installed, install the Go software development environment. Make sure that the GOPATH environment variable is also set.

The application can downloaded and built with the following commands:

```
mkdir -p $GOPATH/src/github.com/CrunchyData
cd $GOPATH/src/github.com/CrunchyData
git clone git@github.com:CrunchyData/pg_featureserv.git
cd pg_featureserv
go build
```

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

```
export DATABASE_URL=postgres://username:password@host/dbname
$GOPATH/bin/pg_featureserv
```

pg_featureserv's architecture is simple. It consists of a single server application, written in Go. It is configured via static (read-only) information sourced from a file, the command line and/or environment variables.

pg_featureserv can run stand-alone or inside a containerized environment. It connects to a Postgres database using an internal database pool (which can itself connect to a database load-balancer such as pgbouncer). It comes with an integrated web server which provides the HTTP interface to clients. The interface provides both a data-centric REST API and a HTML-based user interface.

In other words, the service integrates with the following:

- A PostGIS-enabled Postgres database instance or cluster, containing the data being served and the catalog metadata describing the data.
- Client software which accesses the HTTP interface. Typically this is a web-mapping application running in a web browser, but it could also be a non-browser application (ranging from a simple data access utility such as curl or OGR, to a desktop GIS application such as QGIS), or a web proxy mediating access to the service.

The context diagram below shows pg_featureserv running alongside pg_tileserv to provide a PostGIS-centric "platform for the spatial web".

Motivation

There are numerous services available that can be used to serve features, such as Geoserver, Mapserver, and pygeoapi. These applications typically provide the capability to read from multiple data sources and generate feature datasets in various formats. They also tend to be large, complex applications which require significant expertise to install, configure, secure and tune.

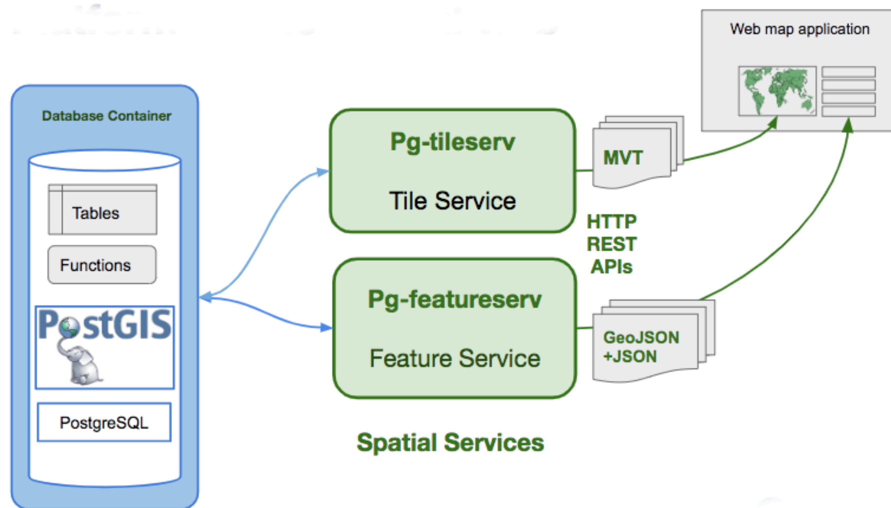

Figure 2: pg_feaureserv Architecture

PostGIS-Only

In contrast, pg_featureserv works exclusively with PostGIS, which allows for greater flexibility of usage. By targetting PostGIS as the sole data provider, pg_featureserv gains significant capabilties:

- **Automatic configuration.** Just point the server at a PostgreSQL / PostGIS database, and the server discovers and automatically publishes all tables it has access to.

  The Postgres system catalog provides all the metadata needed to support publishing datasets (such as primary key columns and table descriptions). Changes to the database are then published automatically without needing to restart the service. You can also take advantage of Postgres' clustering capabilites to provide scale-out and high availability.

- **Full SQL power.** The server relies on the database to conduct all data operations, including converting geometry records into GeoJSON. Since the database is optimized to perform operations such as filtering and sorting, this increases your application's performance.

  By using functions as data sources, the server can run any SQL at all to generate features. Any data processing, feature filtering, or record aggregation that you can express in SQL can be published as feature datasets. Function parameters are also exposed as URL query parameters, which allows dynamically changing the data returned.

Using the full power of SQL means that it is easy to publish any existing database functionality via the service, and the learning curve for developers can be minimized.

- **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.

By using a single powerful spatial data source, the pg_featureserv codebase is significantly smaller and simpler. This means more rapid development, fewer software defects, a more secure interface, and easier deployment on a wider variety of platforms.

Modern web service architecture

pg_featureserv follows the modern architectural paradigm of web-friendly, RESTful microservices.

As noted in the W3C/OGC Spatial Data on the Web Best Practices, exposing spatial data using modern web standards improves spatial data discoverability, accessibility and interoperability.

A key benefit of following the lightweight OGC API for Features Core standard is the ease of extending it to expose service-specific capabilities, including the powerful spatial capabilities of PostGIS. For instance, with pg_featureserv you can query spatial functions as well as static collections, using a similar API.

By focussing on the single aspect of serving spatial features, pg_featureserv makes it easier to deploy, provision, manage, and secure feature services within a containerized environment.

PostGIS for the Web

pg_featureserv is one component of *PostGIS for the Web* (aka "PostGIS FTW"), a growing family of spatial micro-services. Database-centric applications naturally have a central source of coordinating state, the database, which allows otherwise independent microservices to provide HTTP-level access to the data with minimal middleware 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.

Feature

A representation of a real-world spatial phenomenon which can be modelled by a geometry and zero or more scalar-valued properties.

Feature collection

A set of **features** from a spatial dataset. In pg_featureserv, these are mapped to database tables and views.

Spatial database

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. For example, it provides the ST_AsGeoJSON() function that pg_featureserv uses.

Web API

An Application Program Interface (API) allows client software to make programmatic requests to a service and retrieve information from it.

A Web API is an API founded on Web technologies. These include:

- Use of the HTTP protocol to provide high-level semantics for operations, as well as efficient mechanisms for querying, security and transporting data to clients
- Following the REST paradigm to simplify the model of interacting with data
- Using the standard JSON and GeoJSON formats as the primary way of encoding data

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.

Source Code

- GitHub

This section describes how to set up pg_featureserv and connect the service to a spatial database.

The first half walks through how to prepare a spatial database and import spatial data, using the terminal. If you already have a spatial database, you can go ahead and start with "Configuring the service."

Database preparation

The following terminal command creates a new database named naturalearth (assuming your user role has the create database privilege):

```
createdb naturalearth
```

Using the psql tool, load the PostGIS extension as superuser (we'll go with postgres):

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

We're going to be tidy and load the data into a schema ne. To create the schema, run the command:

```
psql -U postgres -d naturalearth -c 'CREATE SCHEMA ne'
```

When we get to the step below to connect pg_featureserv to the database, the user must have access to the new schema as well.

Import data

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

In that directory, run the following terminal command to load the data into the ne schema in the naturalearth database. This creates a new table countries, with the application user as the owner.

```
shp2pgsql -D -s 4326 ne_50m_admin_0_countries.shp ne.countries | psql -U <user
```

You should see the ne.countries table using the \dt ne.* command in the psql SQL shell.

For more information about publishing spatial tables in pg_featureserv, refer to the Feature Collections and Security sections.

Configuring the service

Make sure that the service database connection specifies the naturalearth database. As described in the Configuration section, this can be provided either by an environment variable:

Linux/OSX

```
export DATABASE_URL=postgresql://username:password@host/naturalearth
```

Windows

```
SET DATABASE_URL=postgresql://username:password@host/naturalearth
```

Or by a configuration file parameter:

```
DbConnection = "postgresql://username:password@localhost/naturalearth"
```

Download the build of the latest code:

- Linux
- Windows
- OSX

Unzip the file, copy the pg_featureserv 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.)

Deploy pg_featureserv

In the directory where the pg_featureserv binary is located, run the service in the terminal:

Linux/OSX

```
./pg_featureserv
```

Windows

```
pg_featureserv.exe
```

With the service running, you should see the layer listed on the web user interface at http://localhost:9000/collections.html. The layer metadata is viewable at http://localhost:9000/collections/ne.countries.html.

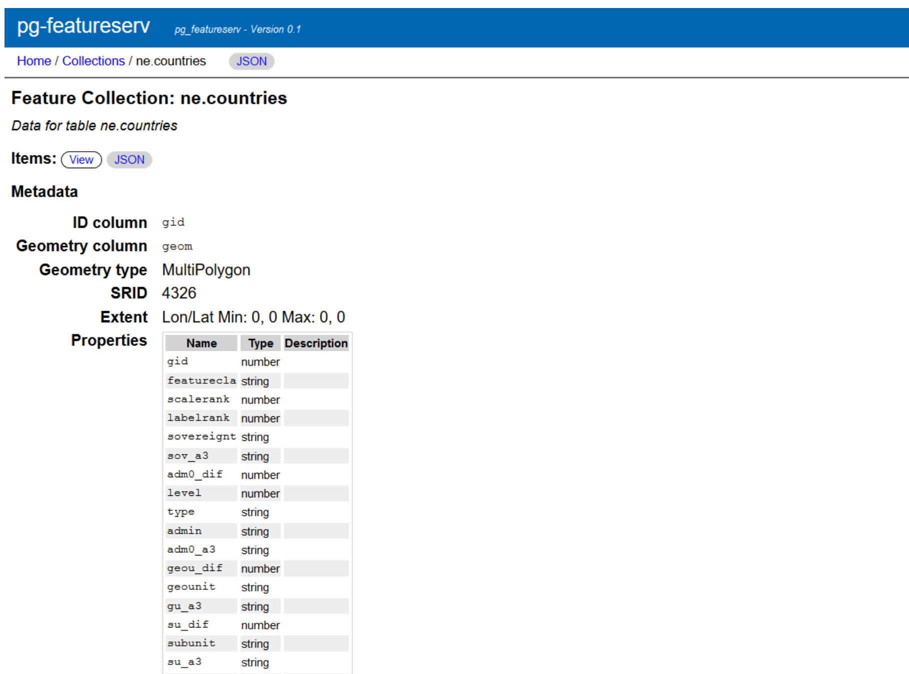

Figure 3: pg_featureserv web interface preview

Approaches

If the service isn't behaving as expected, there are a few approaches you can use to determine the issue.

HTTP Response

The service indicates the status of reponses using standard HTTP status codes, along with text messages. See the API section for details of status codes and their meanings.

HTTP status codes and headers returned in service responses can be displayed by querying them with a command-line utility like curl:

```
curl -I http://localhost:9000/home.json
```

Alternatively, most web browsers provide a debugger which can display detailed response information.

Service Logging

The service outputs logging information to the console. By default, the log level is set to show errors and warnings only. Running the service with debug level logging will provide more information about request processing. This can include the actual SQL emitted to the database, SQL errors, and timing of queries and responses.

To invoke debug mode, run the server with the --debug command-line parameter:

```
./pg_featureserv --debug
```

You can also turn on debug logging in the configuration file:

```
Debug = true
```

SQL Logging

The debug mode of the server logs the SQL that is being emitted to the database. If you have access to the database that the service is querying, it can be useful to try manually executing the SQL. This can provide more detailed database error reporting.

For issues involving access permissions, it may be useful to connect as the same user that the service is using.

To delve more deeply into 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 feature server, it can be reported on the GitHub issue tracker:

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

When reporting an issue, please provide the software version being used. This can be obtained from the service log, or by running:

```
./pg_featureserv --version
```

This section describes how to use pg_featureserv. It covers the following topics:

- How the Web Service API works

- How to publish feature collections backed by PostGIS tables or views
- How to query features from feature collections
- How to publish database functions
- How to execute functions

pg_featureserv provides a HTTP-based RESTful web service API to access metadata about as well as data from the PostGIS objects it publishes. This section discusses general aspects of the API.

OGC API - Features

The service implements a broad subset of the OGC API - Features standard. It implements the following paths defined by the standard:

- / - landing page
- /conformance - links to conformance resources
- /api - API specification OpenAPI document
- /collections - list of feature collections
- /collections/{id} - metadata for a feature collection
- /collections/{id}/items - data set of features from a feature collection
- /collections/{id}/items/{fid} - data for a specific feature

The standard defines various query parameters for certain paths. Many of these are provided by the service, although some are not yet implemented.

The service extends the standard API to provide richer access to the capabilities of PostGIS. Extensions include the /functions paths, and additional query parameters. See the other Usage sections for more details.

Linked data

The **OCG API - Features** standard promotes the concept of Linked Data. This makes web data more usable by providing stable links between related resources. To enable this the standard, we make sure that response documents include structured links to other resources. Like most service resources, pg_featureserv API response includes a links property containing an array of links to related resources.

A structured link includes the following properties:

- rel - the name describing the relationship of the current resource to the linked resource
- href - the URI for the link
- type - the format of the linked resource
- title - a title for the linked resource

OpenAPI

The service API is described by an OpenAPI specification. This is available as a JSON document at the path /api.

The service provides an interactive user interface for the API at /api.html. On this page, you can view the service paths and parameters, and the schemas for the responses. It allows you to try out the API as well.

CORS

The server supports Cross-origin Resource Sharing (CORS) to allow service resources to be requested by web pages which originate from another domain. The Access−Control−Allow−Origin header required by CORS-compatible responses can be set via the CORSOrigins configuration parameter.

Request headers

The service behaviour can be influenced by some request headers. These include:

- Forwarded allows a proxy to specify host and protocol for the service Base URL.
- X−Forwarded−Host allows a proxy to specify host for Base URL.
- X−Forwarded−Proto allows a sproxy to specify protocol for Base URL.
- Accept allows a client to indicate what response format(s) it can accept. Supported values are:
- text/html: indicates HTML
- application/json: indicates JSON
- application/geo+json: indicates GeoJSON

Request methods

Currently the service provides only Read-Only access to resources. The only HTTP method supported is GET.

Response formats

The service returns responses in several different formats, depending on the nature of the request. Formats include:

- JSON-formatted text, for non-spatial data
- GeoJSON for feature collections and features

- HTML documents for user interface pages

For some requests, there may be more than one format that could be returned. In particular, many paths provide both a data document (JSON or GeoJSON) and an HTML view of the data. The actual format returned is determined in one of the following ways (in descending order of precedence):

- The path extension. Values allowed are:
- .json, which indicates JSON or GeoJSON (the resource itself determines which)
- .html, which indicates an HTML page should be returned, if available
- The Accept request header value (see above for supported values).
- If the path extension or Accept request header is not specified, the default is to return a data document (JSON or GeoJSON).

When using a web browser to query the service, the browser generally provides an Accept header of text/html. So you may need to explicitly specify the .json extension to retrieve a data document instead of an HTML page.

Status codes and messages

The HTTP protocol defines a standard set of status codes returned by responses. pg_featureserv can return the following codes:

Code	Meaning
200 OK	The request has succeeded.
400 Bad Request	The server could not understand the request due to invalid syntax.
404 Not Found	The server can not find the requested resource.
500 Internal Server Error	The server has encountered a situation it is unable to handle.
503 Service Unavailable	The server is unable to handle the request. Can indicate a timeout caused by

The basic principle of security in pg_featureserv is to connect the server to the database with a user that has just the access you want it to have, and no more. (Note: Postgres uses the term database role when discussing user access permisions.)

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.

```sql
CREATE USER featureserver;
```

To support different access patterns, create different users with access to different tables/functions. Then, run multiple service instances, connecting with those different users.

Table and view access

If your tables are in a schema other than public, you must also grant usage on that schema to your user.

```sql
GRANT USAGE ON SCHEMA myschema TO featureserver;
```

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

```sql
GRANT SELECT ON TABLE myschema.mytable TO featureserver;
```

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

```sql
GRANT SELECT ON ALL TABLES IN SCHEMA myschema TO featureserver;
```

Function access

As noted above, functions that access table data effectively are 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.

```sql
-- 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 postgisftw.myfunction FROM public;
-- Just to be sure, also revoke execute from the user
REVOKE EXECUTE ON FUNCTION postgisftw.myfunction FROM featureserver;
```

Home page

The home page shows the service title and description, and provides links to the listings of collections and functions, the OpenAPI definition, and the conformance metadata.

```
http://localhost:9000/index.html
```

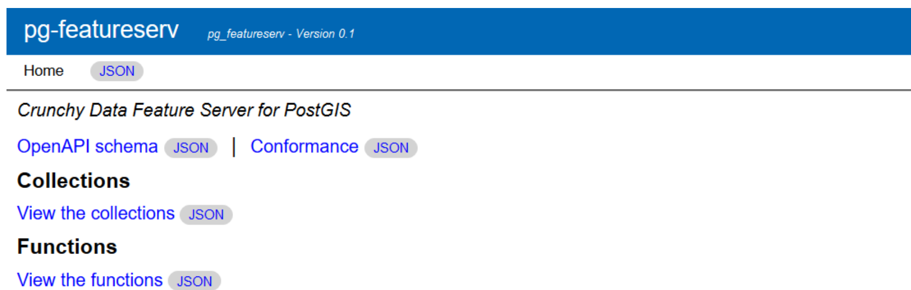

Figure 4: pg_featureserv UI home page

API user interface

A user interface for the service API is available at the path /api.html.

List feature collections

The path /collections.html shows a list of the feature collections published by the service.

Show feature collection metadata

The path /collections/{collid}.html shows metadata about the specified feature collection.

View features on a map

The path /collections/{collid}/items.html shows the features returned by a basic query in a web map interface. The map interface provides a simple UI that allows setting some basic query parameters.

Any applicable query parameters may be appended to the URL.

View a feature on a map

The path /collections/{collid}/items/{fid} shows the feature requested by the query in a web map interface.

Any applicable query parameters may be appended to the URL.

List functions

The path /functions.html shows a list of the functions published by the service.

Show function metadata

The path /functions/{funid}.html shows metadata about the specified function.

View function result data on a map

The path /functions/{funid}/items.html shows the features returned by a basic function query in a web map interface. The map interface provides a simple UI that allows specifying function arguments and setting some basic query parameters.

Note that only functions with spatial results can be viewed on a map.

Any applicable query parameters may be appended to the URL.

Following the OGC Features information model, the service API publishes PostGIS tables and views as **feature collections**.

The available feature collections are listed. Each feature collection can report metadata about its definition, and can be queried to return data sets of features. It is also possible to query individual features in tables which have defined primary keys.

Publish tables and views as feature collections

pg_featureserv publishes all spatial tables and views which are visible in the database.

Spatial tables and views are those which:

- include a geometry column;
- declare a geometry type; and,
- declare an SRID (spatial reference ID).

Example of a spatial table Here is a simple example of defining a spatial table which contains polygon geometries using coordinate system SRID = 4326. (See the PostGIS documentation for more information about creating spatial tables and using spatial reference systems.)

```sql
CREATE TABLE mytable (
    geom Geometry(Polygon, 4326),
    pid text,
    address text
);
```

Tables and views are visible when they are available for access based on the database access permissions defined for the service database user (role). See the Security section for examples of setting role privileges.

If a view directly uses the geometry column of an underlying table, the spatial column metadata is inherited for the view. But if a view column is defined as the result of a spatial function, then the column must be explicitly cast to a geometry type providing the type and SRID. Depending on the spatial function used, it may also be necessary to explicitly set the SRID of the created geometry.

```sql
CREATE VIEW my_points AS
  SELECT ST_SetSRID( ST_MakePoint( lon, lat ), 4326)::geometry(Point, 4326)
  FROM my_geo_table AS t;
```

The service uses the database catalog information to provide metadata about a feature collection backed by a table or view:

- The feature collection ID is the schema-qualified name of the table or view.
- The feature collection description is provided by the comment on the table or view.
- The feature geometry is provided by the spatial column of the table or view.
- The identifier for features is provided by the primary key column for a table (if any).
- The property names and types are provided by the non-spatial columns of the table or view.
- The description for properties is provided by the comments on table/view columns.

```sql
COMMENT ON TABLE mytable IS 'This is my spatial table';
COMMENT ON COLUMN mytable.geom IS 'The geometry column contains polygons in S
COMMENT ON COLUMN mytable.pid IS 'The Parcel Identifier is the primary key';
COMMENT ON COLUMN mytable.address IS 'The address of the Parcel';
```

List feature collections

The path /collections returns a JSON document containing a list of the feature collections published by the service.

```
http://localhost:9000/collections
```

Each listed feature collection is described by a subset of its metadata, including name, title, description and extent. A list of links provide URLs for accessing:

- self - the feature collection metadata
- alternate - the feature collection metadata as an HTML view
- items - the feature collection data items

Describe feature collection metadata

The path /collections/{coll-name} returns a JSON object describing the metadata for a feature collection. {coll-name} is the schema-qualified name of the database table or view backing the feature collection.

```
http://localhost:9000/collections/ne.admin_0_countries
```

The response is a JSON document containing metadata about the collection, including:

- The geometry column name
- The geometry type
- The geometry spatial reference code (SRID)
- The extent of the feature collection (if available)
- The column name providing the feature identifiers (if any)
- A list of the properties and their JSON types

A list of links provide URLs for accessing:

- self - the feature collection metadata
- alternate - the feature collection metadata as an HTML view
- items - the data items returned by querying the feature collection

A powerful feature of Postgres is the ability to create user-defined functions. Functions allow encapsulating complex logic behind a simple interface (namely, providing some input arguments and getting output as a set of records). This makes them easy to publish via a simple web API.

Functions can execute any data processing that is possible to perform with Postgres and PostGIS. They can return either spatial or non-spatial results (as GeoJSON or plain JSON). They thus provide a further extension to the capabilities of the pg_featureserv API.

Potential uses for functions include:

- Query a spatial database table or view with custom SQL, which can include more complex filters than the API provides, joins to other tables, or aggregation.
- Query a non-spatial table or view to return data objects or a summary record. For example, this could be used to provide values for a client-side drop-down list or an autocomplete feature.
- Generate spatial data controlled by a set of parameters.
- Provide a geometric computation, by accepting a geometric input value and returning a single record containing the result.
- Update data (as long as appropriate security is in place).

Publish database functions

The service can publish any function which returns a set of rows using the return type SETOF record or the equivalent (and more standard) TABLE (see the Postgres manual section on set-returning functions.)

Because there are usually many functions in a Postgres database, the service only publishes functions defined in the postgisftw schema.

A function specifies zero or more input parameters. An input parameter can be of any Postgres type which has a cast from a text representation. This includes the PostGIS geometry and geography types, which support text representations of WKT or WKB. Input parameter names are exposed as query parameters, so you should avoid using names which are existing API qeuery parameters.

A function must return a set of records containing one or more columns, of any Postgres type. A **spatial function** is one which returns a column of type

geometry or geography. Output from spatial functions is returned as GeoJSON datasets, while output from non-spatial functions is returned as JSON datasets.

Geometry values returned by a function can be in any coordinate system, but must have their SRID set to the appropriate value. If required, they are re-projected to geographic coordinates (SRID = 4326) in the output GeoJSON. If geometry is queried from an existing table, the SRID may already be set; otherwise the function should set it explicitly.

The example below illustrates the basic structure of a spatial set-returning function. See the Examples section for further examples.

Example of a spatial function This function returns a filtered subset of a table (created using the Natural Earth ne_50m_admin_0_countries dataset which is in EPSG:4326). The filter in this case is the first letters of the country name.

The name_prefix parameter includes a **default value**: this is useful for clients that read arbitrary function definitions and need a default value to fill into interface fields. The preview interface for pg_featureserv is an example.

```
CREATE OR REPLACE FUNCTION postgisftw.countries_name(
    name_prefix text DEFAULT 'A')
RETURNS TABLE(name text, abbrev text, continent text, geom geometry)
AS $$
BEGIN
    RETURN QUERY
        SELECT t.name::text,
            t.abbrev::text,
            t.continent::text,
            t.geom
    FROM ne.admin_0_countries t
    WHERE t.name ILIKE name_prefix || '%';
END;
$$
LANGUAGE 'plpgsql' STABLE PARALLEL SAFE;

COMMENT ON FUNCTION postgisftw.countries_name IS 'Filters the countries table
```

Notes:

- The function is defined in the postgisftw schema.
- Tt has a single input parameter name_prefix, with the DEFAULT value 'A'.
- It returns a table (set) of type (name text, geom geometry).

- The function body is a simple SELECT query which uses the input parameter as part of a ILIKE filter, and returns a column list matching the output table definition.
- The geometry values are assumed to carry an SRID specified in the queried table.
- The function "volatility" is declared as STABLE because within a transaction context, multiple calls 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 state that might be altered by making multiple concurrent calls to the function.

The function can be called via the API by providing a value for the name_prefix parameter (which could be omitted, due to the presence of a default value):

```
http://localhost:9000/functions/countries_name/items?name_prefix=T
```

The response is a GeoJSON document containing the 13 countries starting with the letter 'T'.

As with feature collections, available functions can be listed, and each function can supply metadata describing it.

List functions

The path /functions returns a JSON document containing a list of the functions available in the service.

```
http://localhost:9000/functions
```

Each listed function is described by a subset of its metadata, including its id and description. A list of links provide URLs for accessing:

- self - the function metadata
- alternate - the function metadata as an HTML view
- items - the function data items

Describe function metadata

The path /functions/{funid} returns a JSON object describing the metadata for a database function. {funid} is the name of the function. It is not schema-qualified, because functions are published from only one schema.

```
http://localhost:9000/functions/geonames_geom
```

The response is a JSON document containing metadata about the function, including:

- The function description
- A list of the input parameters, described by name, type, description, and default value (if any)
- A list of the properties and their JSON types

A list of links provides URLs for accessing:

- self - the function metadata
- alternate - the function metadata as an HTML view
- items - the data items returned by querying the function

Feature collections can be queried to provide sets of features, or to return a single feature.

Query features

The path /collections/{collid}/items is the basic query to return a set of features from a feature collection. The response is a GeoJSON feature collection containing the result.

```
http://localhost:9000/collections/ne.countries/items
```

Additional query parameters can be appended to the basic query to provide control over what sets of features are returned.

These are similar to using SQL statement clauses to control the results of a query. In fact, the service implements these parameters by generating the equivalent SQL. This allows the Postgres SQL engine to optimize the query execution plan.

Filter by bounding box

The query parameter bbox=MINX,MINY,MAXX,MAXY limits the features returned to those that intersect a specified bounding box. The bounding box is specified in geographic coordinates (longitude/latitude, SRID = 4326). If the source data has a non-geographic coordinate system, the bounding box is transformed to the source coordinate system to perform the query.

```
http://localhost:9000/collections/ne.countries/items?bbox=10.4,43.3,26.4,47.7
```

Filter by properties

The response feature set can be filtered to include only features which have a given value for one or more properties. This is done by including query parameters which have the same name as the property to be filtered. The value of the parameter is the desired property value.

```
http://localhost:9000/collections/ne.countries/items?continent=Europe
```

Specify respones properties

The query parameter properties=PROP1,PROP2,PROP3... specifies the feature properties returned in the response. This can reduce the response size of feature collections which have a large number of properties.

```
http://localhost:9000/collections/ne.countries/items?properties=name,abbrev,p
```

Limiting and paging

The query parameter limit=N controls the maximum number of features returned in a response document. There is also a server-defined maximum which cannot be exceeded.

The query parameter offset=N specifies the offset in the actual query result at which the response feature set starts.

When used together, these two parameters allow paging through large result sets.

```
http://localhost:9000/collections/ne.countries/items?limit=50&offset=200
```

Even if the limit parameter is not specified, the response feature count is limited to avoid overloading the server and client. The default number of features in a response is set by the configuration parameter LimitDefault. The maximum number of features which can be requested in the limit parameter is set by the configuration parameters LimitMax.

Ordering

The result set can be ordered by any property it contains. This allows performing "greatest N" or "smallest N" queries.

- orderBy=PROP orders results by PROP in ascending order

The sort order can be specified by appending :A (ascending) or :D (descending) to the ordering property name. The default is ascending order.

- orderBy=PROP:A orders results by PROP in ascending order
- orderBy=PROP:D orders results by PROP in descending order

```
http://localhost:9000/collections/ne.countries/items?orderBy=name
```

Query a single feature

The path /collections/{collid}/items/{fid} allows querying a single feature in a feature collection by specifying its ID.

The response is a GeoJSON feature containing the result.

```
http://localhost:9000/collections/ne.countries/items/23
```

Restrict properties

The query parameter properties=PROP1,PROP2,PROP3... restricts the properties which are returned in the response.

```
http://localhost:9000/collections/ne.countries/items/23?properties=name,abbrev
```

Functions can be executed to provide sets of features or data.

Execute a function

The path /functions/{funid}/items is the basic query to execute a function and return the set of features or data it produces.

The response from a *spatial* function is a GeoJSON feature collection containing the result. The response from a *non-spatial* function is a JSON dataset containing the result.

These are similar to using SQL statement clauses to control the results of a query. In fact, the service implements these parameters by generating the equivalent SQL. However, these filters are applied to the results of an executed function, so it doesn't necessarily allow optimizing the execution of the function. (For example, specifying a bounding box only filters the results generated by the function; it is not available to the function to reduce the number of records generated.)

Function arguments

Functions provide query parameters of the form param=arg−value to provide an argument value for each function parameter. Omitted parameters use the default specified in the function definition (if any). If a function parameter does not provide a default then a value must be supplied.

```
http://localhost:9000/functions/countries_name/items?name_prefix=T
```

Filter by bounding box

The query parameter bbox=MINX,MINY,MAXX,MAXY is used to limit the features returned to those that intersect a specified bounding box. The bounding box is specified in geographic coordinates (longitude/latitude, SRID = 4326). If the source data has a non-geographic coordinate system the bounding box is transformed to the source coordinate system to perform the query.

This parameter is only useful for spatial functions.

```
http://localhost:9000/functions/countries_name/items?bbox=10.4,43.3,26.4,47.7
```

Specify response properties

The query parameter properties=PROP1,PROP2,PROP3... specifies the properties returned in the response. This reduces the response size of functions that produce a large number of records.

```
http://localhost:9000/functions/countries_name/items?properties=name
```

Limiting and paging

The query parameter limit=N can controls the maximum number of data items returned in a response.

The query parameter offset=N specifies the offset in the actual query result at which the response data set starts.

When used together, these two parameters allow paging through large result sets.

```
http://localhost:9000/functions/countries_name/items?limit=50&offset=200
```

The default page size and the maximum page size are set by the configuration parameters LimitDefault and LimitMax.

Ordering

The result set can be ordered by any property it contains. This allows performing "greatest N" or "smallest N" queries.

- orderBy=PROP orders results by PROP in ascending order

The sort order can be specified by appending :A (ascending) or :D (descending) to the ordering property name. The default is ascending order.

- orderBy=PROP:A orders results by PROP in ascending order
- orderBy=PROP:D orders results by PROP in descending order

```
http://localhost:9000/functions/countries_name/items?orderBy=name
```

The examples in this section help further illustrate how pg_featureserv is used.

We encourage you to check this Github repository for a heat map demo and an address autocomplete demo, including sample source code so you can run the demos in a browser.

This example shows how to use the pg_featureserv API to query the ne.countries feature collection created in the Quick Start section.

For more information about querying feature collections, see the Usage section.

Basic query

The most basic query against a feature collection is to retrieve an unfiltered list of the features in a collection. The number of features returned is limited by the service configuration for the default feature limit.

The following query returns a partial list of the countries in the ne.countries collection, as a GeoJSON FeatureCollection:

```
http://localhost:9000/collections/ne.countries/items.json
```

The query can also be returned as a map view in the web UI:

```
http://localhost:9000/collections/ne.countries/items.html
```

which should display a page like this:

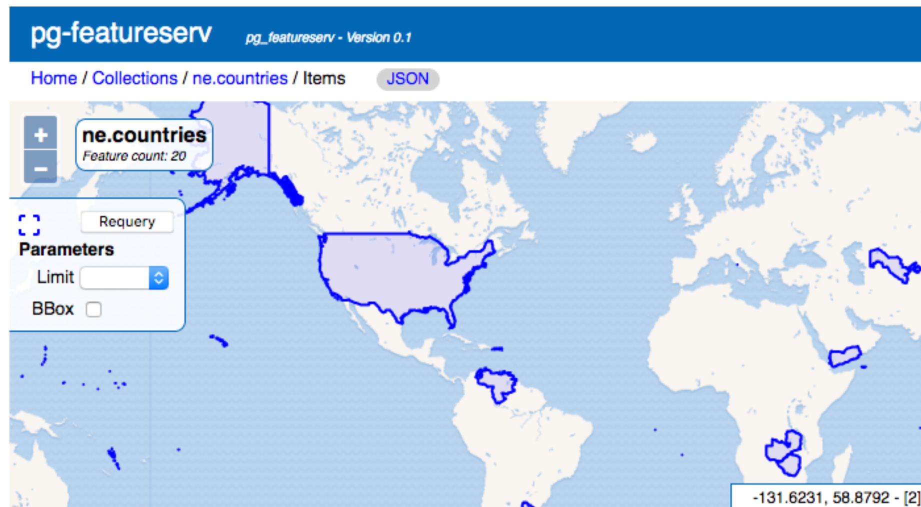

Figure 5: Map view of basic query

Query using a bounding box filter and limit

You can control the extent as well as number of features returned with the bbox and limit query parameters.

For example, to query the countries in the Caribbean and ensure that all of them are returned, you can use the query parameters like so:

```
http://localhost:9000/collections/ne.countries/items.html
?bbox=-93.0688,9.3746,-54.0296,25.9053&limit=100
```

Figure 6: Map view of query with bbox and limit

Query with a property filter and properties list

Another way to limit the features returned is via a property filter query parameter. For instance, the countries in Europe can be returned using the query parameter continent=Europe.

To make it easy to verify the result, the properties query parameter has been restricted to only three properties (including continent itself). And as before, a higher limit value ensures that all features are returned.

```
http://localhost:9000/collections/ne.countries/items.html
?continent=Europe&properties=gid,name,continent&limit=100
```

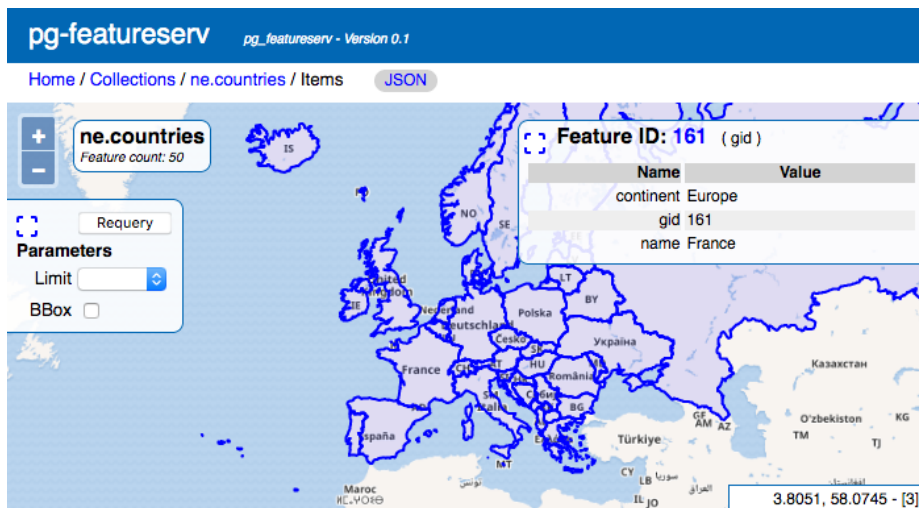

Figure 7: Map view of query with property filter

Query a Feature by ID

You can also query a single feature by providing the feature ID as part of the resource path.

Most query parameters do not apply to single feature queries. With that said, the properties parameter can be used to specify what response properties are included.

```
http://localhost:9001/collections/ne.countries/items/55.html
?properties=gid,name,continent
```

This is the same spatial function example shown in the Usage section, but we'll include a sample GeoJSON response, as well as the web UI preview.

Create a spatial function that returns a filtered set of countries

```
CREATE OR REPLACE FUNCTION postgisftw.countries_name(
    name_prefix text DEFAULT 'A')
RETURNS TABLE(name text, abbrev text, continent text, geom geometry)
AS $$
BEGIN
    RETURN QUERY
        SELECT t.name::text,
```

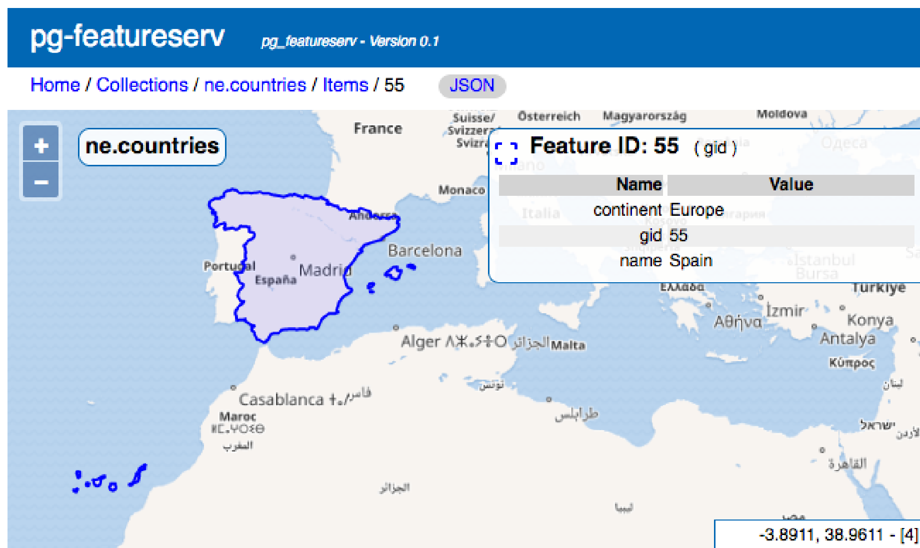

Figure 8: Map view of query for feature by ID

```
        t.abbrev::text,
        t.continent::text,
        t.geom
    FROM ne.admin_0_countries t
    WHERE t.name ILIKE name_prefix || '%';
END;
$$
LANGUAGE 'plpgsql' STABLE PARALLEL SAFE;

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

Example of API query

The function can be called via the API by providing a value for the name_prefix parameter.

http://localhost:9000/functions/countries_name/items?name_prefix=Mo

Since a default value is included in the function declaration, you could omit the parameter in the call – a random sample of features will be returned.

Sample GeoJSON response

The response is a GeoJSON document containing the 7 countries starting with the letters 'Mo'.


```
{
        "type":"FeatureCollection",
        "features":[
            {
              "type":"Feature",
              "geometry":{
                  "type":"MultiPolygon",
                  "coordinates":[
                      [
                          [
                              [
                                  -62.1484375,
                                  16.74033203125
                              ],
                              [
                                -62.154248046875,
                                16.681201171875
                              ],
                            ...
                            [
                              -62.1484375,
                              16.74033203125
                            ]
                      ]
                  ]
              ]
      },
      "properties":{
            "abbrev":"Monts.",
            "continent":"North America",
            "name":"Montserrat"
      }
  },
  ...
 ],
 "numberReturned":7,
 "timeStamp":"2020-03-18T03:15:15Z",
 "links":[
 {
"href":"http://localhost:9000/collections/countries_name/items.json",
           "rel":"self",
```

```
            "type":"application/json",
            "title":"This document as JSON"
        },
        {
"href":"http://localhost:9000/collections/countries_name/items.html",
            "rel":"alternate",
            "type":"text/html",
            "title":"This document as HTML"
        }
    ]
}
```

Web preview

This example shows how to generate geometry data from a function, controlled by some input parameters.

This particular function does not query an existing table in the database; rather, it uses PostGIS functions to generate spatial data. Grids generated in this way could be used for data visualization, analysis, or clustering.

Create a spatial function that generates a grid over a desired area

```
CREATE OR REPLACE FUNCTION postgisftw.geo_grid(
   num_x integer DEFAULT 10,
   num_y integer DEFAULT 10,
   lon_min numeric DEFAULT -180.0,
   lat_min numeric DEFAULT -90.0,
   lon_max numeric DEFAULT 180.0,
   lat_max numeric DEFAULT 90.0)
RETURNS TABLE(id text, geom geometry)
AS $$
DECLARE
     dlon numeric;
     dlat numeric;
BEGIN
     dlon := (lon_max - lon_min) / num_x;
     dlat := (lat_max - lat_min) / num_y;
     RETURN QUERY
          SELECT
```

```
                x.x::text || '_' || y.y::text AS id,
                ST_MakeEnvelope(
                    lon_min + (x.x - 1) * dlon, lat_min + (y.y - 1) * dlat,
                    lon_min + x.x * dlon, lat_min + y.y * dlat, 4326
                ) AS geom
            FROM generate_series(1, num_x) AS x(x)
            CROSS JOIN generate_series(1, num_y) AS y(y);
END;
$$
LANGUAGE 'plpgsql'
STABLE
STRICT;
```

Notes:

- The geo_grid function accepts a num_x and a num_y value to define the number of grid cells along the longitudinal (X) and latitudinal (Y) axes respectively. It also takes in minimum and maximum longitude and latitude values for the map area we want covered.
- The function first calculates the lengths of the sides of the grid (dlon and dlat).
- A CROSS JOIN on two generate_series() functions produces X and Y indices for each grid cell.
- The PostGIS function ST_MakeEnvelope() contructs a rectangular polygon for each cell. An id value is also returned that encodes the grid index.

Example of API query

http://localhost:9000/functions/geo_grid/items?num_x=5&num_y=5&lon_min=−128&lat_min=25&l

This generates a 5x5 grid over the United States.

The server returns a limited number of features by default, so we add a limit parameter in the call to ensure that we get all the grid cells. See *Limiting and Paging* in Executing Functions for more details on the limit parameter.

Sample GeoJSON response

The function returns a feature collection of Polygons.

```
{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "id": "1_1",
      "geometry": {
        "type": "Polygon",
        "coordinates": [
          [
            [
              -128,
              25
            ],
            [
              -128,
              29.8
            ],
            [
              -115.4,
              29.8
            ],
            [
              -115.4,
              25
            ],
            [
              -128,
              25
            ]
          ]
        ]
      },
      "properties": {
        "id": "1_1"
      }
    },
    ...
    {
      "type": "Feature",
      "id": "5_5",
      "geometry": {
        "type": "Polygon",
        "coordinates": [
          [
            [
```

```
                −77.6,
                44.2
              ],
              ...
              [
                −77.6,
                44.2
              ]
            ]
          ]
        },
        "properties": {
          "id": "5_5"
        }
      }
    ],
    "numberReturned": 25,
    "timeStamp": "2020−04−05T19:54:17Z",
    "links": [
      {
        "href": "http://localhost:9000/collections/geo_grid/items.json",
        "rel": "self",
        "type": "application/json",
        "title": "This document as JSON"
      },
      {
        "href": "http://localhost:9000/collections/geo_grid/items.html",
        "rel": "alternate",
        "type": "text/html",
        "title": "This document as HTML"
      }
    ]
}
```

Each cell has an id value that also indicates where it is on the grid. Since longitude and latitude values increase as you move east and north respectively, the cell with id 1_1 is the most southwestern corner of the grid, while cell 1_2 is immediately east and cell 2_1 immediately north.

Web preview

Non-spatial functions (i.e. functions that don't return spatial data) can also be accessed via pg_featureserv, as long as they are published in the postgisftw schema.

The following function example can be used with the ne.countries collection created in the Quick Start section. It shows a function that accepts longitude and latitude values, and returns the corresponding country (if any). Unlike the other function examples in this section, it does not return a table with a geometry type column.

Any kind of function in the postgisftw schema is published, which allows you even more flexible access to data. You can create functions that return statistics, summary records, populate dropdown lists or autocomplete suggestions, and more.

Create a non-spatial function that locates the country for a given coordinate pair

```
CREATE OR REPLACE FUNCTION postgisftw.country_by_loc(
  lon numeric DEFAULT 0.0,
  lat numeric DEFAULT 0.0)
RETURNS TABLE(name text, abbrev text, postal text)
AS $$
BEGIN
      RETURN QUERY
   SELECT c.name::text, c.abbrev::text, c.postal::text
   FROM ne.countries c
   WHERE ST_Intersects(c.geom,
             ST_SetSRID(ST_MakePoint(lon, lat), 4326))
   LIMIT 1;
END;
$$
LANGUAGE 'plpgsql' STABLE STRICT;

COMMENT ON FUNCTION postgisftw.country_by_loc
IS 'Finds␣the␣country␣at␣a␣geographic␣location';
```

Notes:

- The function generates a Point based on the longitude and latitude values provided in the parameters.
- The ne.countries table is filtered based on whether the point intersects a country polygon.
- It's possible that a point lies exactly on the boundary between two countries. Both country records will be included in the query result set, but LIMIT 1 restricts the result to a single record.

Example of API query

The coordinate pair (47,8) can be passed into the function:

http://localhost:9000/functions/country_by_loc/items.json?lat=47&lon=8

Sample JSON response

The service returns data from non-spatial functions in JSON, instead of GeoJSON.


```
[
    {
        "abbrev":"Switz.",
        "name":"Switzerland",
        "postal":"CH"
    }
]
```


This section describes how to obtain, install and run pg_featureserv.

Requirements

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

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

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

Basic operation

The service can be run with minimal configuration. Only the database connection information is required. (The only situtation when this is not needed is when running with the −−test option.)

The database connection information can be provided in an environment variable DATABASE_URL containing a Postgres connection string. It can also be provided in the configuration file DbConnection parameter.

Linux or OSX

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

Windows

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

Command options

Option	Description
−?	Show command usage
−−config <file>.toml	Specify configuration file to use.
−−debug	Set logging level to TRACE (can also be set in config file).
−−devel	Run in development mode. Assets are reloaded on every request.
−−test	Run in test mode. Uses an internal catalog of sample tables and data. Does not r

Configuration file

The configuration file is automatically read from the file config/pg_featureserv.toml in the directory the application starts in, if it exists.

If you want to specify a different file, use the −−config commandline parameter to pass in a full path to the configuration file. When using the −−config option, the local configuration file is ignored.

```
./pg_featureserv --config /opt/pg_featureserv/config.toml
```

If no configuration is specified, the server runs using internal defaults (which are the same as provided in the example configuration file below). Where possible, the program autodetects values such as the UrlBase.

The only required configuration is the DbConnection setting, if not provided in the environment variable DATABASE_URL. (This is not required if the server is run with the −−test flag.)

An example configuration file is shown below.

```
[Server]
# Accept connections on this subnet (default accepts on all)
HttpHost = "0.0.0.0"

# Accept connections on this port
HttpPort = 9000

# Advertise URLs relative to this server name and path
# default is to look this up from incoming request headers
# Note: do not add a trailing slash.
# UrlBase = "http://localhost:9000/"

# String to return for Access-Control-Allow-Origin header
# CORSOrigins = "*"

# set Debug to true to run in debug mode (can also be set on cmd-line)
# Debug = true

# Read html templates from this directory
AssetsPath = "/usr/share/pg_featurserv/assets"

# Maximum duration for reading entire request (in seconds)
ReadTimeoutSec = 1

# Maximum duration for writing response (in seconds)
# Also controls maximum time for processing request
WriteTimeoutSec = 30

[Database]
# Database connection
# postgresql://username:password@host/dbname
# DbConnection = "postgresql://username:password@host/dbname"
```

```
# Close pooled connections after this interval
# 1d, 1h, 1m, 1s, see https://golang.org/pkg/time/#ParseDuration
# DbPoolMaxConnLifeTime = "1h"

# Hold no more than this number of connections in the database pool
# DbPoolMaxConns = 4

[Paging]
# The default number of features in a response
LimitDefault = 20
# Maxium number of features in a response
LimitMax = 10000

[Metadata]
# Title for this service
#Title = "pg-featureserv"
# Description of this service
#Description = "Crunchy Data Feature Server for PostGIS"
```

Configuration options

HttpHost The IP address at which connections are accepted.

HttpPort The IP port at which connections are accepted.

UrlBase The Base URL is the URL endpoint at which the service is advertised. It is also used for any URL paths published by the service (such as URLs for links in response documents).

The UrlBase parameter specifies a value for the Base URL. This accomodates running the service behind a reverse proxy.

The provided URL should not have a trailing slash.

```
UrlBase = https://my-server.org/features
```

If UrlBase is not set, pg_featureserv dynamically detects the base URL. Also, if the HTTP headers Forwarded or X−Forwarded−Proto and X−Forwarded−Host are present, they are respected. Otherwise the base URL is determined by inspecting the incoming request.

CORSOrigins The string to return in the Access−Control−Allow−Origin HTTP header, which allows providing **Cross-Origin Resource Sharing** (CORS).

Debug Set to true to run in debug mode. This provides debug-level logging.

AssetsPath The directory containing file assets used by the service (such as the HTML templates). It may be more convenient to deploy the asset files in a location which is not relative to the service application path.

ReadTimeoutSec The maximum duration (in seconds) the service allows for reading the HTTP request. This can be relatively short, since service requests are small.

WriteTimeoutSec The maximum duration (in seconds) the service allows for processing and writing the HTTP response. This should be long enough to allow expected requests to complete, but not so long that the service can be saturated by long-running requests. Long request times may be caused by long execution times for database queries or functions, or by returning very large responses.

DbConnection The connection to the database can be set in this parameter, using a Postgres connection string. The database connection can also be set via the DATABASE_URL environment variable, which takes precedence over this parameter.

DbPoolMaxConnLifeTime The maximum duration for the lifetime for a pooled connection. Specified using a Go duration constant such as 1d, 2.5h, or 30m.

DbPoolMaxConns The maximum number of database connections held in the connection pool.

LimitDefault The default number of features in a response, if not specified by the limit query parameter.

LimitMax The maximum number of features that can be returned in a response. This cannot be overridden by the limit query paramater.

Title The title for the service. Appears in the HTML web pages, JSON responses, and the log.

Description The description for the service. Appears in the HTML web pages and JSON responses.