pg_featureserv

Contents

pg_featureserv

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. The service API follows the OGC API for Features Version 1.0 standard. It extends the API to expose more of the powerful spatial capabilities of PostGIS.

This guide walks you through how to install and use pg_featureserv for your spatial applications. The Usage section goes in-depth on how the service works. We'll soon be adding more basic examples of web map applications that source feature data from pg_featureserv.

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

Feature Server

The server outputs logging information to the console. By default, the log level is set to show errors and warnings only.

To get more information about what is going on behind the scenes, run the server with the −−debug commandline parameter on

```
./pg_fileserv --debug
```

Or, turn on debugging in the configuration file:

```
Debug = true
```

Web Layer

Hitting the service endpoints with a command-line utility like curl can also yield useful information:

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

Database Layer

The debug mode of the feature server returns the SQL that is being called on the database. If you want 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, bugs can be reported on GitHub at the issue tracker:

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

This section describes how to use pg_featureserv to expose database tables, views and functions; how to query them using the service API; and how to use the Web user interface provided by the service.

The basic principle of security 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.

```
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.

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

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

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

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

```
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.

```
-- 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/home.html
```

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 exposed 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 to allow 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 exposed 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 to allow 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.

OGC API For Features

- landing page
- collections
- feature collection
- feature
- Conformance
- linked data

OpenAPI

Request Headers

Response formats

- links

Error codes and messages

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

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

Expose Tables and Views as Feature Collections

pg_featureserv exposes 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)

Visible tables and views are the available for access by virtue of by the database access permissions defined for the service database user. See the [Security]({{< relref “security” >}}) section for more information.

The service relies on the database catalog information to provide metadata about a table or view. The metadata includes:

- 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 columsn of the table or view
- The description for properties is provided by the comments on table/view columns

List Feature Collections

The path /collections returns a list of the feature collections available in the service as a JSON document.

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

Each listed feature collection provides a name, title, description and extent. A set of links provide the URLs for accessing:

- self - the feature collection metadata
- alt - the feature collection metadata as an HTML view
- items - the feature collection items

Describe Feature Collection metadata

The path / collections /{ coll −name} returns a JSON object describing the metadata for a feature collection. The { 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 ontaining 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 set of links provide the URLs for accessing:

- self - the feature collection metadata
- alt - the feature collection metadata as an HTML view
- items - the feature collection items

List Functions

/functions

/functions.json

Describe Function metadata

/functions/{funid}

/functions/{funid}.json

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.admin_0_countries/items
```

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

These are analagous to using SQL statement clauses to control the results of a query. In fact the service implements these using exactly that technique. This provides maximum performance since it allows the Postgres SQL engine to optimize the query execution plan.

Limiting and paging results

The query parameter limit=N can be added to the query to control 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.

Used together these two parameters allow paging through large result sets.

```
http://localhost:9000/collections/ne.admin_0_countries/items?limit=50&offset=
```

Ordering results

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.admin_0_countries/items?orderBy=name
```

Filter by bbox

The query parameter bbox=MINX,MINY,MAXX,MAXY can be used to limit the features returned to those that intersect a specified bounding box. The bounding box is specified in geographic coordinates (longitude/latitude).

```
http://localhost:9000/collections/ne.admin_0_countries/items?bbox=10.4,43.3,2
```

Specify result properties

The query parameter properties=PROP1,PROP2,PROP3... can be used to restrict the properties which are returned in the response. This allows reducing the response size of feature collections which have a large number of properties.

```
http://localhost:9000/collections/ne.admin_0_countries/items?properties=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.admin_0_countries/items/23
```

Specify properties in result

The query parameter properties=PROP1,PROP2,PROP3... can be used to restrict the properties which are returned in the response.

```
http://localhost:9000/collections/ne.admin_0_countries/items/23?properties=nar
```

Query Function features or data

/functions/{funid}/items /functions/{funid}/items.json

- response is GeoJSON for result dataset

Function arguments

param=value

Omitted arguments will use the default specified in the function definition (if any).

Limiting and paging results

limit=N

offset=N

Ordering results

orderBy=PROP

orderBy=PROP:A

orderBy=PROP:D

Filter by bbox

bbox=MINX,MINY,MAXX,MAXY

- extent is in lon/lat (4326)

Specify properties in result

properties=PROP1,PROP2,PROP3...

We're currently working on adding examples of using pg_featureserv in this guide.

In the meantime, we'd encourage you to check this Github repository for a heat map demo and an address autocomplete demo.

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 might 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.

Configuration File

The configuration file is automatically read from the file config.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/pg_featureserv.toml
```

If no configuration is specified, the server runs using internal defaults (which are the same as provided in the example configuration file). 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. (Even this can be omitted if the server is run with the --test flag.)

The default configuration file is shown below.

```
[Server]
# The hostname to use in links
HttpHost = "0.0.0.0"

# The IP port to listen on
HttpPort = 9000

# Advertise URLs relative to this server name
# default is to look this up from incoming request headers
# 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 done on cmd-line)
# Debug = true

# Read html templates from this directory
```

```
AssetsPath = "./assets"

[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

UrlBase

The Base URL is the URL endpoint at which users access the service. It is also used for any URL paths returned by the service (such as response links).

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

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.

Basic Operation

The service can be run with minimal configuration. Only the database connection information is required. (Even that can be omitted if run with the −−test option.) The database connection information can be provided in an environment variable DATABASE_URL containing a Postgres connection string.

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

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://user:pass@host/dbname -p 9000:9000 pram
```

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://user:pass@host/dbname
$GOPATH/bin/pg_featureserv
```

Motivation

There are numerous services available which 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 tend to be large, complex applications which require significnat expertise to install, configure, secure and tune.

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. It is also straightforward to 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 exposed 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 expose any already-developed 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 utilizing 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.

A key benefit of following the lightweight OGC API for Features Core standard is the ease of extending it to expose service-specific capabilities. For instance, pg_featureserv allows querying 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 database 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.

pg_featureserv has a simple architecture. It consists of a single server application, written in Go. It is configured via static (read-only) information sourced from a file, 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 contains 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.

The pg_featureserv server integrates with the following:

- a PostGIS-enabled Postgres database instance or cluster, containing the data being served and the catalog metadata describing it.
- 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".

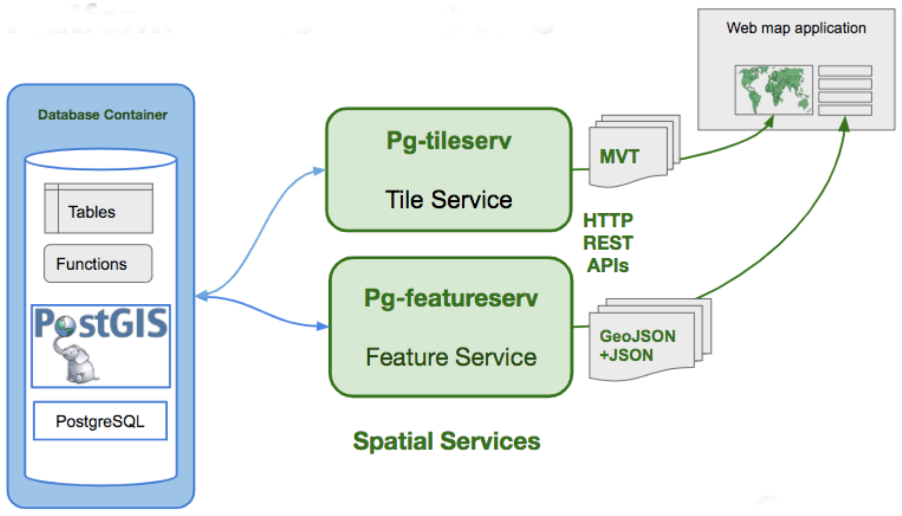

Figure 1: pg_feaureserv Architecture

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