PostgREST Documentation

Release 5.0.0

Joe Nelson

Aug 27, 2020

Release Notes

PostgREST is a standalone web server that turns your PostgreSQL database directly into a RESTful API. The structural constraints and permissions in the database determine the API endpoints and operations.

CHAPTER 1

Motivation

Using PostgREST is an alternative to manual CRUD programming. Custom API servers suffer problems. Writing business logic often duplicates, ignores or hobbles database structure. Object-relational mapping is a leaky abstraction leading to slow imperative code. The PostgREST philosophy establishes a single declarative source of truth: the data itself.

CHAPTER 2

Declarative Programming

It's easier to ask PostgreSQL to join data for you and let its query planner figure out the details than to loop through rows yourself. It's easier to assign permissions to db objects than to add guards in controllers. (This is especially true for cascading permissions in data dependencies.) It's easier to set constraints than to litter code with sanity checks.

CHAPTER 3

Leak-proof Abstraction

There is no ORM involved. Creating new views happens in SQL with known performance implications. A database administrator can now create an API from scratch with no custom programming.

CHAPTER 4

Embracing the Relational Model

In 1970 E. F. Codd criticized the then-dominant hierarchical model of databases in his article A Relational Model of Data for Large Shared Data Banks. Reading the article reveals a striking similarity between hierarchical databases and nested http routes. With PostgREST we attempt to use flexible filtering and embedding rather than nested routes.

CHAPTER 5

One Thing Well

PostgREST has a focused scope. It works well with other tools like Nginx. This forces you to cleanly separate the data-centric CRUD operations from other concerns. Use a collection of sharp tools rather than building a big ball of mud.

CHAPTER 6

Shared Improvements

As with any open source project, we all gain from features and fixes in the tool. It's more beneficial than improvements locked inextricably within custom code-bases.

CHAPTER 7

Getting Support

The project has a friendly and growing community. Join our chat room for discussion and help. You can also report or search for bugs/features on the Github issues page.

CHAPTER 8

Supporting development

You can help PostgREST ongoing maintenance and development by:

- Making a regular donation through Patreon
- Alternatively, you can make a one-time donation via Paypal

Every donation will be spent on making PostgREST better for the whole community.

8.1 Translations

- Chinese (latest version `v0.4.2.0`)

8.1.1 Release Notes

Here we'll include the most relevant changes so you can migrate to newer versions easily. You can see the full changelog of each release in the PostgREST repository.

8.1.2 v5.2.0

- Explicit qualification introduced in `v5.0` is no longer necessary, this section will not be included from this version onwards. A *db-extra-search-path* configuration parameter was introduced to avoid the need to explictly qualify database objects. If you install PostgreSQL extensions on the `public` schema, they'll work normally from now on.
- Now you can filter *Table / Columns with spaces*.
- Included the ability to quote columns that have *Reserved characters*.
- Thanks to Zhou Feng, now is possible to reference an external file in *db-uri*.
- Thanks to Russell Davies, Json Web Key Sets are now accepted by *jwt-secret*.

Thanks

This release was made possible thanks to:

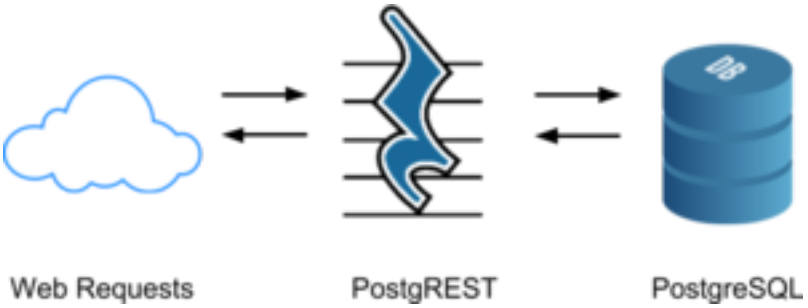
- Daniel Babiak
- Michel Pelletier
- Tsingson Qin
- Jay Hannah
- Victor Adossi
- Petr Beles

If you like to join them please consider *supporting PostgREST development*.

8.1.3 Tutorial 0 - Get it Running

Welcome to PostgREST! In this pre-tutorial we're going to get things running so you can create your first simple API.

PostgREST is a standalone web server which turns a PostgreSQL database into a RESTful API. It serves an API that is customized based on the structure of the underlying database.



To make an API we'll simply be building a database. All the endpoints and permissions come from database objects like tables, views, roles, and stored procedures. These tutorials will cover a number of common scenarios and how to model them in the database.

By the end of this tutorial you'll have a working database, PostgREST server, and a simple single-user todo list API.

Step 1. Relax, we'll help

As you begin the tutorial, pop open the project chat room in another tab. There are a nice group of people active in the project and we'll help you out if you get stuck.

Step 2. Install PostgreSQL

You'll need a modern copy of the database running on your system, either natively or in a Docker instance. We require PostgreSQL 9.3 or greater, but recommend at least 9.5 for row-level security features that we'll use in future tutorials.

If you're already familiar with using PostgreSQL and have it installed on your system you can use the existing installation. For this tutorial we'll describe how to use the database in Docker because database configuration is otherwise too complicated for a simple tutorial.

If Docker is not installed, you can get it here. Next, let's pull and start the database image:

```
sudo docker run --name tutorial -p 5433:5432 \
                -e POSTGRES_PASSWORD=mysecretpassword \
                -d postgres
```

This will run the Docker instance as a daemon and expose port 5433 to the host system so that it looks like an ordinary PostgreSQL server to the rest of the system.

Step 3. Install PostgREST

PostgREST is distributed as a single binary, with versions compiled for major distributions of Linux/BSD/Windows. Visit the latest release for a list of downloads. In the event that your platform is not among those already pre-built, see *Build from Source* for instructions how to build it yourself. Also let us know to add your platform in the next release.

The pre-built binaries for download are `.tar.xz` compressed files (except Windows which is a zip file). To extract the binary, go into the terminal and run

```
# download from https://github.com/PostgREST/postgrest/releases/latest

tar xfJ postgrest-<version>-<platform>.tar.xz
```

The result will be a file named simply `postgrest` (or `postgrest.exe` on Windows). At this point try running it with

```
./postgrest
```

If everything is working correctly it will print out its version and information about configuration. You can continue to run this binary from where you downloaded it, or copy it to a system directory like `/usr/local/bin` on Linux so that you will be able to run it from any directory.

Note: PostgREST requires libpq, the PostgreSQL C library, to be installed on your system. Without the library you'll get an error like "error while loading shared libraries: libpq.so.5." Here's how to fix it:

Step 4. Create Database for API

Connect to the SQL console (psql) inside the container. To do so, run this from your command line:

```
sudo docker exec -it tutorial psql -U postgres
```

You should see the psql command prompt:

```
psql (9.6.3)
Type "help" for help.

postgres=#
```

The first thing we'll do is create a named schema for the database objects which will be exposed in the API. We can choose any name we like, so how about "api." Execute this and the other SQL statements inside the psql prompt you started.

```
create schema api;
```

Our API will have one endpoint, `/todos`, which will come from a table.

```
create table api.todos (
  id serial primary key,
  done boolean not null default false,
  task text not null,
  due timestamptz
);

insert into api.todos (task) values
  ('finish tutorial 0'), ('pat self on back');
```

Next make a role to use for anonymous web requests. When a request comes in, PostgREST will switch into this role in the database to run queries.

```
create role web_anon nologin;

grant usage on schema api to web_anon;
grant select on api.todos to web_anon;
```

The `web_anon` role has permission to access things in the `api` schema, and to read rows in the `todos` table.

It's a good practice to create a dedicated role for connecting to the database, instead of using the highly privileged `postgres` role. So we'll do that, name the role `authenticator` and also grant him the ability to switch to the `web_anon` role :

```
create role authenticator noinherit login password 'mysecretpassword';
grant web_anon to authenticator;
```

Now quit out of psql; it's time to start the API!

```
\q
```

Step 5. Run PostgREST

PostgREST uses a configuration file to tell it how to connect to the database. Create a file `tutorial.conf` with this inside:

```
db-uri = "postgres://authenticator:mysecretpassword@localhost:5433/postgres"
db-schema = "api"
db-anon-role = "web_anon"
```

The configuration file has other *options*, but this is all we need. Now run the server:

```
./postgrest tutorial.conf
```

You should see

```
Listening on port 3000
Attempting to connect to the database...
Connection successful
```

It's now ready to serve web requests. There are many nice graphical API exploration tools you can use, but for this tutorial we'll use `curl` because it's likely to be installed on your system already. Open a new terminal (leaving the one open that PostgREST is running inside). Try doing an HTTP request for the todos.

```
curl http://localhost:3000/todos
```

The API replies:

```
[
  {
    "id": 1,
    "done": false,
    "task": "finish tutorial 0",
    "due": null
  },
  {
    "id": 2,
    "done": false,
    "task": "pat self on back",
    "due": null
  }
]
```

With the current role permissions, anonymous requests have read-only access to the `todos` table. If we try to add a new todo we are not able.

```
curl http://localhost:3000/todos -X POST \
     -H "Content-Type: application/json" \
     -d '{"task": "do bad thing"}'
```

Response is 401 Unauthorized:

```
{
  "hint": null,
  "details": null,
  "code": "42501",
  "message": "permission denied for relation todos"
}
```

There we have it, a basic API on top of the database! In the next tutorials we will see how to extend the example with more sophisticated user access controls, and more tables and queries.

Now that you have PostgREST running, try the next tutorial, *Tutorial 1 - The Golden Key*

8.1.4 Tutorial 1 - The Golden Key

In *Tutorial 0 - Get it Running* we created a read-only API with a single endpoint to list todos. There are many directions we can go to make this API more interesting, but one good place to start would be allowing some users to change data in addition to reading it.

Step 1. Add a Trusted User

The previous tutorial created a `web_anon` role in the database with which to execute anonymous web requests. Let's make a role called `todo_user` for users who authenticate with the API. This role will have the authority to do anything to the todo list.

```
-- run this in psql using the database created
-- in the previous tutorial

create role todo_user nologin;
grant todo_user to authenticator;
```

(continues on next page)

(continued from previous page)

```
grant usage on schema api to todo_user;
grant all on api.todos to todo_user;
grant usage, select on sequence api.todos_id_seq to todo_user;
```

Step 2. Make a Secret

Clients authenticate with the API using JSON Web Tokens. These are JSON objects which are cryptographically signed using a password known to only us and the server. Because clients do not know the password, they cannot tamper with the contents of their tokens. PostgREST will detect counterfeit tokens and will reject them.

Let's create a password and provide it to PostgREST. Think of a nice long one, or use a tool to generate it. **Your password must be at least 32 characters long.**

Note: Unix tools can generate a nice password for you:

```
# Allow "tr" to process non-utf8 byte sequences
export LC_CTYPE=C

# read random bytes and keep only alphanumerics
< /dev/urandom tr -dc A-Za-z0-9 | head -c32
```

Open the `tutorial.conf` (created in the previous tutorial) and add a line with the password:

```
# PASSWORD MUST BE AT LEAST 32 CHARS LONG
# add this line to tutorial.conf:

jwt-secret = "<the password you made>"
```

If the PostgREST server is still running from the previous tutorial, restart it to load the updated configuration file.

Step 3. Sign a Token

Ordinarily your own code in the database or in another server will create and sign authentication tokens, but for this tutorial we will make one "by hand." Go to jwt.io and fill in the fields like this:

Remember to fill in the password you generated rather than the word "secret". After you have filled in the password and payload, the encoded data on the left will update. Copy the encoded token.

Note: While the token may look well obscured, it's easy to reverse engineer the payload. The token is merely signed, not encrypted, so don't put things inside that you don't want a determined client to see.

Step 4. Make a Request

Back in the terminal, let's use `curl` to add a todo. The request will include an HTTP header containing the authentication token.

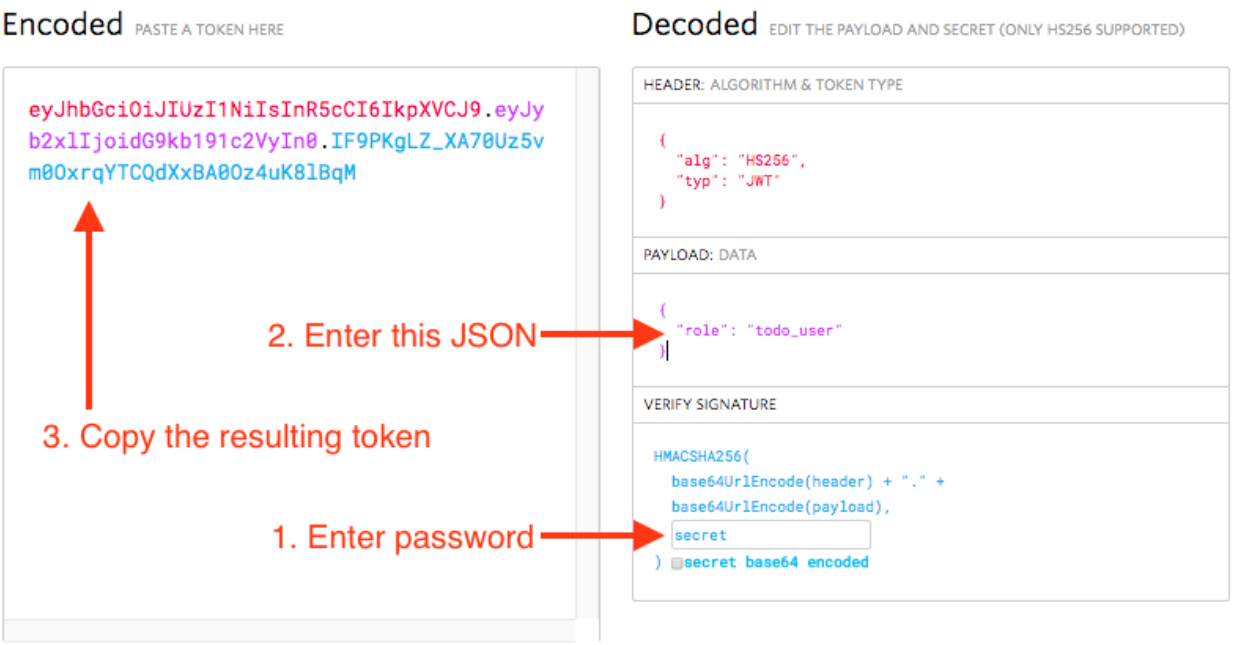


Fig. 1: How to create a token at https://jwt.io

```
export TOKEN="<paste token here>"

curl http://localhost:3000/todos -X POST \
     -H "Authorization: Bearer $TOKEN"   \
     -H "Content-Type: application/json" \
     -d '{"task": "learn how to auth"}'
```

And now we have completed all three items in our todo list, so let's set `done` to true for them all with a `PATCH` request.

```
curl http://localhost:3000/todos -X PATCH \
     -H "Authorization: Bearer $TOKEN"    \
     -H "Content-Type: application/json"  \
     -d '{"done": true}'
```

A request for the todos shows three of them, and all completed.

```
curl http://localhost:3000/todos
```

```
[
  {
    "id": 1,
    "done": true,
    "task": "finish tutorial 0",
    "due": null
  },
  {
    "id": 2,
    "done": true,
    "task": "pat self on back",
    "due": null
```

(continues on next page)

(continued from previous page)

```
  },
  {
    "id": 3,
    "done": true,
    "task": "learn how to auth",
    "due": null
  }
]
```

Step 4. Add Expiration

Currently our authentication token is valid for all eternity. The server, as long as it continues using the same JWT password, will honor the token.

It's better policy to include an expiration timestamp for tokens using the `exp` claim. This is one of two JWT claims that PostgREST treats specially.

Claim	Interpretation
`role`	The database role under which to execute SQL for API request
`exp`	Expiration timestamp for token, expressed in "Unix epoch time"

Note: Epoch time is defined as the number of seconds that have elapsed since 00:00:00 Coordinated Universal Time (UTC), January 1st 1970, minus the number of leap seconds that have taken place since then.

To observe expiration in action, we'll add an `exp` claim of five minutes in the future to our previous token. First find the epoch value of five minutes from now. In psql run this:

```
select extract(epoch from now() + '5 minutes'::interval) :: integer;
```

Go back to jwt.io and change the payload to

```
{
  "role": "todo_user",
  "exp": <computed epoch value>
}
```

Copy the updated token as before, and save it as a new environment variable.

```
export NEW_TOKEN="<paste new token>"
```

Try issuing this request in curl before and after the expiration time:

```
curl http://localhost:3000/todos \
     -H "Authorization: Bearer $NEW_TOKEN"
```

After expiration, the API returns HTTP 401 Unauthorized:

```
{"message":"JWT expired"}
```

Bonus Topic: Immediate Revocation

Even with token expiration there are times when you may want to immediately revoke access for a specific token. For instance, suppose you learn that a disgruntled employee is up to no good and his token is still valid.

To revoke a specific token we need a way to tell it apart from others. Let's add a custom `email` claim that matches the email of the client issued the token.

Go ahead and make a new token with the payload

```
{
  "role": "todo_user",
  "email": "disgruntled@mycompany.com"
}
```

Save it to an environment variable:

```
export WAYWARD_TOKEN="<paste new token>"
```

PostgREST allows us to specify a stored procedure to run during attempted authentication. The function can do whatever it likes, including raising an exception to terminate the request.

First make a new schema and add the function:

```
create schema auth;
grant usage on schema auth to web_anon, todo_user;

create or replace function auth.check_token() returns void
  language plpgsql
  as $$
begin
  if current_setting('request.jwt.claim.email', true) =
     'disgruntled@mycompany.com' then
    raise insufficient_privilege
      using hint = 'Nope, we are on to you';
  end if;
end
$$;
```

Next update `tutorial.conf` and specify the new function:

```
# add this line to tutorial.conf

pre-request = "auth.check_token"
```

Restart PostgREST for the change to take effect. Next try making a request with our original token and then with the revoked one.

```
# this request still works

curl http://localhost:3000/todos -X PATCH \
     -H "Authorization: Bearer $TOKEN"    \
     -H "Content-Type: application/json"  \
     -d '{"done": true}'

# this one is rejected

curl http://localhost:3000/todos -X PATCH      \
```

(continues on next page)

(continued from previous page)

```
    -H "Authorization: Bearer $WAYWARD_TOKEN" \
    -H "Content-Type: application/json"       \
    -d '{"task": "AAAHHHH!", "done": false}'
```

The server responds with 403 Forbidden:

```
{
  "hint": "Nope, we are on to you",
  "details": null,
  "code": "42501",
  "message": "insufficient_privilege"
}
```

8.1.5 TimescaleDB for Time-Series Data

TimescaleDB is an open-source database designed to make SQL scalable for time-series data. It is engineered up from PostgreSQL, providing automatic partitioning across time and space, while retaining the standard PostgreSQL interface.

PostgREST turns your PostgreSQL database directly into a RESTful API, since TimescaleDB is packaged as a PostgreSQL extension it works with PostgREST as well.

In this tutorial we'll explore some of TimescaleDB features through PostgREST.

Install Docker

For an easier setup we're going to use Docker, make sure you have it installed.

Run TimescaleDB

First, let's pull and start the TimescaleDB container image:

```
docker run --name tsdb_tut \
  -e POSTGRES_PASSWORD=mysecretpassword \
  -p 5433:5432 \
  -d timescale/timescaledb:latest-pg11
```

This will run the container as a daemon and expose port `5433` to the host system so that it doesn't conflict with another PostgreSQL installation.

Set up TimescaleDB

Now, we'll create the `timescaledb` extension in our database.

Run `psql` in the container we created in the previous step.

```
docker exec -it tsdb_tut psql -U postgres
## Run all the following commands inside psql
```

And create the extension:

```
create extension if not exists timescaledb cascade;
```

Create an Hypertable

Hypertables are the core abstraction TimescaleDB offers for dealing with time-series data.

To create an `hypertable`, first we need to create standard PostgreSQL tables:

```
create table if not exists locations(
  device_id    text  primary key
, location     text
, environment  text
);

create table if not exists conditions(
  time         timestamp with time zone  not null
, device_id    text                      references locations(device_id)
, temperature  numeric
, humidity     numeric
);
```

Now, we'll convert `conditions` into an hypertable with create_hypertable:

```
SELECT create_hypertable('conditions', 'time', chunk_time_interval => interval '1 day
↪');
-- This also implicitly creates an index: CREATE INDEX ON "conditions"(time DESC);

-- Exit psql
exit
```

Load sample data

To have some data to play with, we'll download the `weather_small` data set from TimescaleDB's sample datasets.

```
## Run bash inside the database container
docker exec -it tsdb_tut bash

## Download and uncompress the data
wget -qO- https://timescaledata.blob.core.windows.net/datasets/weather_small.tar.gz |␣
↪tar xvz

## Copy data into the database
psql -U postgres <<EOF
  \COPY locations  FROM weather_small_locations.csv  CSV
  \COPY conditions FROM weather_small_conditions.csv CSV
EOF

## Exit bash
exit
```

Run PostgREST

For the last step in the setup, pull and start the official PostgREST image:

```
docker run --rm -p 3000:3000 \
  --name tsdb_pgrst \
  --link tsdb_tut \
```

(continues on next page)

(continued from previous page)

```
  -e PGRST_DB_URI="postgres://postgres:mysecretpassword@tsdb_tut/postgres" \
  -e PGRST_DB_ANON_ROLE="postgres" \
  -d postgrest/postgrest:latest
```

PostgREST on Hypertables

We'll now see how to read data from hypertables through PostgREST.

Since hypertables can be queried using standard SELECT statements, we can query them through PostgREST normally.

Suppose we want to run this query on `conditions`:

```
select
  time,
  device_id,
  humidity
from conditions
where
  humidity > 90           and
  time     < '2016-11-16'
order by time desc
limit 10;
```

Using PostgREST *horizontal*/*vertical* filtering, this query can be expressed as:

```
curl -G "localhost:3000/conditions" \
  -d select=time,device_id,humidity \
  -d humidity=gt.90 \
  -d time=lt.2016-11-16 \
  -d order=time.desc \
  -d limit=10
## This command is equivalent to:
## curl "localhost:3000/conditions?select=time,device_id,humidity&humidity=gt.90&
↪time=lt.2016-11-16&order=time.desc&limit=10"
## Here we used -G and -d to make the command more readable
```

The response will be:

```
[{"time":"2016-11-15T23:58:00+00:00","device_id":"weather-pro-000982","humidity":90.
↪90000000000006},
 {"time":"2016-11-15T23:58:00+00:00","device_id":"weather-pro-000968","humidity":92.3}
↪,
 {"time":"2016-11-15T23:58:00+00:00","device_id":"weather-pro-000963","humidity":96.
↪29999999999993},
 {"time":"2016-11-15T23:58:00+00:00","device_id":"weather-pro-000951","humidity":94.
↪39999999999998},
 {"time":"2016-11-15T23:58:00+00:00","device_id":"weather-pro-000950","humidity":93.
↪69999999999982},
 {"time":"2016-11-15T23:58:00+00:00","device_id":"weather-pro-000915","humidity":94.
↪69999999999997},
 {"time":"2016-11-15T23:58:00+00:00","device_id":"weather-pro-000911","humidity":93.
↪2000000000001},
 {"time":"2016-11-15T23:58:00+00:00","device_id":"weather-pro-000910","humidity":91.
↪30000000000017},
```

(continues on next page)

(continued from previous page)

```
 {"time":"2016-11-15T23:58:00+00:00","device_id":"weather-pro-000901","humidity":92.
↪30000000000005},
 {"time":"2016-11-15T23:58:00+00:00","device_id":"weather-pro-000895","humidity":91.
↪00000000000014}]
```

JOINs with relational tables

Hypertables support all standard PostgreSQL constraints . We can make use of the foreign key defined on `locations` to make a JOIN through PostgREST. A query such as:

```
select
  c.time,
  c.temperature,
  l.location,
  l.environment
from conditions c
left join locations l on
  c.device_id = l.device_id
order by time desc
limit 10;
```

Can be expressed in PostgREST by using *Resource Embedding*.

```
curl -G localhost:3000/conditions \
  -d select="time,temperature,device:locations(location,environment)" \
  -d order=time.desc \
  -d limit=10
```

```
[{"time":"2016-11-16T21:18:00+00:00","temperature":69.49999999999991,"device":{
↪"location":"office-000202","environment":"inside"}},
 {"time":"2016-11-16T21:18:00+00:00","temperature":90,"device":{"location":"field-
↪000205","environment":"outside"}},
 {"time":"2016-11-16T21:18:00+00:00","temperature":60.499999999999986,"device":{
↪"location":"door-00085","environment":"doorway"}},
 {"time":"2016-11-16T21:18:00+00:00","temperature":91,"device":{"location":"swamp-
↪000188","environment":"outside"}},
 {"time":"2016-11-16T21:18:00+00:00","temperature":42,"device":{"location":"arctic-
↪000219","environment":"outside"}},
 {"time":"2016-11-16T21:18:00+00:00","temperature":70.80000000000003,"device":{
↪"location":"office-000201","environment":"inside"}},
 {"time":"2016-11-16T21:18:00+00:00","temperature":62.699999999999974,"device":{
↪"location":"door-00084","environment":"doorway"}},
 {"time":"2016-11-16T21:18:00+00:00","temperature":85.49999999999918,"device":{
↪"location":"field-000204","environment":"outside"}},
 {"time":"2016-11-16T21:18:00+00:00","temperature":42,"device":{"location":"arctic-
↪000218","environment":"outside"}},
 {"time":"2016-11-16T21:18:00+00:00","temperature":42,"device":{"location":"arctic-
↪000217","environment":"outside"}}]
```

Time-Oriented Analytics

TimescaleDB includes new aggregate functions for time-oriented analytics.

For using aggregate queries with PostgREST you must create VIEWs or *Stored Procedures*. Here's an example for using time_bucket:

```
-- Run psql in the database container
docker exec -it tsdb_tut psql -U postgres

-- Create the function
create or replace function temperature_summaries(gap interval default '1 hour',␣
↪prefix text default 'field')
returns table(hour text, avg_temp numeric, min_temp numeric, max_temp numeric) as $$
  select
    time_bucket(gap, time)::text as hour,
    trunc(avg(temperature), 2),
    trunc(min(temperature), 2),
    trunc(max(temperature), 2)
  from conditions c
  where c.device_id in (
    select device_id from locations
    where location like prefix || '-%')
  group by hour
$$ language sql stable;

-- Exit psql
exit
```

Every time the schema is changed you must reload PostgREST *schema cache* so it can pick up the function parameters correctly. To reload, run:

```
docker kill --signal=USR1 tsdb_pgrst
```

Now, since the function is `stable`, we can call it with `GET` as:

```
curl -G "localhost:3000/rpc/temperature_summaries" \
  -d gap=2minutes \
  -d order=hour.asc \
  -d limit=10 \
  -H "Accept: text/csv"
## time_bucket accepts an interval type as it's argument
## so you can pass gap=5minutes or gap=5hours
```

```
hour,avg_temp,min_temp,max_temp
"2016-11-15 12:00:00+00",72.97,68.00,78.00
"2016-11-15 12:02:00+00",73.01,68.00,78.00
"2016-11-15 12:04:00+00",73.05,68.00,78.10
"2016-11-15 12:06:00+00",73.07,68.00,78.10
"2016-11-15 12:08:00+00",73.11,68.00,78.10
"2016-11-15 12:10:00+00",73.14,68.00,78.10
"2016-11-15 12:12:00+00",73.17,68.00,78.19
"2016-11-15 12:14:00+00",73.21,68.10,78.19
"2016-11-15 12:16:00+00",73.24,68.10,78.29
"2016-11-15 12:18:00+00",73.27,68.10,78.39
```

Note you can use PostgREST standard filtering on function results. Here we also changed the *Response Format* to CSV.

Fast Ingestion with Bulk Insert

You can use PostgREST *Bulk Insert* to leverage TimescaleDB fast ingestion.

Let's do an insert of three rows:

```
curl "localhost:3000/conditions" \
  -H "Content-Type: application/json" \
  -H "Prefer: return=representation" \
  -d @- << EOF
  [
    {"time": "2019-02-21 01:00:01-05", "device_id": "weather-pro-000000", "temperature
↪": 40.0, "humidity": 59.9},
    {"time": "2019-02-21 01:00:02-05", "device_id": "weather-pro-000000", "temperature
↪": 42.0, "humidity": 69.9},
    {"time": "2019-02-21 01:00:03-05", "device_id": "weather-pro-000000", "temperature
↪": 44.0, "humidity": 79.9}
  ]
EOF
```

By using the `Prefer: return=representation` header we can see the successfully inserted rows:

```
[{"time":"2019-02-21T06:00:01+00:00","device_id":"weather-pro-000000","temperature
↪":40.0,"humidity":59.9},
 {"time":"2019-02-21T06:00:02+00:00","device_id":"weather-pro-000000","temperature
↪":42.0,"humidity":69.9},
 {"time":"2019-02-21T06:00:03+00:00","device_id":"weather-pro-000000","temperature
↪":44.0,"humidity":79.9}]
```

Let's now insert a thousand rows, we'll use jq for constructing the array.

```
yes "{\"time\": \"$(date +'%F %T')\", \"device_id\": \"weather-pro-000001\", \
↪"temperature\": 50, \"humidity\": 60}" | \
head -n 1000 | jq -s '.' | \
curl -i -d @- "http://localhost:3000/conditions" \
  -H "Content-Type: application/json" \
  -H "Prefer: count=exact"
```

With `Prefer: count=exact` we can know how many rows were inserted. Check out the response:

```
HTTP/1.1 201 Created
Transfer-Encoding: chunked
Date: Fri, 22 Feb 2019 16:47:05 GMT
Server: postgrest/5.2.0 (9969262)
Content-Range: */1000
```

You can see in `Content-Range` that the total number of inserted rows is `1000`.

Summing it up

There you have it, with PostgREST you can get an instant and performant RESTful API for a TimescaleDB database.

For a more in depth exploration of TimescaleDB capabilities, check their docs.

8.1.6 Binary Release

[Download from release page]

The release page has pre-compiled binaries for Mac OS X, Windows, and several Linux distributions. Extract the tarball and run the binary inside with the `--help` flag to see usage instructions:

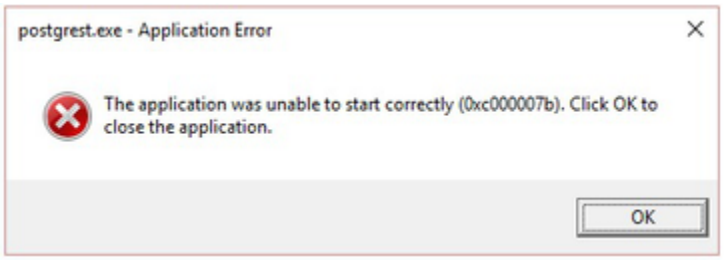
```
# Untar the release (available at https://github.com/PostgREST/postgrest/releases/
↪latest)

$ tar Jxf postgrest-[version]-[platform].tar.xz

# Try running it
$ ./postgrest --help

# You should see a usage help message
```

Note: If you see a dialog box like this on Windows, it may be that the `pg_config` program is not in your system path.



It usually lives in `C:\Program Files\PostgreSQL\<version>\bin`. See this article about how to modify the system path.

8.1.7 PostgreSQL dependency

To use PostgREST you will need an underlying database (PostgreSQL version 9.5 or greater is required). You can use something like Amazon RDS but installing your own locally is cheaper and more convenient for development.

- Instructions for OS X
- Instructions for Ubuntu 14.04
- Installer for Windows

On Windows, PostgREST will fail to run unless the PostgreSQL binaries are on the system path. To test whether this is the case, run `pg_config` from the command line. You should see it output a list of paths.

8.1.8 Configuration

The PostgREST server reads a configuration file to determine information about the database and how to serve client requests. There is no predefined location for this file, you must specify the file path as the one and only argument to the server:

```
./postgrest /path/to/postgrest.conf
```

The file must contain a set of key value pairs. At minimum you must include these keys:

```
# postgrest.conf

# The standard connection URI format, documented at
# https://www.postgresql.org/docs/current/static/libpq-connect.html#AEN45347
db-uri       = "postgres://user:pass@host:5432/dbname"

# The name of which database schema to expose to REST clients
db-schema    = "api"

# The database role to use when no client authentication is provided.
# Can (and probably should) differ from user in db-uri
db-anon-role = "anon"
```

The user specified in the db-uri is also known as the authenticator role. For more information about the anonymous vs authenticator roles see the *Overview of Role System*.

Here is the full list of configuration parameters.

Name	Type	Default	Required
db-uri	String		Y
db-schema	String		Y
db-anon-role	String		Y
db-pool	Int	10	
db-extra-search-path	String	public	
server-host	String	127.0.0.1	
server-port	Int	3000	
server-proxy-uri	String		
jwt-secret	String		
jwt-aud	String		
secret-is-base64	Bool	False	
max-rows	Int	∞	
pre-request	String		
app.settings.*	String		
role-claim-key	String	.role	

db-uri

The standard connection PostgreSQL URI format. Symbols and unusual characters in the password or other fields should be percent encoded to avoid a parse error. If enforcing an SSL connection to the database is required you can use sslmode in the URI, for example `postgres://user:pass@host:5432/dbname?sslmode=require`.

When running PostgREST on the same machine as PostgreSQL, it is also possible to connect to the database using a Unix socket and the Peer Authentication method as an alternative to TCP/IP communication and authentication with a password, this also grants higher performance. To do this you can omit the host and the password, e.g. `postgres://user@/dbname`, see the libpq connection string documentation for more details.

On older systems like Centos 6, with older versions of libpq, a different db-uri syntax has to be used. In this case the URI is a string of space separated key-value pairs (key=value), so the example above would be `"host=host user=user port=5432 dbname=dbname password=pass"`.

Choosing a value for this parameter beginning with the at sign such as `@filename` (e.g. `@./configs/my-config`) loads the secret out of an external file.

db-schema

The database schema to expose to REST clients. Tables, views and stored procedures in this schema will get API endpoints.

This schema gets added to the search_path of every request.

db-anon-role

The database role to use when executing commands on behalf of unauthenticated clients.

db-pool

Number of connections to keep open in PostgREST's database pool. Having enough here for the maximum expected simultaneous client connections can improve performance. Note it's pointless to set this higher than the `max_connections` GUC in your database.

db-extra-search-path

Extra schemas to add to the search_path of every request. These schemas tables, views and stored procedures don't get API endpoints, they can only be referred from the database objects exposed in your *db-schema*.

Multiple schemas can be added in a comma-separated string, e.g. `public, extensions`.

server-host

Where to bind the PostgREST web server. In addition to the usual address options, PostgREST interprets these reserved addresses with special meanings:

- `*` - any IPv4 or IPv6 hostname
- `*4` - any IPv4 or IPv6 hostname, IPv4 preferred
- `!4` - any IPv4 hostname
- `*6` - any IPv4 or IPv6 hostname, IPv6 preferred
- `!6` - any IPv6 hostname

server-port

The port to bind the web server.

server-proxy-uri

Overrides the base URL used within the OpenAPI self-documentation hosted at the API root path. Use a complete URI syntax `scheme:[//[user:password@]host[:port]][/]path[?query][#fragment]`. Ex. `https://postgrest.com`

```
{
  "swagger": "2.0",
  "info": {
    "version": "0.4.3.0",
    "title": "PostgREST API",
    "description": "This is a dynamic API generated by PostgREST"
  },
  "host": "postgrest.com:443",
  "basePath": "/",
  "schemes": [
    "https"
  ]
}
```

jwt-secret

The secret or JSON Web Key (JWK) (or set) used to decode JWT tokens clients provide for authentication. For security the key must be **at least 32 characters long**. If this parameter is not specified then PostgREST refuses authentication requests. Choosing a value for this parameter beginning with the at sign such as `@filename` loads the secret out of an external file. This is useful for automating deployments. Note that any binary secrets must be base64 encoded. Both symmetric and asymmetric cryptography are supported. For more info see *Asymmetric Keys*.

jwt-aud

Specifies the JWT audience claim. If this claim is present in the client provided JWT then you must set this to the same value as in the JWT, otherwise verifying the JWT will fail.

secret-is-base64

When this is set to `true`, the value derived from `jwt-secret` will be treated as a base64 encoded secret.

max-rows

A hard limit to the number of rows PostgREST will fetch from a view, table, or stored procedure. Limits payload size for accidental or malicious requests.

pre-request

A schema-qualified stored procedure name to call right after switching roles for a client request. This provides an opportunity to modify SQL variables or raise an exception to prevent the request from completing.

app.settings.*

Arbitrary settings that can be used to pass in secret keys directly as strings, or via OS environment variables. For instance: `app.settings.jwt_secret = "$(MYAPP_JWT_SECRET)"` will take `MYAPP_JWT_SECRET` from the environment and make it available to postgresql functions as `current_setting('app.settings.jwt_secret')`.

role-claim-key

A JSPath DSL that specifies the location of the `role` key in the JWT claims. This can be used to consume a JWT provided by a third party service like Auth0, Okta or Keycloak. Usage examples:

```
# {"postgrest":{"roles": ["other", "author"]}}
# the DSL accepts characters that are alphanumerical or one of "_$@" as keys
role-claim-key = ".postgrest.roles[1]"

# {"https://www.example.com/role": { "key": "author }}
# non-alphanumerical characters can go inside quotes(escaped in the config␣
↪value)
role-claim-key = ".\"https://www.example.com/role\".key"
```

8.1.9 Running the Server

PostgREST outputs basic request logging to stdout. When running it in an SSH session you must detach it from stdout or it will be terminated when the session closes. The easiest technique is redirecting the output to a log file or to the syslog:

```
ssh foo@example.com \
  'postgrest foo.conf </dev/null >/var/log/postgrest.log 2>&1 &'

# another option is to pipe the output into "logger -t postgrest"
```

(Avoid `nohup postgrest` because the HUP signal is used for manual *Schema Reloading*.)

8.1.10 Docker

You can get the official PostgREST Docker image with:

```
docker pull postgrest/postgrest
```

The image consults an internal `/etc/postgrest.conf` file. To customize this file you can either mount a replacement configuration file into the container, or use environment variables. The environment variables will be interpolated into the default config file.

These variables match the options shown in our *Configuration* section, except they are capitalized, have a `PGRST_` prefix, and use underscores. To get a list of the available environment variables, run this:

```
docker inspect -f "{{.Config.Env}}" postgrest/postgrest
```

There are two ways to run the PostgREST container: with an existing external database, or through docker-compose.

Containerized PostgREST with native PostgreSQL

The first way to run PostgREST in Docker is to connect it to an existing native database on the host.

```
# Run the server
docker run --rm --net=host -p 3000:3000 \
  -e PGRST_DB_URI="postgres://postgres@localhost/postgres" \
  -e PGRST_DB_ANON_ROLE="postgres" \
  postgrest/postgrest
```

The database connection string above is just an example. Adjust the role and password as necessary. You may need to edit PostgreSQL's `pg_hba.conf` to grant the user local login access.

Note: Docker on Mac does not support the `--net=host` flag. Instead you'll need to create an IP address alias to the host. Requests for the IP address from inside the container are unable to resolve and fall back to resolution by the host.

```
sudo ifconfig lo0 10.0.0.10 alias
```

You should then use 10.0.0.10 as the host in your database connection string. Also remember to include the IP address in the `listen_address` within postgresql.conf. For instance:

```
listen_addresses = 'localhost,10.0.0.10'
```

Containerized PostgREST *and* db with docker-compose

To avoid having to install the database at all, you can run both it and the server in containers and link them together with docker-compose. Use this configuration:

```
# docker-compose.yml

version: '3'
services:
  server:
    image: postgrest/postgrest
    ports:
      - "3000:3000"
    links:
      - db:db
    environment:
      PGRST_DB_URI: postgres://app_user:password@db:5432/app_db
      PGRST_DB_SCHEMA: public
      PGRST_DB_ANON_ROLE: app_user #In production this role should not be the same as␣
↪the one used for the connection
    depends_on:
      - db
  db:
    image: postgres
    ports:
      - "5432:5432"
    environment:
      POSTGRES_DB: app_db
      POSTGRES_USER: app_user
      POSTGRES_PASSWORD: password
  # Uncomment this if you want to persist the data.
  # volumes:
  #   - "./pgdata:/var/lib/postgresql/data"
```

Go into the directory where you saved this file and run `docker-compose up`. You will see the logs of both the database and PostgREST, and be able to access the latter on port 3000.

If you want to have a visual overview of your API in your browser you can add swagger-ui to your `docker-compose.yml`:

```
swagger:
  image: swaggerapi/swagger-ui
  ports:
    - "8080:8080"
  expose:
    - "8080"
  environment:
    API_URL: http://localhost:3000/
```

With this you can see the swagger-ui in your browser on port 8080.

8.1.11 Deploying to Heroku

Assuming your making modifications locally and then pushing to GitHub, it's easy to deploy to Heroku.

1. Create a new app on Heroku
2. In Settings add the following buildpack `https://github.com/PostgREST/postgrest-heroku`
3. Add the require Config Vars in Heroku (see https://github.com/PostgREST/postgrest/blob/master/app.json#L7-L57 for more details)
4. Modify your postgrest.conf file as required to match your Config Vars in Heroku
5. Create your `Procfile` and add `./env-to-config ./postgrest postgrest.conf`
6. Push your changes to GitHub
7. Set Heroku to automatically deploy from Master and then manually deploy the branch for the first build

8.1.12 Build from Source

Note: We discourage building and using PostgREST on **Alpine Linux** because of a reported GHC memory leak on that platform.

When a pre-built binary does not exist for your system you can build the project from source. You'll also need to do this if you want to help with development. Stack makes it easy. It will install any necessary Haskell dependencies on your system.

- Install Stack for your platform
- Install Library Dependencies

Operating System	Dependencies
Ubuntu/Debian	libpq-dev, libgmp-dev
CentOS/Fedora/Red Hat	postgresql-devel, zlib-devel, gmp-devel
BSD	postgresql95-client
OS X	libpq, gmp

- Build and install binary

```
git clone https://github.com/PostgREST/postgrest.git
cd postgrest
```

(continues on next page)

(continued from previous page)

```
# adjust local-bin-path to taste
stack build --install-ghc --copy-bins --local-bin-path /usr/local/bin
```

Note: If building fails and your system has less than 1GB of memory, try adding a swap file.

- Check that the server is installed: `postgrest --help`.

PostgREST Test Suite

Creating the Test Database

To properly run postgrest tests one needs to create a database. To do so, use the test creation script `create_test_database` in the `test/` folder.

The script expects the following parameters:

```
test/create_test_db connection_uri database_name [test_db_user] [test_db_user_
↪password]
```

Use the connection URI to specify the user, password, host, and port. Do not provide the database in the connection URI. The PostgreSQL role you are using to connect must be capable of creating new databases.

The `database_name` is the name of the database that `stack test` will connect to. If the database of the same name already exists on the server, the script will first drop it and then re-create it.

Optionally, specify the database user `stack test` will use. The user will be given necessary permissions to reset the database after every test run.

If the user is not specified, the script will generate the role name `postgrest_test_` suffixed by the chosen database name, and will generate a random password for it.

Optionally, if specifying an existing user to be used for the test connection, one can specify the password the user has.

The script will return the db uri to use in the tests–this uri corresponds to the `db-uri` parameter in the configuration file that one would use in production.

Generating the user and the password allows one to create the database and run the tests against any PostgreSQL server without any modifications to the server. (Such as allowing accounts without a password or setting up trust authentication, or requiring the server to be on the same localhost the tests are run from).

Running the Tests

To run the tests, one must supply the database uri in the environment variable `POSTGREST_TEST_CONNECTION`.

Typically, one would create the database and run the test in the same command line, using the *postgres* superuser:

```
POSTGREST_TEST_CONNECTION=$(test/create_test_db "postgres://postgres:pwd@database-host
↪" test_db) stack test
```

For repeated runs on the same database, one should export the connection variable:

```
export POSTGREST_TEST_CONNECTION=$(test/create_test_db "postgres://
↪postgres:pwd@database-host" test_db)
stack test
stack test
...
```

If the environment variable is empty or not specified, then the test runner will default to connection uri

```
postgres://postgrest_test@localhost/postgrest_test
```

This connection assumes the test server on the `localhost:code:` with the user *postgrest_test* without the password and the database of the same name.

Destroying the Database

The test database will remain after the test, together with four new roles created on the PostgreSQL server. To permanently erase the created database and the roles, run the script `test/delete_test_database`, using the same superuser role used for creating the database:

```
test/destroy_test_db connection_uri database_name
```

Testing with Docker

The ability to connect to non-local PostgreSQL simplifies the test setup. One elegant way of testing is to use a disposable PostgreSQL in docker.

For example, if local development is on a mac with Docker for Mac installed:

```
$ docker run --name db-scripting-test -e POSTGRES_PASSWORD=pwd -p 5434:5432 -d␣
↪postgres
$ POSTGREST_TEST_CONNECTION=$(test/create_test_db "postgres://
↪postgres:pwd@localhost:5434" test_db) stack test
```

Additionally, if one creates a docker container to run stack test (this is necessary on Mac OS Sierra with GHC below 8.0.1, where `stack test` fails), one can run PostgreSQL in a separate linked container, or use the locally installed PostgreSQL app.

Build the test container with `test/Dockerfile.test`:

```
$ docker build -t pgst-test - < test/Dockerfile.test
$ mkdir .stack-work-docker ~/.stack-linux
```

The first run of the test container will take a long time while the dependencies get cached. Creating the `~/.stack-linux` folder and mapping it as a volume into the container ensures that we can run the container in disposable mode and not worry about subsequent runs being slow. `.stack-work-docker` is also mapped into the container and must be specified when using stack from Linux, not to interfere with the `.stack-work` for local development. (On Sierra, `stack build` works, while `stack test` fails with GHC 8.0.1).

Linked containers:

```
$ docker run --name pg -e POSTGRES_PASSWORD=pwd  -d postgres
$ docker run --rm -it -v `pwd`:`pwd` -v ~/.stack-linux:/root/.stack --link pg:pg -w=
↪"`pwd`" -v `pwd`/.stack-work-docker:`pwd`/.stack-work pgst-test bash -c "POSTGREST_
↪TEST_CONNECTION=$(test/create_test_db "postgres://postgres:pwd@pg" test_db) stack␣
↪test"
```

Stack test in Docker for Mac, PostgreSQL app on mac:

```
$ host_ip=$(ifconfig en0 | grep 'inet ' | cut -f 2 -d' ')
$ export POSTGREST_TEST_CONNECTION=$(test/create_test_db "postgres://postgres@$HOST"␣
↪test_db)
$ docker run --rm -it -v `pwd`:`pwd` -v ~/.stack-linux:/root/.stack -v `pwd`/.stack-
↪work-docker:`pwd`/.stack-work -e "HOST=$host_ip" -e "POSTGREST_TEST_CONNECTION=
↪$POSTGREST_TEST_CONNECTION" -w="`pwd`" pgst-test bash -c "stack test"
$ test/destroy_test_db "postgres://postgres@localhost" test_db
```

8.1.13 Tables and Views

All views and tables in the exposed schema and accessible by the active database role for a request are available for querying. They are exposed in one-level deep routes. For instance the full contents of a table *people* is returned at

```
GET /people HTTP/1.1
```

There are no deeply/nested/routes. Each route provides OPTIONS, GET, POST, PATCH, and DELETE verbs depending entirely on database permissions.

Note: Why not provide nested routes? Many APIs allow nesting to retrieve related information, such as `/films/1/director`. We offer a more flexible mechanism (inspired by GraphQL) to embed related information. It can handle one-to-many and many-to-many relationships. This is covered in the section about *Resource Embedding*.

Horizontal Filtering (Rows)

You can filter result rows by adding conditions on columns, each condition a query string parameter. For instance, to return people aged under 13 years old:

```
GET /people?age=lt.13 HTTP/1.1
```

Multiple parameters can be logically conjoined by:

```
GET /people?age=gte.18&student=is.true HTTP/1.1
```

Multiple parameters can be logically disjoined by:

```
GET /people?or=(age.gte.14,age.lte.18) HTTP/1.1
```

Complex logic can also be applied:

```
GET /people?and=(grade.gte.90,student.is.true,or(age.gte.14,age.is.null)) HTTP/1.1
```

These operators are available:

Abbreviation	Meaning	PostgreSQL Equivalent
eq	equals	=
gt	greater than	>
gte	greater than or equal	>=
lt	less than	<
lte	less than or equal	<=
neq	not equal	<> or !=
like	LIKE operator (use * in place of %)	LIKE
ilike	ILIKE operator (use * in place of %)	ILIKE
in	one of a list of values e.g. `?a=in.(1,2,3)` – also supports commas in quoted strings like `?a=in.("hi,there","yes,you")`	IN
is	checking for exact equality (null,true,false)	IS
fts	*Full-Text Search* using to_tsquery	@@
plfts	*Full-Text Search* using plainto_tsquery	@@
phfts	*Full-Text Search* using phraseto_tsquery	@@
cs	contains e.g. `?tags=cs.{example, new}`	@>
cd	contained in e.g. `?values=cd.{1,2,3}`	<@
ov	overlap (have points in common), e.g. `?period=ov.[2017-01-01, 2017-06-30]`	&&
sl	strictly left of, e.g. `?range=sl.(1,10)`	<<
sr	strictly right of	>>
nxr	does not extend to the right of, e.g. `?range=nxr.(1,10)`	&<
nxl	does not extend to the left of	&>
adj	is adjacent to, e.g. `?range=adj.(1,10)`	-|-
not	negates another operator, see below	NOT

To negate any operator, prefix it with `not` like `?a=not.eq.2` or `?not.and=(a.gte.0,a.lte.100)` .

For more complicated filters you will have to create a new view in the database, or use a stored procedure. For instance, here's a view to show "today's stories" including possibly older pinned stories:

```
CREATE VIEW fresh_stories AS
SELECT *
  FROM stories
 WHERE pinned = true
    OR published > now() - interval '1 day'
ORDER BY pinned DESC, published DESC;
```

The view will provide a new endpoint:

```
GET /fresh_stories HTTP/1.1
```

Important: Views are invoked with the privileges of the view owner, much like stored procedures with the `SECURITY DEFINER` option. When created by a SUPERUSER role, all row-level security will be bypassed unless a different, non-SUPERUSER owner is specified.

```
-- Workaround:
-- non-SUPERUSER role to be used as the owner of the views
CREATE ROLE api_views_owner;
-- alter the view owner so RLS can work normally
ALTER VIEW sample_view OWNER TO api_views_owner;
```

Full-Text Search

The `fts` filter mentioned above has a number of options to support flexible textual queries, namely the choice of plain vs phrase search and the language used for stemming. Suppose that `tsearch` is a table with column `my_tsv`, of type tsvector. The following examples illustrate the possibilities.

```
GET /tsearch?my_tsv=fts(french).amusant HTTP/1.1
```

```
GET /tsearch?my_tsv=plfts.The%20Fat%20Cats HTTP/1.1
```

```
GET /tsearch?my_tsv=not.phfts(english).The%20Fat%20Cats HTTP/1.1
```

Using phrase search mode requires PostgreSQL of version at least 9.6 and will raise an error in earlier versions of the database.

Vertical Filtering (Columns)

When certain columns are wide (such as those holding binary data), it is more efficient for the server to withhold them in a response. The client can specify which columns are required using the `select` parameter.

```
GET /people?select=first_name,age HTTP/1.1

[
  {"first_name": "John", "age": 30},
  {"first_name": "Jane", "age": 20}
]
```

The default is `*`, meaning all columns. This value will become more important below in *Resource Embedding*.

You can rename the columns by prefixing them with an alias followed by the colon `:` operator.

```
GET /people?select=fullName:full_name,birthDate:birth_date HTTP/1.1

[
  {"fullName": "John Doe", "birthDate": "04/25/1988"},
  {"fullName": "Jane Doe", "birthDate": "01/12/1998"}
]
```

Casting the columns is possible by suffixing them with the double colon `::` plus the desired type.

```
GET /people?select=full_name,salary::text HTTP/1.1

[
  {"full_name": "John Doe", "salary": "90000.00"},
  {"full_name": "Jane Doe", "salary": "120000.00"}
]
```

JSON Columns

You can specify a path for a `json` or `jsonb` column using the arrow operators(`->` or `->>`) as per the PostgreSQL docs.

```
GET /people?select=id,json_data->>blood_type,json_data->phones HTTP/1.1

[
  { "id": 1, "blood_type": "A+", "phones": [{"country_code": "61", "number": "917-929-
↪5745"}] },
  { "id": 2, "blood_type": "O+", "phones": [{"country_code": "43", "number": "512-446-
↪4988"}, {"country_code": "43", "number": "213-891-5979"}] }
]
```

```
GET /people?select=id,json_data->phones->0->>number HTTP/1.1

[
  { "id": 1, "number": "917-929-5745"},
  { "id": 2, "number": "512-446-4988"}
]
```

Computed Columns

Filters may be applied to computed columns as well as actual table/view columns, even though the computed columns will not appear in the output. For example, to search first and last names at once we can create a computed column that will not appear in the output but can be used in a filter:

```
CREATE TABLE people (
  fname text,
  lname text
);

CREATE FUNCTION full_name(people) RETURNS text AS $$
  SELECT $1.fname || ' ' || $1.lname;
$$ LANGUAGE SQL;

-- (optional) add an index to speed up anticipated query
CREATE INDEX people_full_name_idx ON people
  USING GIN (to_tsvector('english', full_name(people)));
```

A full-text search on the computed column:

```
GET /people?full_name=fts.Beckett HTTP/1.1
```

As mentioned, computed columns do not appear in the output by default. However you can include them by listing them in the vertical filtering `select` param:

```
GET /people?select=*,full_name HTTP/1.1
```

Important: Computed columns must be created under the *exposed schema* to be used in this way.

Unicode support

PostgREST supports unicode in schemas, tables, columns and values. To access a table with unicode name, use percent encoding.

To request this:

```
GET / HTTP/1.1
```

Do this:

```
GET /%D9%85%D9%88%D8%A7%D8%B1%D8%AF HTTP/1.1
```

Table / Columns with spaces

You can request table/columns with spaces in them by percent encoding the spaces with `%20`:

```
GET /Order%20Items?Unit%20Price=lt.200 HTTP/1.1
```

Reserved characters

If filters include PostgREST reserved characters(`,`, `.`, `:`, `()`) you'll have to surround them in percent encoded double quotes `%22` for correct processing.

Here `Hebdon,John` and `Williams,Mary` are values.

```
GET /employees?name=in.(%22Hebdon,John%22,%22Williams,Mary%22) HTTP/1.1
```

Here `information.cpe` is a column name.

```
GET /vulnerabilities?%22information.cpe%22=like.*MS* HTTP/1.1
```

Ordering

The reserved word `order` reorders the response rows. It uses a comma-separated list of columns and directions:

```
GET /people?order=age.desc,height.asc HTTP/1.1
```

If no direction is specified it defaults to ascending order:

```
GET /people?order=age HTTP/1.1
```

If you care where nulls are sorted, add nullsfirst or nullslast:

```
GET /people?order=age.nullsfirst HTTP/1.1
```

```
GET /people?order=age.desc.nullslast HTTP/1.1
```

You can also use *Computed Columns* to order the results, even though the computed columns will not appear in the output.

Limits and Pagination

PostgREST uses HTTP range headers to describe the size of results. Every response contains the current range and, if requested, the total number of results:

```
HTTP/1.1 200 OK
Range-Unit: items
Content-Range: 0-14/*
```

Here items zero through fourteen are returned. This information is available in every response and can help you render pagination controls on the client. This is an RFC7233-compliant solution that keeps the response JSON cleaner.

There are two ways to apply a limit and offset rows: through request headers or query params. When using headers you specify the range of rows desired. This request gets the first twenty people.

```
GET /people HTTP/1.1
Range-Unit: items
Range: 0-19
```

Note that the server may respond with fewer if unable to meet your request:

```
HTTP/1.1 200 OK
Range-Unit: items
Content-Range: 0-17/*
```

You may also request open-ended ranges for an offset with no limit, e.g. `Range:  10-`.

The other way to request a limit or offset is with query parameters. For example

```
GET /people?limit=15&offset=30 HTTP/1.1
```

This method is also useful for embedded resources, which we will cover in another section. The server always responds with range headers even if you use query parameters to limit the query.

In order to obtain the total size of the table or view (such as when rendering the last page link in a pagination control), specify your preference in a request header:

```
GET /bigtable HTTP/1.1
Range-Unit: items
Range: 0-24
Prefer: count=exact
```

Note that the larger the table the slower this query runs in the database. The server will respond with the selected range and total

```
HTTP/1.1 206 Partial Content
Range-Unit: items
Content-Range: 0-24/3573458
```

Response Format

PostgREST uses proper HTTP content negotiation (RFC7231) to deliver the desired representation of a resource. That is to say the same API endpoint can respond in different formats like JSON or CSV depending on the client request.

Use the Accept request header to specify the acceptable format (or formats) for the response:

```
GET /people HTTP/1.1
Accept: application/json
```

The current possibilities are

- */*

- text/csv
- application/json
- application/openapi+json
- application/octet-stream

The server will default to JSON for API endpoints and OpenAPI on the root.

Singular or Plural

By default PostgREST returns all JSON results in an array, even when there is only one item. For example, requesting `/items?id=eq.1` returns

```
[
  { "id": 1 }
]
```

This can be inconvenient for client code. To return the first result as an object unenclosed by an array, specify `vnd.pgrst.object` as part of the `Accept` header

```
GET /items?id=eq.1 HTTP/1.1
Accept: application/vnd.pgrst.object+json
```

This returns

```
{ "id": 1 }
```

When a singular response is requested but no entries are found, the server responds with an error message and 406 Not Acceptable status code rather than the usual empty array and 200 status:

```
{
  "message": "JSON object requested, multiple (or no) rows returned",
  "details": "Results contain 0 rows, application/vnd.pgrst.object+json requires 1 row
↪"
}
```

Note: Many APIs distinguish plural and singular resources using a special nested URL convention e.g. */stories* vs */stories/1*. Why do we use */stories?id=eq.1*? The answer is because a singular resource is (for us) a row determined by a primary key, and primary keys can be compound (meaning defined across more than one column). The more familiar nested urls consider only a degenerate case of simple and overwhelmingly numeric primary keys. These so-called artificial keys are often introduced automatically by Object Relational Mapping libraries.

Admittedly PostgREST could detect when there is an equality condition holding on all columns constituting the primary key and automatically convert to singular. However this could lead to a surprising change of format that breaks unwary client code just by filtering on an extra column. Instead we allow manually specifying singular vs plural to decouple that choice from the URL format.

Binary output

If you want to return raw binary data from a `bytea` column, you must specify `application/octet-stream` as part of the `Accept` header and select a single column `?select=bin_data`.

```
GET /items?select=bin_data&id=eq.1 HTTP/1.1
Accept: application/octet-stream
```

You can also request binary output when calling *Stored Procedures* and since they can return a scalar value you are not forced to use `select` for this case.

```
CREATE FUNCTION closest_point(..) RETURNS bytea ..
```

```
POST /rpc/closest_point HTTP/1.1
Accept: application/octet-stream
```

If the stored procedure returns non-scalar values, you need to do a `select` in the same way as for GET binary output.

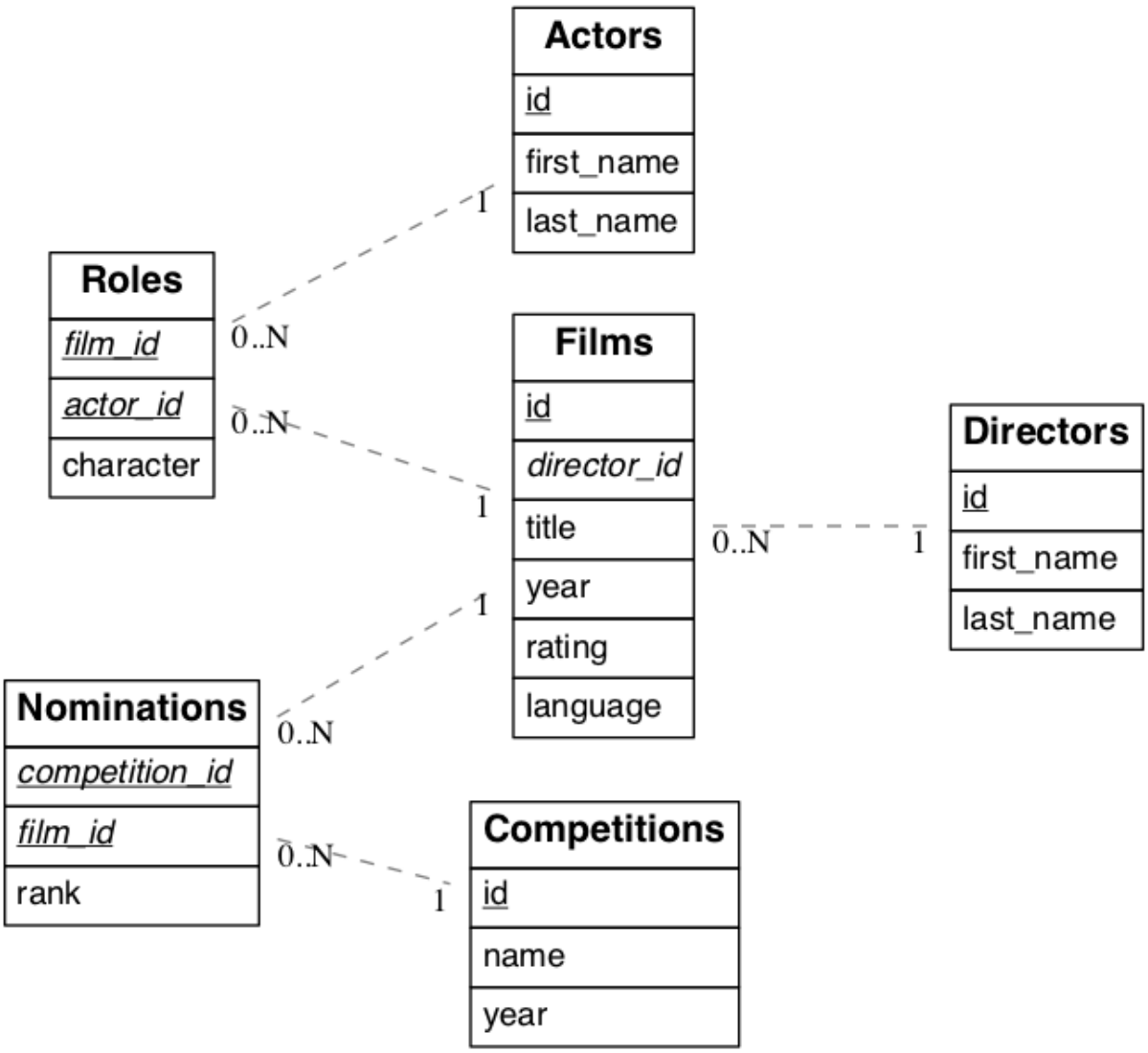
```
CREATE FUNCTION overlapping_regions(..) RETURNS SETOF TABLE(geom_twkb bytea, ..) ..
```

```
POST /rpc/overlapping_regions?select=geom_twkb HTTP/1.1
Accept: application/octet-stream
```

Note: If more than one row would be returned the binary results will be concatenated with no delimiter.

8.1.14 Resource Embedding

In addition to providing RESTful routes for each table and view, PostgREST allows related resources to be included together in a single API call. This reduces the need for multiple API requests. The server uses foreign keys to determine which tables and views can be returned together. For example, consider a database of films and their awards:



As seen above in *Vertical Filtering (Columns)* we can request the titles of all films like this:

```
GET /films?select=title HTTP/1.1
```

This might return something like

```
[
  { "title": "Workers Leaving The Lumière Factory In Lyon" },
  { "title": "The Dickson Experimental Sound Film" },
  { "title": "The Haunted Castle" }
]
```

However because a foreign key constraint exists between Films and Directors, we can request this information be included:

```
GET /films?select=title,directors(id,last_name) HTTP/1.1
```

Which would return

```
[
  { "title": "Workers Leaving The Lumière Factory In Lyon",
    "directors": {
      "id": 2,
      "last_name": "Lumière"
    }
  },
  { "title": "The Dickson Experimental Sound Film",
    "directors": {
      "id": 1,
      "last_name": "Dickson"
    }
  },
  { "title": "The Haunted Castle",
    "directors": {
      "id": 3,
      "last_name": "Méliès"
    }
  }
]
```

In this example, since the relationship is a forward relationship, there is only one director associated with a film. As the table name is plural it might be preferable for it to be singular instead. An table name alias can accomplish this:

```
GET /films?select=title,director:directors(id,last_name) HTTP/1.1
```

PostgREST can also detect relations going through join tables. Thus you can request the Actors for Films (which in this case finds the information through Roles). You can also reverse the direction of inclusion, asking for all Directors with each including the list of their Films:

```
GET /directors?select=films(title,year) HTTP/1.1
```

Important: Whenever foreign key relations change in the database schema you must refresh PostgREST's schema cache to allow resource embedding to work properly. See the section *Schema Reloading*.

Embedded Filters

Embedded resources can be shaped similarly to their top-level counterparts. To do so, prefix the query parameters with the name of the embedded resource. For instance, to order the actors in each film:

```
GET /films?select=*,actors(*)&actors.order=last_name,first_name HTTP/1.1
```

This sorts the list of actors in each film but does *not* change the order of the films themselves. To filter the roles returned with each film:

```
GET /films?select=*,roles(*)&roles.character=in.(Chico,Harpo,Groucho) HTTP/1.1
```

Once again, this restricts the roles included to certain characters but does not filter the films in any way. Films without any of those characters would be included along with empty character lists.

An `or` filter can be used for a similar operation:

```
GET /films?select=*,roles(*)&roles.or=(character.eq.Gummo,character.eq.Zeppo) HTTP/1.1
```

Limit and offset operations are possible:

```
GET /films?select=*,actors(*)&actors.limit=10&actors.offset=2 HTTP/1.1
```

Embedded resources can be aliased and filters can be applied on these aliases:

```
GET /films?select=*,90_comps:competitions(name),91_comps:competitions(name)&90_comps.
↪year=eq.1990&91_comps.year=eq.1991 HTTP/1.1
```

8.1.15 Custom Queries

The PostgREST URL grammar limits the kinds of queries clients can perform. It prevents arbitrary, potentially poorly constructed and slow client queries. It's good for quality of service, but means database administrators must create custom views and stored procedures to provide richer endpoints. The most common causes for custom endpoints are

- Table unions
- More complicated joins than those provided by *Resource Embedding*
- Geo-spatial queries that require an argument, like "points near (lat,lon)"
- More sophisticated full-text search than a simple use of the `fts` filter

8.1.16 Stored Procedures

Every stored procedure in the API-exposed database schema is accessible under the `/rpc` prefix. The API endpoint supports POST (and in some cases GET) to execute the function.

```
POST /rpc/function_name HTTP/1.1
```

Such functions can perform any operations allowed by PostgreSQL (read data, modify data, and even DDL operations).

To supply arguments in an API call, include a JSON object in the request payload and each key/value of the object will become an argument.

For instance, assume we have created this function in the database.

```
CREATE FUNCTION add_them(a integer, b integer)
RETURNS integer AS $$
 SELECT a + b;
$$ LANGUAGE SQL IMMUTABLE STRICT;
```

Important: Whenever you create or change a function you must refresh PostgREST's schema. See the section *Schema Reloading*.

The client can call it by posting an object like

```
POST /rpc/add_them HTTP/1.1

{ "a": 1, "b": 2 }

3
```

You can also call a function that takes a single parameter of type json by sending the header `Prefer: params=single-object` with your request. That way the JSON request body will be used as the single argument.

```
CREATE FUNCTION mult_them(param json) RETURNS int AS $$
  SELECT (param->>'x')::int * (param->>'y')::int
$$ LANGUAGE SQL;
```

```
POST /rpc/mult_them HTTP/1.1
Prefer: params=single-object

{ "x": 4, "y": 2 }

8
```

Procedures must be declared with named parameters, procedures declared like:

```
CREATE FUNCTION non_named_args(integer, text, integer) ...
```

Can not be called with PostgREST, since we use named notation internally.

Note that PostgreSQL converts identifier names to lowercase unless you quote them like:

```
CREATE FUNCTION "someFunc"("someParam" text) ...
```

PostgreSQL has four procedural languages that are part of the core distribution: PL/pgSQL, PL/Tcl, PL/Perl, and PL/Python. There are many other procedural languages distributed as additional extensions. Also, plain SQL can be used to write functions (as shown in the example above).

Note: For versions prior to PostgreSQL 10, to pass a PostgreSQL native array you need to quote it as a string:

```
POST /rpc/native_array_func HTTP/1.1

{ "arg": "{1,2,3}" }
```

In these versions we recommend using function arguments of type json to accept arrays from the client:

```
POST /rpc/json_array_func HTTP/1.1

{ "arg": [1,2,3] }
```

Starting from PostgreSQL 10, a json array from the client gets mapped normally to a PostgreSQL native array.

Note: Why the */rpc* prefix? One reason is to avoid name collisions between views and procedures. It also helps emphasize to API consumers that these functions are not normal restful things. The functions can have arbitrary and surprising behavior, not the standard "post creates a resource" thing that users expect from the other routes.

Immutable and stable functions

Procedures in PostgreSQL marked with `stable` or `immutable` volatility can only read, not modify, the database and PostgREST executes them in a read-only transaction compatible for read-replicas. Stable and immutable functions can be called with the HTTP GET verb if desired.

Note: The volatility marker is a promise about the behavior of the function. PostgreSQL will let you mark a function that modifies the database as `immutable/stable` without failure. However the function will fail when called

through PostgREST since it executes it in a read-only transaction.

Because `add_them` was declared IMMUTABLE, we can alternately call the function with a GET request:

```
GET /rpc/add_them?a=1&b=2 HTTP/1.1
```

The function parameter names match the JSON object keys in the POST case, for the GET case they match the query parameters `?a=1&b=2`.

Scalar functions

PostgREST will detect if the function is scalar or table-valued and will shape the response format accordingly:

```
GET /rpc/add_them?a=1&b=2 HTTP/1.1

3
```

```
GET /rpc/best_films_2017 HTTP/1.1

[
  { "title": "Okja", "rating": 7.4},
  { "title": "Call me by your name", "rating": 8},
  { "title": "Blade Runner 2049", "rating": 8.1}
]
```

Function filters

A function that returns a table type response can be shaped using the same filters as the ones used for tables and views:

```
CREATE FUNCTION best_films_2017() RETURNS SETOF films ..
```

```
GET /rpc/best_films_2017?select=title,director:directors(*) HTTP/1.1
```

```
GET /rpc/best_films_2017?rating=gt.8&order=title.desc HTTP/1.1
```

Function privileges

By default, a function is executed with the privileges of the user who calls it. This means that the user has to have all permissions to do the operations the procedure performs.

Another option is to define the function with the `SECURITY DEFINER` option. Then only one permission check will take place, the permission to call the function, and the operations in the function will have the authority of the user who owns the function itself. See PostgreSQL documentation for more details.

Warning: Unlike tables/views, functions privileges work as a blacklist, so they're executable for all the roles by default. You can workaround this by revoking the PUBLIC privileges of the function and then granting privileges to specific roles:

```
REVOKE ALL PRIVILEGES ON FUNCTION private_func() FROM PUBLIC;
GRANT EXECUTE ON FUNCTION private_func() TO a_role;
```

Also to avoid doing `REVOKE` on every function you can enable this behavior by default with:

```
ALTER DEFAULT PRIVILEGES REVOKE EXECUTE ON FUNCTIONS FROM PUBLIC;
```

See PostgreSQL alter default privileges for more details.

Overloaded functions

You can call overloaded functions with different number of arguments.

```
CREATE FUNCTION rental_duration(customer_id integer) ..

CREATE FUNCTION rental_duration(customer_id integer, from_date date) ..
```

```
GET /rpc/rental_duration?customer_id=232 HTTP/1.1
```

```
GET /rpc/rental_duration?customer_id=232&from_date=2018-07-01 HTTP/1.1
```

Accessing Request Headers, Cookies and JWT claims

Stored procedures can access request headers, cookies and jwt claims by reading GUC variables set by PostgREST per request. They are named `request.header.XYZ`, `request.cookie.XYZ` and `request.jwt.claim.XYZ`.

```
-- To read the value of the Origin request header:
SELECT current_setting('request.header.origin', true);
-- To read the value of sessionId in a cookie:
SELECT current_setting('request.cookie.sessionId', true);
-- To read the value of the email claim in a jwt:
SELECT current_setting('request.jwt.claim.email', true);
```

Note: `request.jwt.claim.role` defaults to the value of *db-anon-role*.

Errors and HTTP Status Codes

Stored procedures can return non-200 HTTP status codes by raising SQL exceptions. For instance, here's a saucy function that always responds with an error:

```
CREATE OR REPLACE FUNCTION just_fail() RETURNS void
  LANGUAGE plpgsql
  AS $$
BEGIN
  RAISE EXCEPTION 'I refuse!'
    USING DETAIL = 'Pretty simple',
          HINT = 'There is nothing you can do.';
END
$$;
```

Calling the function returns HTTP 400 with the body

```
{
  "message":"I refuse!",
  "details":"Pretty simple",
  "hint":"There is nothing you can do.",
  "code":"P0001"
}
```

One way to customize the HTTP status code is by raising particular exceptions according to the PostgREST *error to status code mapping*. For example, `RAISE insufficient_privilege` will respond with HTTP 401/403 as appropriate.

For even greater control of the HTTP status code, raise an exception of the `PTxyz` type. For instance to respond with HTTP 402, raise 'PT402':

```
RAISE sqlstate 'PT402' using
  message = 'Payment Required',
  detail = 'Quota exceeded',
  hint = 'Upgrade your plan';
```

Returns:

```
HTTP/1.1 402 Payment Required
Content-Type: application/json; charset=utf-8

{"hint":"Upgrade your plan","details":"Quota exceeded"}
```

Setting Response Headers

PostgREST reads the `response.headers` SQL variable to add extra headers to the HTTP response. Stored procedures can modify this variable. For instance, this statement would add caching headers to the response:

```
-- tell client to cache response for two days

SET LOCAL "response.headers" =
  '[{"Cache-Control": "public"}, {"Cache-Control": "max-age=259200"}]';
```

Notice that the variable should be set to an *array* of single-key objects rather than a single multiple-key object. This is because headers such as `Cache-Control` or `Set-Cookie` need to be repeated when setting multiple values and an object would not allow the repeated key.

8.1.17 Insertions / Updates

All tables and auto-updatable views can be modified through the API, subject to permissions of the requester's database role.

To create a row in a database table post a JSON object whose keys are the names of the columns you would like to create. Missing properties will be set to default values when applicable.

```
POST /table_name HTTP/1.1

{ "col1": "value1", "col2": "value2" }
```

The response will include a `Location` header describing where to find the new object. If the table is write-only then constructing the Location header will cause a permissions error. To successfully insert an item to a write-only table you will need to suppress the Location response header by including the request header `Prefer: return=minimal`.

On the other end of the spectrum you can get the full created object back in the response to your request by including the header `Prefer: return=representation`. That way you won't have to make another HTTP call to discover properties that may have been filled in on the server side. You can also apply the standard *Vertical Filtering (Columns)* to these results.

URL encoded payloads can be posted with `Content-Type: application/x-www-form-urlencoded`.

```
POST /people HTTP/1.1
Content-Type: application/x-www-form-urlencoded

name=John+Doe&age=50&weight=80
```

Note: When inserting a row you must post a JSON object, not quoted JSON.

```
Yes
{ "a": 1, "b": 2 }

No
"{ \"a\": 1, \"b\": 2 }"
```

Some javascript libraries will post the data incorrectly if you're not careful. For best results try one of the *Client-Side Libraries* built for PostgREST.

To update a row or rows in a table, use the PATCH verb. Use *Horizontal Filtering (Rows)* to specify which record(s) to update. Here is an example query setting the `category` column to child for all people below a certain age.

```
PATCH /people?age=lt.13 HTTP/1.1

{ "category": "child" }
```

Updates also support `Prefer: return=representation` plus *Vertical Filtering (Columns)*.

Warning: Beware of accidentally updating every row in a table. To learn to prevent that see *Block Full-Table Operations*.

Bulk Insert

Bulk insert works exactly like single row insert except that you provide either a JSON array of objects having uniform keys, or lines in CSV format. This not only minimizes the HTTP requests required but uses a single INSERT statement on the back-end for efficiency. Note that using CSV requires less parsing on the server and is much faster.

To bulk insert CSV simply post to a table route with `Content-Type: text/csv` and include the names of the columns as the first row. For instance

```
POST /people HTTP/1.1
Content-Type: text/csv

name,age,height
J Doe,62,70
Jonas,10,55
```

An empty field (`,,`) is coerced to an empty string and the reserved word `NULL` is mapped to the SQL null value. Note that there should be no spaces between the column names and commas.

To bulk insert JSON post an array of objects having all-matching keys

```
POST /people HTTP/1.1
Content-Type: application/json

[
  { "name": "J Doe", "age": 62, "height": 70 },
  { "name": "Janus", "age": 10, "height": 55 }
]
```

Upsert

You can make an UPSERT with `POST` and the `Prefer: resolution=merge-duplicates` header:

```
POST /employees HTTP/1.1
Prefer: resolution=merge-duplicates

[
  { "id": 1, "name": "Old employee 1", "salary": 30000 },
  { "id": 2, "name": "Old employee 2", "salary": 42000 },
  { "id": 3, "name": "New employee 3", "salary": 50000 }
]
```

UPSERT operates based on the primary key columns, you must specify all of them. You can also choose to ignore the duplicates with `Prefer: resolution=ignore-duplicates`. UPSERT works best when the primary key is natural, but it's also possible to use it if the primary key is surrogate (example: "id serial primary key"). For more details read this issue.

Important: After creating a table or changing its primary key, you must refresh PostgREST schema cache for UPSERT to work properly. To learn how to refresh the cache see *Schema Reloading*.

A single row UPSERT can be done by using `PUT` and filtering the primary key columns with `eq`:

```
PUT /employees?id=eq.4 HTTP/1.1

{ "id": 4, "name": "Sara B.", "salary": 60000 }
```

All the columns must be specified in the request body, including the primary key columns.

Note: This feature is only available starting from PostgreSQL 9.5 since it uses the ON CONFLICT clause.

8.1.18 Deletions

To delete rows in a table, use the DELETE verb plus *Horizontal Filtering (Rows)*. For instance deleting inactive users:

```
DELETE /user?active=is.false HTTP/1.1
```

Warning: Beware of accidentally deleting all rows in a table. To learn to prevent that see *Block Full-Table Operations*.

8.1.19 OpenAPI Support

Every API hosted by PostgREST automatically serves a full OpenAPI description on the root path. This provides a list of all endpoints(tables, foreign tables, views, functions), along with supported HTTP verbs and example payloads. For extra customization, the OpenAPI output contains a "description" field for every SQL comment on any database object. For instance,

```
COMMENT ON SCHEMA mammals IS
  'A warm-blooded vertebrate animal of a class that is distinguished by the secretion␣
↪of milk by females for the nourishment of the young';

COMMENT ON TABLE monotremes IS
  'Freakish mammals lay the best eggs for breakfast';

COMMENT ON COLUMN monotremes.has_venomous_claw IS
  'Sometimes breakfast is not worth it';
```

These unsavory comments will appear in the generated JSON as the fields, `info.description`, `definitions.monotremes.description` and `definitions.monotremes.properties.has_venomous_claw.description`.

Also if you wish to generate a `summary` field you can do it by having a multiple line comment, the `summary` will be the first line and the `description` the lines that follow it:

```
COMMENT ON TABLE entities IS
  $$Entities summary

  Entities description that
  spans
  multiple lines$$;
```

You can use a tool like Swagger UI to create beautiful documentation from the description and to host an interactive web-based dashboard. The dashboard allows developers to make requests against a live PostgREST server, and provides guidance with request headers and example request bodies.

Important: The OpenAPI information can go out of date as the schema changes under a running server. To learn how to refresh the cache see *Schema Reloading*.

8.1.20 HTTP Status Codes

PostgREST translates PostgreSQL error codes into HTTP status as follows:

PostgreSQL error code(s)	HTTP status	Error description
08*	503	pg connection err
09*	500	triggered action exception
0L*	403	invalid grantor
0P*	403	invalid role specification
23503	409	foreign key violation
23505	409	uniqueness violation
25*	500	invalid transaction state
28*	403	invalid auth specification
2D*	500	invalid transaction termination
38*	500	external routine exception
39*	500	external routine invocation
3B*	500	savepoint exception
40*	500	transaction rollback
53*	503	insufficient resources
54*	413	too complex
55*	500	obj not in prerequisite state
57*	500	operator intervention
58*	500	system error
F0*	500	conf file error
HV*	500	foreign data wrapper error
P0001	400	default code for “raise”
P0*	500	PL/pgSQL error
XX*	500	internal error
42883	404	undefined function
42P01	404	undefined table
42501	if authenticated 403, else 401	insufficient privileges
other	500	

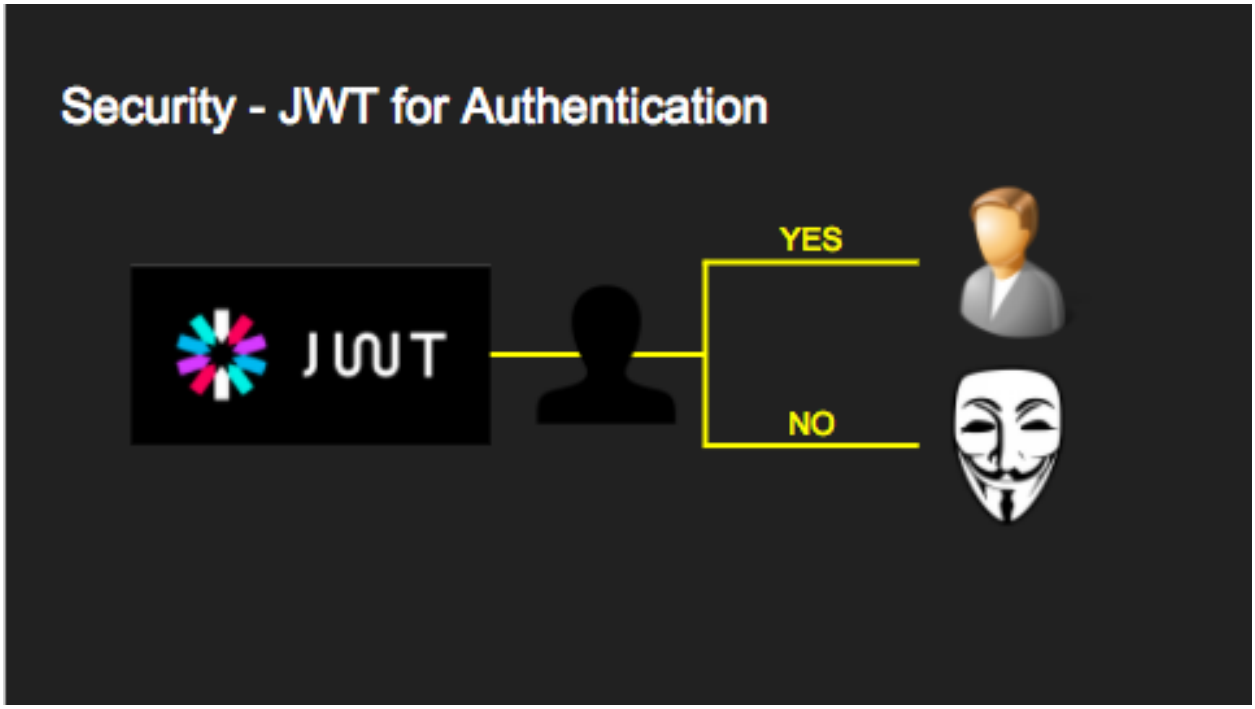
8.1.21 Overview of Role System

PostgREST is designed to keep the database at the center of API security. All authorization happens through database roles and permissions. It is PostgREST's job to **authenticate** requests – i.e. verify that a client is who they say they are – and then let the database **authorize** client actions.

Authentication Sequence

There are three types of roles used by PostgREST, the **authenticator**, **anonymous** and **user** roles. The database administrator creates these roles and configures PostgREST to use them.

The authenticator should be created `NOINHERIT` and configured in the database to have very limited access. It is a chameleon whose job is to "become" other users to service authenticated HTTP requests. The picture below shows how the server handles authentication. If auth succeeds, it switches into the user role specified by the request, otherwise it switches into the anonymous role.



Here are the technical details. We use JSON Web Tokens to authenticate API requests. As you'll recall a JWT contains a list of cryptographically signed claims. All claims are allowed but PostgREST cares specifically about a claim called role.

```
{
  "role": "user123"
}
```

When a request contains a valid JWT with a role claim PostgREST will switch to the database role with that name for the duration of the HTTP request.

```
SET LOCAL ROLE user123;
```

Note that the database administrator must allow the authenticator role to switch into this user by previously executing

```
GRANT user123 TO authenticator;
```

If the client included no JWT (or one without a role claim) then PostgREST switches into the anonymous role whose actual database-specific name, like that of with the authenticator role, is specified in the PostgREST server configuration file. The database administrator must set anonymous role permissions correctly to prevent anonymous users from seeing or changing things they shouldn't.

Users and Groups

PostgreSQL manages database access permissions using the concept of roles. A role can be thought of as either a database user, or a group of database users, depending on how the role is set up.

Roles for Each Web User

PostgREST can accommodate either viewpoint. If you treat a role as a single user then the the JWT-based role switching described above does most of what you need. When an authenticated user makes a request PostgREST will switch into the role for that user, which in addition to restricting queries, is available to SQL through the `current_user` variable.

You can use row-level security to flexibly restrict visibility and access for the current user. Here is an example from Tomas Vondra, a chat table storing messages sent between users. Users can insert rows into it to send messages to other users, and query it to see messages sent to them by other users.

```
CREATE TABLE chat (
  message_uuid    UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  message_time    TIMESTAMP NOT NULL DEFAULT now(),
  message_from    NAME      NOT NULL DEFAULT current_user,
  message_to      NAME      NOT NULL,
  message_subject VARCHAR(64) NOT NULL,
  message_body    TEXT
);
```

We want to enforce a policy that ensures a user can see only those messages sent by him or intended for him. Also we want to prevent a user from forging the message_from column with another person's name.

PostgreSQL (9.5 and later) allows us to set this policy with row-level security:

```
CREATE POLICY chat_policy ON chat
  USING ((message_to = current_user) OR (message_from = current_user))
  WITH CHECK (message_from = current_user)
```

Anyone accessing the generated API endpoint for the chat table will see exactly the rows they should, without our needing custom imperative server-side coding.

Web Users Sharing Role

Alternately database roles can represent groups instead of (or in addition to) individual users. You may choose that all signed-in users for a web app share the role webuser. You can distinguish individual users by including extra claims in the JWT such as email.

```
{
  "role": "webuser",
  "email": "john@doe.com"
}
```

SQL code can access claims through GUC variables set by PostgREST per request. For instance to get the email claim, call this function:

```
current_setting('request.jwt.claim.email', true)
```

This allows JWT generation services to include extra information and your database code to react to it. For instance the RLS example could be modified to use this current_setting rather than current_user. The second 'true' argument tells current_setting to return NULL if the setting is missing from the current configuration.

Hybrid User-Group Roles

There is no performance penalty for having many database roles, although roles are namespaced per-cluster rather than per-database so may be prone to collision within the database. You are free to assign a new role for every user in a web application if desired. You can mix the group and individual role policies. For instance we could still have a webuser role and individual users which inherit from it:

```
CREATE ROLE webuser NOLOGIN;
-- grant this role access to certain tables etc

CREATE ROLE user000 NOLOGIN;
GRANT webuser TO user000;
-- now user000 can do whatever webuser can

GRANT user000 TO authenticator;
-- allow authenticator to switch into user000 role
-- (the role itself has nologin)
```

Custom Validation

PostgREST honors the `exp` claim for token expiration, rejecting expired tokens. However it does not enforce any extra constraints. An example of an extra constraint would be to immediately revoke access for a certain user. The configuration file parameter `pre-request` specifies a stored procedure to call immediately after the authenticator switches into a new role and before the main query itself runs.

Here's an example. In the config file specify a stored procedure:

```
pre-request = "public.check_user"
```

In the function you can run arbitrary code to check the request and raise an exception to block it if desired.

```
CREATE OR REPLACE FUNCTION check_user() RETURNS void AS $$
BEGIN
  IF current_user = 'evil_user' THEN
```

(continues on next page)

(continued from previous page)

```
    RAISE EXCEPTION 'No, you are evil'
      USING HINT = 'Stop being so evil and maybe you can log in';
  END IF;
END
$$ LANGUAGE plpgsql;
```

8.1.22 Client Auth

To make an authenticated request the client must include an `Authorization` HTTP header with the value `Bearer <jwt>`. For instance:

```
GET /foo HTTP/1.1
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJyb2xlIjoiamRvZSIsImV4cCI6MTQ3NTUxNjI1MH0.GYDZV3yM0gqvuEtJmfpplLBXSGYnke_
↪Pvnl0tbKAjB4
```

JWT Generation

You can create a valid JWT either from inside your database or via an external service. Each token is cryptographically signed with a secret key. In the case of symmetric cryptography the signer and verifier share the same secret passphrase. In asymmetric cryptography the signer uses the private key and the verifier the public key. PostgREST supports both symmetric and asymmetric cryptography.

JWT from SQL

You can create JWT tokens in SQL using the pgjwt extension. It's simple and requires only pgcrypto. If you're on an environment like Amazon RDS which doesn't support installing new extensions, you can still manually run the SQL inside pgjwt (you'll need to replace `@extschema@` with another schema or just delete it) which creates the functions you will need.

Next write a stored procedure that returns the token. The one below returns a token with a hard-coded role, which expires five minutes after it was issued. Note this function has a hard-coded secret as well.

```
CREATE TYPE jwt_token AS (
  token text
);

CREATE FUNCTION jwt_test() RETURNS public.jwt_token AS $$
  SELECT public.sign(
    row_to_json(r), 'reallyreallyreallyreallyverysafe'
  ) AS token
  FROM (
    SELECT
      'my_role'::text as role,
      extract(epoch from now())::integer + 300 AS exp
  ) r;
$$ LANGUAGE sql;
```

PostgREST exposes this function to clients via a POST request to */rpc/jwt_test*.

Note: To avoid hard-coding the secret in stored procedures, save it as a property of the database.

```
-- run this once
ALTER DATABASE mydb SET "app.jwt_secret" TO 'reallyreallyreallyreallyverysafe';

-- then all functions can refer to app.jwt_secret
SELECT sign(
  row_to_json(r), current_setting('app.jwt_secret')
) AS token
FROM ...
```

JWT from Auth0

An external service like Auth0 can do the hard work transforming OAuth from Github, Twitter, Google etc into a JWT suitable for PostgREST. Auth0 can also handle email signup and password reset flows.

To use Auth0, create an application for your app and an API for your PostgREST server. Auth0 supports both HS256 and RS256 scheme for the issued tokens for APIs. For simplicity, you may first try HS256 scheme while creating your API on Auth0. Your application should use your PostgREST API's API identifier by setting it with the audience parameter during the authorization request. This will ensure that Auth0 will issue an access token for your PostgREST API. For PostgREST to verify the access token, you will need to set `jwt-secret` on PostgREST config file with your API's signing secret.

Our code requires a database role in the JWT. To add it you need to save the database role in Auth0 app metadata. Then, you will need to write a rule that will extract the role from the user's app_metadata and set it as a custom claim in the access token. Note that, you may use Auth0's core authorization feature for more complex use cases. Metadata solution is mentioned here for simplicity.

```
function (user, context, callback) {

  // Follow the documentations at http://postgrest.org/en/v7.0.0/configuration.html
↪#role-claim-key
  // to set a custom role claim on PostgREST and use it as custom claim attribute in␣
↪this rule
  const myRoleClaim = 'https://myapp.com/role';

  user.app_metadata = user.app_metadata || {};
  context.accessToken[myRoleClaim] = user.app_metadata.role;
  callback(null, user, context);
}
```

Asymmetric Keys

As described in the *Configuration* section, PostgREST accepts a `jwt-secret` config file parameter. If it is set to a simple string value like "reallyreallyreallyreallyverysafe" then PostgREST interprets it as an HMAC-SHA256 passphrase. However you can also specify a literal JSON Web Key (JWK) or set. For example, you can use an RSA-256 public key encoded as a JWK:

```
{
  "alg":"RS256",
  "e":"AQAB",
  "key_ops":["verify"],
  "kty":"RSA",
```

(continues on next page)

(continued from previous page)

```
  "n":"9zKNYTaYGfGm1tBMpRT6FxOYrM720GhXdettc02uyakYSEHU2IJz90G_MLlEl4-WWWYoS_
↪QKFupw3s7aPYlaAjamG22rAnvWu-rRkP5sSSkKvud_IgKL4iE6Y2WJx2Bkl1XUFkdZ8wlEUR6O1ft3TS4uA-
↪qKifSZ43CahzAJyUezOH9shI--tirC028lNg767ldEki3WnVr3zokSujC9YJ_9XXjw2hFBfmJUrNb0-
↪wldvxQbFU8RPXip-GQ_JPTrCTZhrzGFeWPvhA6Rqmc3b1PhM9jY7Dur1sjYWYVyXlFNCK3c-
↪6feo5WlRfe1aCWmwZQh6O18eTmLeT4nWYkDzQ"
}
```

Note: This could also be a JSON Web Key Set (JWKS) if it was contained within an array assigned to a *keys* member, e.g. `{ keys: [jwk1, jwk2] }`.

Just pass it in as a single line string, escaping the quotes:

```
jwt-secret = "{ \"alg\":\"RS256\", ... }"
```

To generate such a public/private key pair use a utility like latchset/jose.

```
jose jwk gen -i '{"alg": "RS256"}' -o rsa.jwk
jose jwk pub -i rsa.jwk -o rsa.jwk.pub

# now rsa.jwk.pub contains the desired JSON object
```

You can specify the literal value as we saw earlier, or reference a filename to load the JWK from a file:

```
jwt-secret = "@rsa.jwk.pub"
```

JWT security

There are at least three types of common critiques against using JWT: 1) against the standard itself, 2) against using libraries with known security vulnerabilities, and 3) against using JWT for web sessions. We'll briefly explain each critique, how PostgREST deals with it, and give recommendations for appropriate user action.

The critique against the JWT standard is voiced in detail elsewhere on the web. The most relevant part for PostgREST is the so-called `alg=none` issue. Some servers implementing JWT allow clients to choose the algorithm used to sign the JWT. In this case, an attacker could set the algorithm to `none`, remove the need for any signature at all and gain unauthorized access. The current implementation of PostgREST, however, does not allow clients to set the signature algorithm in the HTTP request, making this attack irrelevant. The critique against the standard is that it requires the implementation of the `alg=none` at all.

Critiques against JWT libraries are only relevant to PostgREST via the library it uses. As mentioned above, not allowing clients to choose the signature algorithm in HTTP requests removes the greatest risk. Another more subtle attack is possible where servers use asymmetric algorithms like RSA for signatures. Once again this is not relevant to PostgREST since it is not supported. Curious readers can find more information in this article. Recommendations about high quality libraries for usage in API clients can be found on jwt.io.

The last type of critique focuses on the misuse of JWT for maintaining web sessions. The basic recommendation is to stop using JWT for sessions because most, if not all, solutions to the problems that arise when you do, do not work. The linked articles discuss the problems in depth but the essence of the problem is that JWT is not designed to be secure and stateful units for client-side storage and therefore not suited to session management.

PostgREST uses JWT mainly for authentication and authorization purposes and encourages users to do the same. For web sessions, using cookies over HTTPS is good enough and well catered for by standard web frameworks.

HTTPS

PostgREST aims to do one thing well: add an HTTP interface to a PostgreSQL database. To keep the code small and focused we do not implement HTTPS. Use a reverse proxy such as NGINX to add this, here's how. Note that some Platforms as a Service like Heroku also add SSL automatically in their load balancer.

8.1.23 Schema Isolation

A PostgREST instance is configured to expose all the tables, views, and stored procedures of a single schema specified in a server configuration file. This means private data or implementation details can go inside a private schema and be invisible to HTTP clients. You can then expose views and stored procedures which insulate the internal details from the outside world. It keeps you code easier to refactor, and provides a natural way to do API versioning. For an example of wrapping a private table with a public view see the *Public User Interface* section below.

8.1.24 SQL User Management

Storing Users and Passwords

As mentioned, an external service can provide user management and coordinate with the PostgREST server using JWT. It's also possible to support logins entirely through SQL. It's a fair bit of work, so get ready.

The following table, functions, and triggers will live in a `basic_auth` schema that you shouldn't expose publicly in the API. The public views and functions will live in a different schema which internally references this internal information.

First we'll need a table to keep track of our users:

```
-- We put things inside the basic_auth schema to hide
-- them from public view. Certain public procs/views will
-- refer to helpers and tables inside.
create schema if not exists basic_auth;

create table if not exists
basic_auth.users (
  email    text primary key check ( email ~* '^.+@.+\..+$' ),
  pass     text not null check (length(pass) < 512),
  role     name not null check (length(role) < 512)
);
```

We would like the role to be a foreign key to actual database roles, however PostgreSQL does not support these constraints against the `pg_roles` table. We'll use a trigger to manually enforce it.

```
create or replace function
basic_auth.check_role_exists() returns trigger as $$
begin
  if not exists (select 1 from pg_roles as r where r.rolname = new.role) then
    raise foreign_key_violation using message =
      'unknown database role: ' || new.role;
    return null;
  end if;
  return new;
end
$$ language plpgsql;

drop trigger if exists ensure_user_role_exists on basic_auth.users;
```

(continues on next page)

(continued from previous page)

```
create constraint trigger ensure_user_role_exists
  after insert or update on basic_auth.users
  for each row
  execute procedure basic_auth.check_role_exists();
```

Next we'll use the pgcrypto extension and a trigger to keep passwords safe in the `users` table.

```
create extension if not exists pgcrypto;

create or replace function
basic_auth.encrypt_pass() returns trigger as $$
begin
  if tg_op = 'INSERT' or new.pass <> old.pass then
    new.pass = crypt(new.pass, gen_salt('bf'));
  end if;
  return new;
end
$$ language plpgsql;

drop trigger if exists encrypt_pass on basic_auth.users;
create trigger encrypt_pass
  before insert or update on basic_auth.users
  for each row
  execute procedure basic_auth.encrypt_pass();
```

With the table in place we can make a helper to check a password against the encrypted column. It returns the database role for a user if the email and password are correct.

```
create or replace function
basic_auth.user_role(email text, pass text) returns name
  language plpgsql
  as $$
begin
  return (
  select role from basic_auth.users
   where users.email = user_role.email
     and users.pass = crypt(user_role.pass, users.pass)
  );
end;
$$;
```

Public User Interface

In the previous section we created an internal table to store user information. Here we create a login function which takes an email address and password and returns JWT if the credentials match a user in the internal table.

Logins

As described in *JWT from SQL*, we'll create a JWT inside our login function. Note that you'll need to adjust the secret key which is hard-coded in this example to a secure (at least thirty-two character) secret of your choosing.

```
-- login should be on your exposed schema
create or replace function
```

(continues on next page)

(continued from previous page)

```
login(email text, pass text) returns basic_auth.jwt_token as $$
declare
  _role name;
  result basic_auth.jwt_token;
begin
  -- check email and password
  select basic_auth.user_role(email, pass) into _role;
  if _role is null then
    raise invalid_password using message = 'invalid user or password';
  end if;

  select sign(
      row_to_json(r), 'reallyreallyreallyreallyverysafe'
    ) as token
    from (
      select _role as role, login.email as email,
         extract(epoch from now())::integer + 60*60 as exp
    ) r
    into result;
  return result;
end;
$$ language plpgsql;
```

An API request to call this function would look like:

```
POST /rpc/login HTTP/1.1

{ "email": "foo@bar.com", "pass": "foobar" }
```

The response would look like the snippet below. Try decoding the token at jwt.io. (It was encoded with a secret of `reallyreallyreallyreallyverysafe` as specified in the SQL code above. You'll want to change this secret in your app!)

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJlbWFpbCI6ImZvb0BiYXIuY29tIiwicGFzcyI6ImZvb2JhciJ9.37066TTRlh-
↪1hXhnA9oO9Pj6lgL6zFuJU0iCHhuCFno"
}
```

Permissions

Your database roles need access to the schema, tables, views and functions in order to service HTTP requests. Recall from the *Overview of Role System* that PostgREST uses special roles to process requests, namely the authenticator and anonymous roles. Below is an example of permissions that allow anonymous users to create accounts and attempt to log in.

```
-- the names "anon" and "authenticator" are configurable and not
-- sacred, we simply choose them for clarity
create role anon;
create role authenticator noinherit;
grant anon to authenticator;

grant usage on schema public, basic_auth to anon;
grant select on table pg_authid, basic_auth.users to anon;
grant execute on function login(text,text) to anon;
```

You may be worried from the above that anonymous users can read everything from the `basic_auth.users` table. However this table is not available for direct queries because it lives in a separate schema. The anonymous role needs access because the public `users` view reads the underlying table with the permissions of the calling user. But we have made sure the view properly restricts access to sensitive information.

8.1.25 Hardening PostgREST

PostgREST is a fast way to construct a RESTful API. Its default behavior is great for scaffolding in development. When it's time to go to production it works great too, as long as you take precautions. PostgREST is a small sharp tool that focuses on performing the API-to-database mapping. We rely on a reverse proxy like Nginx for additional safeguards.

The first step is to create an Nginx configuration file that proxies requests to an underlying PostgREST server.

```
http {
  # ...
  # upstream configuration
  upstream postgrest {
    server localhost:3000;
    keepalive 64;
  }
  # ...
  server {
    # ...
    # expose to the outside world
    location /api/ {
      default_type  application/json;
      proxy_hide_header Content-Location;
      add_header Content-Location  /api/$upstream_http_content_location;
      proxy_set_header  Connection "";
      proxy_http_version 1.1;
      proxy_pass http://postgrest/;
    }
    # ...
  }
}
```

Block Full-Table Operations

Each table in the admin-selected schema gets exposed as a top level route. Client requests are executed by certain database roles depending on their authentication. All HTTP verbs are supported that correspond to actions permitted to the role. For instance if the active role can drop rows of the table then the DELETE verb is allowed for clients. Here's an API request to delete old rows from a hypothetical logs table:

```
DELETE /logs?time=lt.1991-08-06 HTTP/1.1
```

However it's very easy to delete the **entire table** by omitting the query parameter!

```
DELETE /logs HTTP/1.1
```

This can happen accidentally such as by switching a request from a GET to a DELETE. To protect against accidental operations use the pg-safeupdate PostgreSQL extension. It raises an error if UPDATE or DELETE are executed without specifying conditions. To install it you can use the PGXN network:

```
sudo -E pgxn install safeupdate

# then add this to postgresql.conf:
# shared_preload_libraries='safeupdate';
```

This does not protect against malicious actions, since someone can add a url parameter that does not affect the result set. To prevent this you must turn to database permissions, forbidding the wrong people from deleting rows, and using row-level security if finer access control is required.

Count-Header DoS

For convenience to client-side pagination controls PostgREST supports counting and reporting total table size in its response. As described in *Limits and Pagination*, responses ordinarily include a range but leave the total unspecified like

```
HTTP/1.1 200 OK
Range-Unit: items
Content-Range: 0-14/*
```

However including the request header `Prefer: count=exact` calculates and includes the full count:

```
HTTP/1.1 206 Partial Content
Range-Unit: items
Content-Range: 0-14/3573458
```

This is fine in small tables, but count performance degrades in big tables due to the MVCC architecture of PostgreSQL. For very large tables it can take a very long time to retrieve the results which allows a denial of service attack. The solution is to strip this header from all requests:

```
Nginx stuff. Remove any prefer header which contains the word count
```

Note: In future versions we will support `Prefer: count=estimated` to leverage the PostgreSQL statistics tables for a fast (and fairly accurate) result.

HTTPS

See the *HTTPS* section of the authentication guide.

Rate Limiting

Nginx supports "leaky bucket" rate limiting (see official docs). Using standard Nginx configuration, routes can be grouped into *request zones* for rate limiting. For instance we can define a zone for login attempts:

```
limit_req_zone $binary_remote_addr zone=login:10m rate=1r/s;
```

This creates a shared memory zone called "login" to store a log of IP addresses that access the rate limited urls. The space reserved, 10 MB (`10m`) will give us enough space to store a history of 160k requests. We have chosen to allow only allow one request per second (`1r/s`).

Next we apply the zone to certain routes, like a hypothetical stored procedure called `login`.

```
location /rpc/login/ {
  # apply rate limiting
  limit_req zone=login burst=5;
}
```

The burst argument tells Nginx to start dropping requests if more than five queue up from a specific IP.

Nginx rate limiting is general and indiscriminate. To rate limit each authenticated request individually you will need to add logic in a *Custom Validation* function.

8.1.26 Debugging

Server Version

When debugging a problem it's important to verify the PostgREST version. At any time you can make a request to the running server and determine exactly which version is deployed. Look for the `Server` HTTP response header, which contains the version number.

HTTP Requests

The PostgREST server logs basic request information to stdout, including the requesting IP address and user agent, the URL requested, and HTTP response status. However this provides limited information for debugging server errors. It's helpful to get full information about both client requests and the corresponding SQL commands executed against the underlying database.

A great way to inspect incoming HTTP requests including headers and query params is to sniff the network traffic on the port where PostgREST is running. For instance on a development server bound to port 3000 on localhost, run this:

```
# sudo access is necessary for watching the network
sudo ngrep -d lo0 port 3000
```

The options to ngrep vary depending on the address and host on which you've bound the server. The binding is described in the *Configuration* section. The ngrep output isn't particularly pretty, but it's legible.

Database Logs

Once you've verified that requests are as you expect, you can get more information about the server operations by watching the database logs. By default PostgreSQL does not keep these logs, so you'll need to make the configuration changes below. Find `postgresql.conf` inside your PostgreSQL data directory (to find that, issue the command `show data_directory;`). Either find the settings scattered throughout the file and change them to the following values, or append this block of code to the end of the configuration file.

```
# send logs where the collector can access them
log_destination = "stderr"

# collect stderr output to log files
logging_collector = on

# save logs in pg_log/ under the pg data directory
log_directory = "pg_log"

# (optional) new log file per day
log_filename = "postgresql-%Y-%m-%d.log"
```

(continues on next page)

(continued from previous page)

```
# log every kind of SQL statement
log_statement = "all"
```

Restart the database and watch the log file in real-time to understand how HTTP requests are being translated into SQL commands.

Note: On Docker you can enable the logs by using a custom `init.sh`:

```
#!/bin/sh
echo "log_statement = 'all'" >> /var/lib/postgresql/data/postgresql.conf
```

After that you can start the container and check the logs with `docker logs`.

```
docker run -v "$(pwd)/init.sh":"/docker-entrypoint-initdb.d/init.sh" -d postgres
docker logs -f <container-id>
```

Schema Reloading

Users are often confused by PostgREST's database schema cache. It is present because detecting foreign key relationships between tables (including how those relationships pass through views) is necessary, but costly. API requests consult the schema cache as part of *Resource Embedding*. However if the schema changes while the server is running it results in a stale cache and leads to errors claiming that no relations are detected between tables.

To refresh the cache without restarting the PostgREST server, send the server process a SIGUSR1 signal:

```
killall -SIGUSR1 postgrest
```

The above is the manual way to do it. To automate the schema reloads, use a database trigger like this:

```
CREATE OR REPLACE FUNCTION public.notify_ddl_postgrest()
  RETURNS event_trigger
 LANGUAGE plpgsql
  AS $$
BEGIN
  NOTIFY ddl_command_end;
END;
$$;

CREATE EVENT TRIGGER ddl_postgrest ON ddl_command_end
   EXECUTE PROCEDURE public.notify_ddl_postgrest();
```

Then run the pg_listen utility to monitor for that event and send a SIGUSR1 when it occurs:

```
pg_listen <db-uri> ddl_command_end "killall -SIGUSR1 postgrest"
```

Now, whenever the structure of the database schema changes, PostgreSQL will notify the `ddl_command_end` channel, which will cause `pg_listen` to send PostgREST the signal to reload its cache.

Important: As of PostgREST v5.1 reloading with SIGHUP is deprecated, it's still supported but will be removed in v6.0. SIGUSR1 should be used instead.

8.1.27 Daemonizing

For linux distros that use **systemd** (ubuntu, debian, archlinux) you can create a daemon in the following way.

First, create postgrest configuration in `/etc/postgrest/config`

```
db-uri = "postgres://<your_user>:<your_password>@localhost:5432/<your_db>"
db-schema = "<your_exposed_schema>"
db-anon-role = "<your_anon_role>"
db-pool = 10

server-host = "127.0.0.1"
server-port = 3000

jwt-secret = "<your_secret>"
```

Then create the systemd service file in `/etc/systemd/system/postgrest.service`

```
[Unit]
Description=REST API for any Postgres database
After=postgresql.service

[Service]
ExecStart=/bin/postgrest /etc/postgrest/config
ExecReload=/bin/kill -SIGUSR1 $MAINPID

[Install]
WantedBy=multi-user.target
```

After that, you can enable the service at boot time and start it with:

```
systemctl enable postgrest
systemctl start postgrest

## For reloading the service
## systemctl restart postgrest
```

8.1.28 Alternate URL Structure

As discussed in *Singular or Plural*, there are no special URL forms for singular resources in PostgREST, only operators for filtering. Thus there are no URLs like `/people/1`. It would be specified instead as

```
GET /people?id=eq.1
Accept: application/vnd.pgrst.object+json
```

This allows compound primary keys and makes the intent for singular response independent of a URL convention.

Nginx rewrite rules allow you to simulate the familiar URL convention. The following example adds a rewrite rule for all table endpoints, but you'll want to restrict it to those tables that have a numeric simple primary key named "id."

```
# support /endpoint/:id url style
location ~ ^/([a-z_]+)/([0-9]+) {

  # make the response singular
  proxy_set_header Accept 'application/vnd.pgrst.object+json';
```

(continues on next page)

(continued from previous page)

```
  # assuming an upstream named "postgrest"
  proxy_pass http://postgrest/$1?id=eq.$2;

}
```

CHAPTER 9

Ecosystem

PostgREST has a growing ecosystem of examples, and libraries, experiments, and users. Here is a selection.

CHAPTER 10

Example Apps

- subzerocloud/postgrest-starter-kit - Boilerplate for new project
- NikolayS/postgrest-google-translate - Calling to external translation service
- CodeforAustralia/heritage-near-me - Elm and PostgREST with PostGIS
- timwis/handsontable-postgrest - An excel-like database table editor
- Recmo/PostgrestSkeleton - Docker Compose, PostgREST, Nginx and Auth0
- benoror/ember-postgrest-dynamic-ui - generating Ember forms to edit data
- ruslantalpa/blogdemo - blog api demo in a vagrant image
- timwis/ext-postgrest-crud - browser-based spreadsheet
- srid/chronicle - tracking a tree of personal memories
- diogob/elm-workshop - building a simple database query UI
- marmelab/ng-admin-postgrest - automatic database admin panel
- myfreeweb/moneylog - accounting web app in Polymer + PostgREST
- tyrchen/goodfilm - example film api
- begriffs/postgrest-example - sqitch versioning for API
- SMRxT/postgrest-demo - multi-tenant logging system
- PierreRochard/postgrest-boilerplate - example auth back-end

CHAPTER 11

Client-Side Libraries

- tomberek/aor-postgrest-client - JS, admin-on-rest
- hugomrdias/postgrest-url - JS, just for generating query URLs
- john-kelly/elm-postgrest - Elm
- mithril.postgrest - JS, Mithril
- lewisjared/postgrest-request - JS, SuperAgent
- JarvusInnovations/jarvus-postgrest-apikit - JS, Sencha framework
- davidthewatson/postgrest_python_requests_client - Python
- datrium/postgrest-pyclient - Python
- calebmer/postgrest-client - JS
- clesiemo3/postgrestR - R
- PierreRochard/postgrest-angular - TypeScript, generate UI from API description
- thejettdurham/postgrest-sharp-client (needs maintainer) - C#, RestSharp
- team142/ng-postgrest - Angular app for browsing, editing data exposed over Postgrest.

CHAPTER 12

External Notification

These are PostgreSQL bridges that propagate LISTEN/NOTIFY to external queues for further processing. This allows stored procedures to initiate actions outside the database such as sending emails.

- diogob/postgres-websockets - expose web sockets for PostgreSQL's LISTEN/NOTIFY
- frafra/postgresql2websocket - Websockets
- matthewmueller/pg-bridge - Amazon SNS
- aweber/pgsql-listen-exchange - RabbitMQ
- SpiderOak/skeeter - ZeroMQ
- FGRibreau/postgresql-to-amqp - AMQP
- daurnimator/pg-kinesis-bridge - Amazon Kinesis

CHAPTER 13

Extensions

- pg-safeupdate - Prevent full-table updates or deletes
- srid/spas - allow file uploads and basic auth
- svmnotn/postgrest-auth - OAuth2-inspired external auth server
- wildsurfer/postgrest-oauth-server - OAuth2 server
- nblumoe/postgrest-oauth - OAuth2 WAI middleware
- criles25/postgrest-auth - email based auth/signup
- ppKrauss/PostgREST-writeAPI - generate Nginx rewrite rules to fit an OpenAPI spec

CHAPTER 14

Commercial

- subZero - Automated GraphQL & REST API with built-in caching (powered in part by PostgREST)

CHAPTER 15

In Production

- Moat
- Catarse
- Redsmin
- Image-charts
- MotionDynamic - Fast highly dynamic video generation at scale
- Drip Depot
- OpenBooking
- Convene by Thomson-Reuters
- eGull
- Elyios
- Simply Connected Systems
- Nimbus
  - See how Nimbus uses PostgREST in Paul Copplestone's blog post.
- triggerFS - A realtime messaging and distributed trigger system
- Datrium

CHAPTER 16

Testimonials

"It's so fast to develop, it feels like cheating!"

—François-G. Ribreau

"I just have to say that, the CPU/Memory usage compared to our Node.js/Waterline ORM based API is ridiculous. It's hard to even push it over 60/70 MB while our current API constantly hits 1GB running on 6 instances (dynos)."

—Louis Brauer

"I really enjoyed the fact that all of a sudden I was writing microservices in SQL DDL (and v8 javascript functions). I dodged so much boilerplate. The next thing I knew, we pulled out a full rewrite of a Spring+MySQL legacy app in 6 months. Literally 10x faster, and code was super concise. The old one took 3 years and a team of 4 people to develop."

—Simone Scarduzio

"I like the fact that PostgREST does one thing, and one thing well. While PostgREST takes care of bridging the gap between our HTTP server and PostgreSQL database, we can focus on the development of our API in a single language: SQL. This puts the database in the center of our architecture, and pushed us to improve our skills in SQL programming and database design."

—Eric Bréchemier, Data Engineer, eGull SAS

"PostgREST is performant, stable, and transparent. It allows us to bootstrap projects really fast, and to focus on our data and application instead of building out the ORM layer. In our k8s cluster, we run a few pods per schema we want exposed, and we scale up/down depending on demand. Couldn't be happier."

—Anupam Garg, Datrium, Inc.