PostgREST Documentation

Release 4.1.0

Joe Nelson

Jul 19, 2017

Contents

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.

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 1

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 2

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 3

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 4

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 5

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 6

Ecosystem

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

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
- calebmer/postgrest-client - JS
- clesiemo3/postgrestR - R
- PierreRochard/postgrest-angular - TypeScript, generate UI from API description
- thejettdurham/postgrest-sharp-client (needs maintainer) - C#, RestSharp

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.

- frafra/postgresql2websocket - Websockets
- matthewmueller/pg-bridge - Amazon SNS

- aweber/pgsql-listen-exchange - RabbitMQ
- SpiderOak/skeeter - ZeroMQ
- FGRibreau/postgresql-to-amqp - AMQP

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 backend

In Production

- Catarse
- iAdvize
- Redsmin
- Image-charts
- Drip Depot
- OpenBooking
- Convene by Thomson-Reuters
- eGull

Extensions

- ppKrauss/PostgREST-writeAPI - generate Nginx rewrite rules to fit an OpenAPI spec
- diogob/postgrest-ws - expose web sockets for PostgreSQL's LISTEN/NOTIFY
- pg-safeupdate - Prevent full-table updates or deletes
- srid/spas - allow file uploads and basic auth
- svmnotn/postgrest-auth - OAuth2-inspired external auth server
- nblumoe/postgrest-oauth - OAuth2 WAI middleware

Commercial

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

CHAPTER 7

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

CHAPTER 8

Getting Support

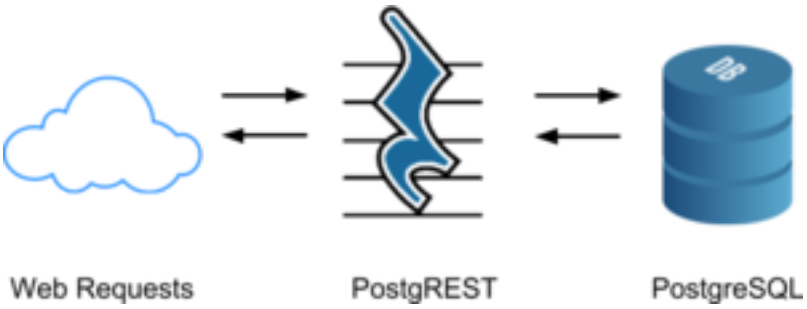
The project has a friendly and growing community. Join our chat room for discussion and help. You can also report or search for bugs/features on the Github issues page.

CHAPTER 9

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 5432:5432 \
                -e POSTGRES_PASSWORD=mysecretpassword \
                -d postgres
```

This will run the Docker instance as a daemon and expose port 5432 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/begriffs/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 to 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 web_anon to postgres;

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.

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://postgres:mysecretpassword@localhost/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*

CHAPTER 10

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 postgres;

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.

Note: The OpenSSL toolkit provides an easy way to generate a secure password. If you have it installed, run

```
openssl rand -base64 32
```

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

```
# add this line to tutorial.conf

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

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:

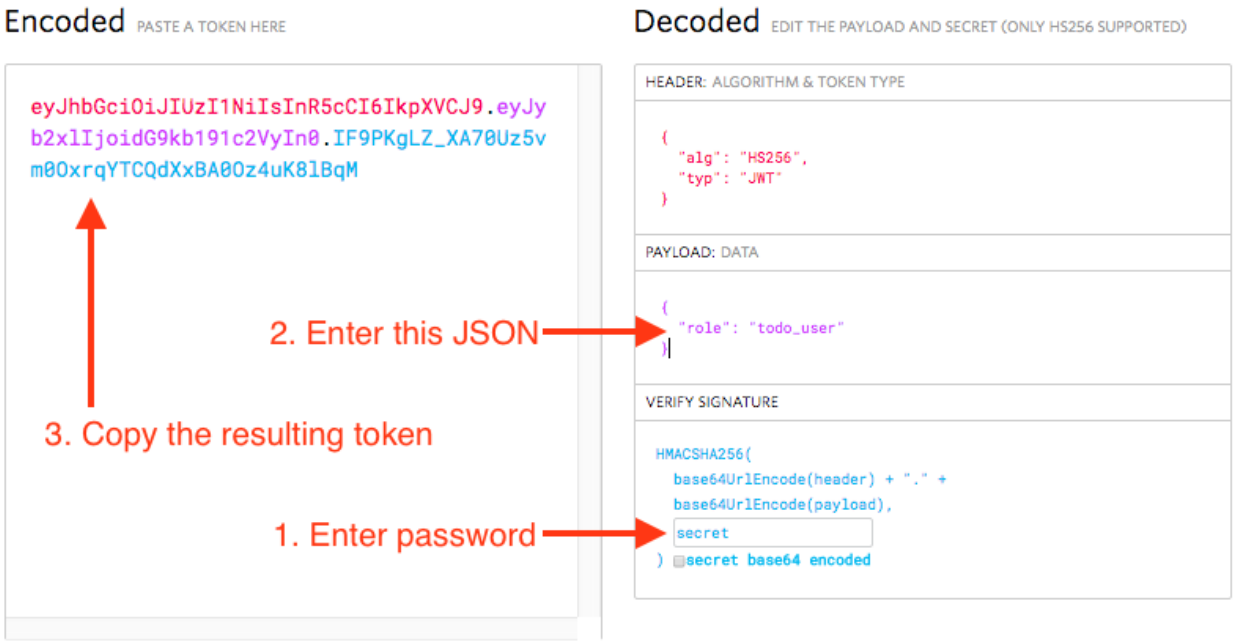


Fig. 10.1: How to create a token at https://jwt.io

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.

```
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
  },
  {
    "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 \
     -H "Authorization: Bearer $TOKEN"

# this one is rejected

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

The server responds with 403 Forbidden:

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

CHAPTER 11

Binary Release

[Download from release page]

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

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

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

# Try running it
$ ./postgrest --help

# You should see a usage help message
```

CHAPTER 12

Homebrew

You can use the Homebrew package manager to install PostgREST on Mac

```
# Ensure brew is up to date
brew update

# Check for any problems with brew's setup
brew doctor

# Install the postgrest package
brew install postgrest
```

This will automatically install PostgreSQL as a dependency. The process tends to take up to 15 minutes to install the package and its dependencies.

After installation completes, the tool is added to your $PATH and can be used from anywhere with:

```
postgrest --help
```

CHAPTER 13

PostgreSQL dependency

To use PostgREST you will need an underlying database (PostgreSQL version 9.3 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

CHAPTER 14

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-server
OS X	postgresql, gmp

- Build and install binary

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

# 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 Postgres 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 postgres server without any modifications to the server. (Such as allowing accounts without a passoword 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 postgres 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 MacOS 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 Postgres.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, Postgres.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
```

CHAPTER 15

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	
server-host	String	*4	
server-port	Int	3000	
server-proxy-uri	String		
jwt-secret	String		
secret-is-base64	Bool	False	
max-rows	Int	∞	
pre-request	String		

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. 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"`. Also allows connections over Unix sockets for higher performance.

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

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.

server-host Where to bind the PostgREST web server.

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": "0.4.0.0",
    "title": "PostgREST API",
    "description": "This is a dynamic API generated by PostgREST"
  },
  "host": "postgrest.com:443",
  "basePath": "/",
  "schemes": [
    "https"
  ]
}
```

jwt-secret The secret used to decode JWT tokens clients provide for authentication. 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.

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.

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 logfile 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*.)

CHAPTER 16

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 *SSL* 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 indescriminate. To rate limit each authenticated request individually you will need to add logic in a *Custom Validation* function.

CHAPTER 17

Debugging

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. Note the `Server` response header as well which identifies the version of server. This is important when submitting bug reports.

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"

# 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.

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 SIGHUP signal:

```
killall -HUP postgrest
```

In the future we're investigating ways to keep the cache updated without manual intervention.

CHAPTER 18

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';

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

}
```

CHAPTER 19

Tables and Views

All views and tables in the active 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
```

Adding multiple parameters conjoins the conditions:

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

These operators are available:

abbreviation	meaning
eq	equals
gte	greater than or equal
gt	greater than
lte	less than or equal
lt	less than
neq	not equal
like	LIKE operator (use * in place of %)
ilike	ILIKE operator (use * in place of %)
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"`
is	checking for exact equality (null,true,false)
@@	full-text search using to_tsquery
@>	contains e.g. `?tags=@>.{example, new}`
<@	contained in e.g. `?values=<@{1,2,3}`
not	negates another operator, see below

To negate any operator, prefix it with `not` like `?a=not.eq.2`.

For more complicated filters (such as those involving disjunctions) 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
```

Note: We're working to extend the PostgREST query grammar to allow more complicated boolean logic, while continuing to prevent performance problems from arbitrary client queries.

Vertical Filtering (Columns)

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

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

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

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=@@.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
```

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 empty body and 404 status code rather than the usual empty array and 200 status.

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 singlular 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`.

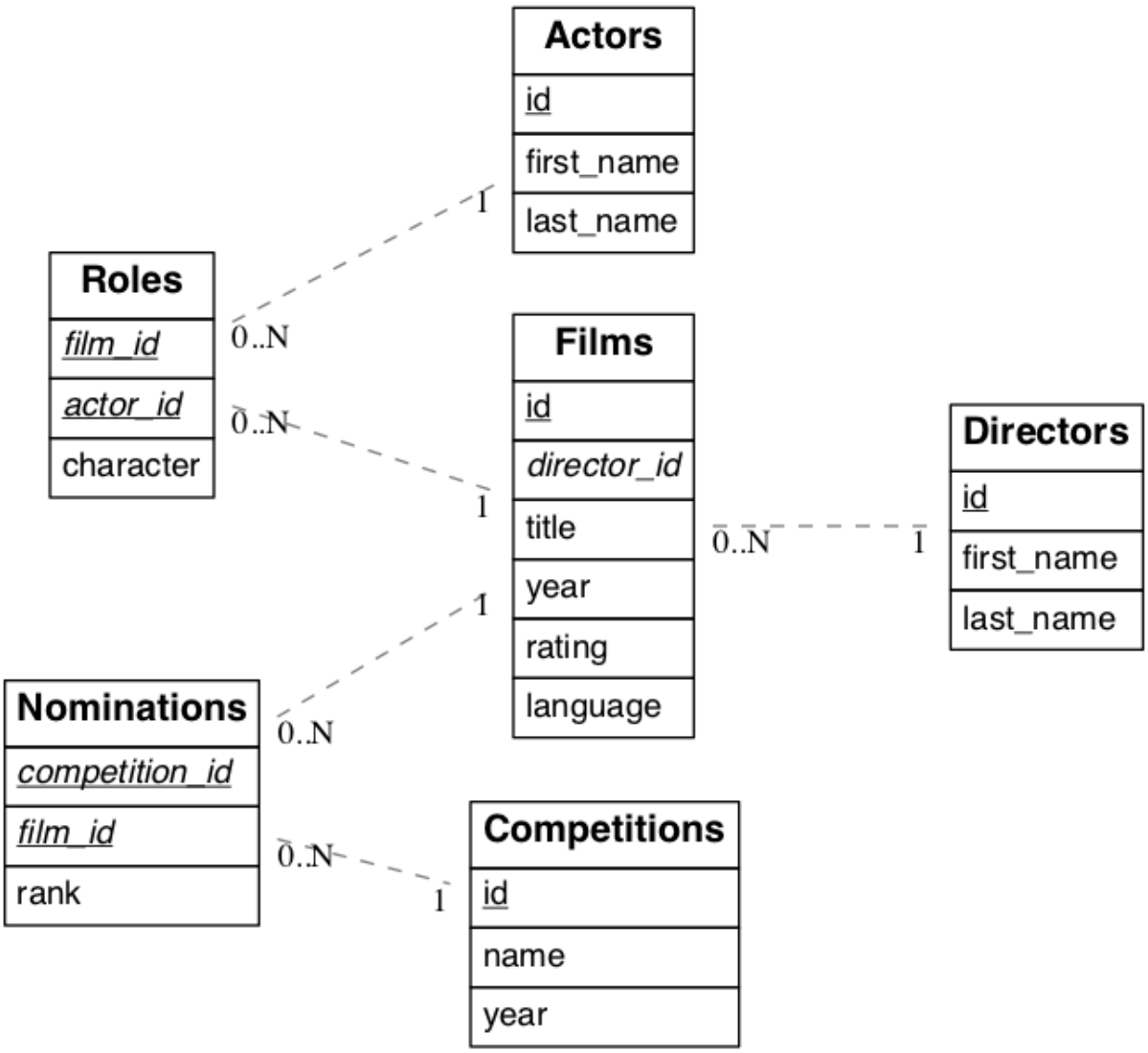
```
GET /items?select=bin_data&id=eq.1 HTTP/1.1
Accept: application/octet-stream
```

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

CHAPTER 20

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(last_name) HTTP/1.1
```

Which would return

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

Note: As of PostgREST v4.1, parens `()` are used rather than brackets `{}` for the list of embedded columns. Brackets are still supported, but are deprecated and will be removed in v5.

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
```

Note: 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 and Order

Embedded tables can be filtered and ordered similarly to their top-level counterparts. To do so, prefix the query parameters with the name of the embedded table. 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.

CHAPTER 21

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 and OR-conditions in the where clause
- More complicated joins than those provided by *Resource Embedding*
- Geospatial queries that require an argument, like "points near (lat,lon)"
- More sophisticated full-text search than a simple use of the `@@` filter

CHAPTER 22

Stored Procedures

Every stored procedure in the API-exposed database schema is accessible under the `/rpc` prefix. The API endpoint supports only POST which executes 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). However 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.

Procedures must be used with named arguments. 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 $$
 SELECT $1 + $2;
$$ LANGUAGE SQL IMMUTABLE STRICT;
```

The client can call it by posting an object like

```
POST /rpc/add_them HTTP/1.1

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

The keys of the object match the parameter names. Note that PostgreSQL converts parameter names to lowercase unless you quote them like `CREATE FUNCTION foo("mixedCase" text) ...`. 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.

Note: We recommend using function arguments of type json to accept arrays from the client. To pass a PostgreSQL native array you'll need to quote it as a string:

```
POST /rpc/native_array_func HTTP/1.1

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

```
POST /rpc/json_array_func HTTP/1.1

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

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).

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 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.

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.

We are considering allowing GET requests for functions that are marked non-volatile. Allowing GET is important for HTTP caching. However we still must decide how to pass function parameters since request bodies are not allowed. Also some query string arguments are already reserved for shaping/filtering the output.

Accessing Request Headers/Cookies

Stored procedures can access request headers and cookies by reading GUC variables set by PostgREST per request. They are named `request.header.XYZ` and `request.cookie.XYZ`. For example, to read the value of the Origin request header:

```
SELECT current_setting('request.header.origin', true);
```

Complex boolean logic

For complex boolean logic you can use stored procedures, an example:

```
CREATE FUNCTION key_customers(country TEXT, company TEXT, salary FLOAT) RETURNS SETOF␣
↪customers AS $$
  SELECT * FROM customers WHERE (country = $1 AND company = $2) OR salary = $3;
$$ LANGUAGE SQL;
```

Then you can query by doing:

```
POST /rpc/key_customers HTTP/1.1

{ "country": "Germany", "company": "Volkswagen", salary": 120000.00 }
```

Raising Errors

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

```
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"
}
```

You can customize the HTTP status code 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.

CHAPTER 23

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.

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 exmaple 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)*.

Note: 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 backend 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 }
]
```

CHAPTER 24

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
```

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

CHAPTER 25

OpenAPI Support

Every API hosted by PostgREST automatically serves a full OpenAPI description on the root path. This provides a list of all endpoints, along with supported HTTP verbs and example payloads.

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, provides guidance with request headers and example request bodies.

Note: 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*.

CHAPTER 26

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 prereq 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 authed 403, else 401	insufficient privileges
other	500	

CHAPTER 27

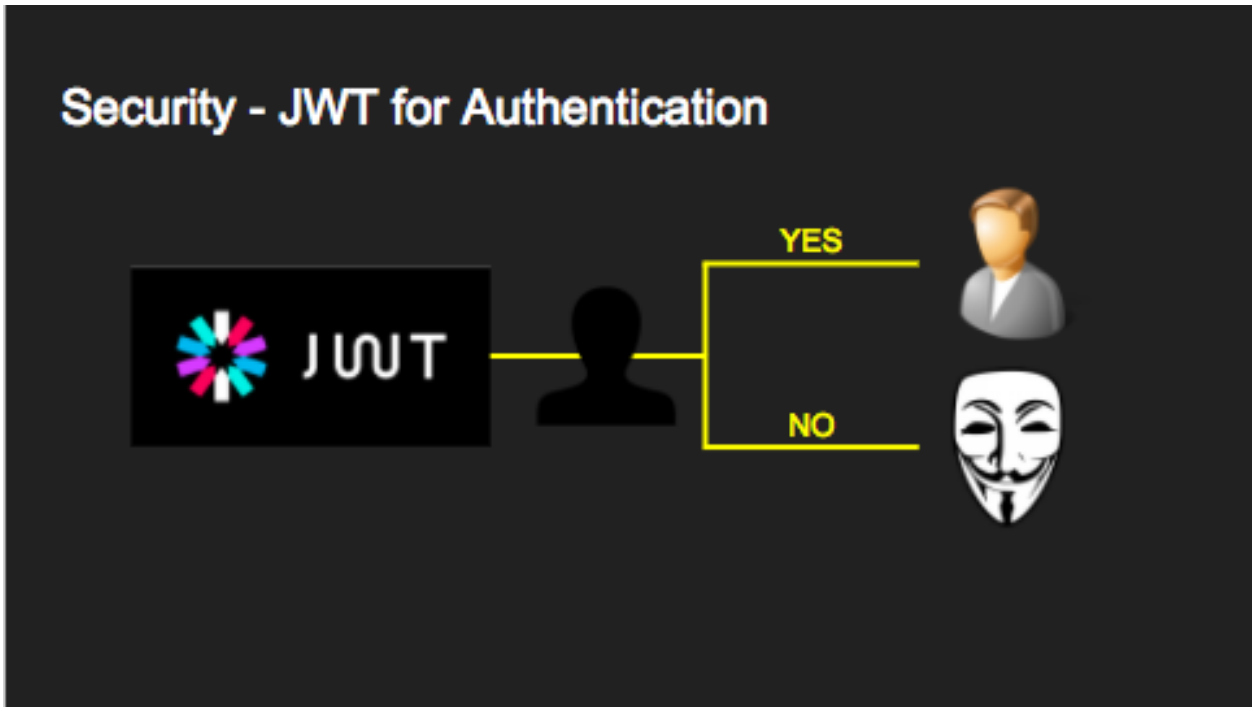
Overview of Role System

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

CHAPTER 28

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 passphrase – the signer and verifier share the secret. Hence any service that shares a passphrase with a PostgREST server can create valid JWT. (PostgREST currently supports only the HMAC-SHA256 signing algorithm.)

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 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
    LANGUAGE sql
    AS $$
```

```
  SELECT sign(
    row_to_json(r), 'mysecret'
  ) AS token
  FROM (
    SELECT
      'my_role'::text as role,
      extract(epoch from now())::integer + 300 AS exp
  ) r;
$$;
```

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 '!!secret!!';

-- 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, copy its client secret into your PostgREST configuration file as the `jwt-secret`. (Old-style Auth0 secrets are Base64 encoded. For these secrets set `secret-is-base64` to `true`, or just refresh the Auth0 secret.) You can find the secret in the client settings of the Auth0 management console.

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 metadata and include a `role` claim in the payload of our user object. Afterwards, in your Auth0Lock code, include the `role` claim in your scope param.

```
// Example Auth0 rule
function (user, context, callback) {
  user.app_metadata = user.app_metadata || {};
  user.role = user.app_metadata.role;
  callback(null, user, context);
}
```

```
// Example using Auth0Lock with role claim in scope
new Auth0Lock ( AUTH0_CLIENTID, AUTH0_DOMAIN, {
  container: 'lock-container',
  auth: {
    params: { scope: 'openid role' },
    redirectUrl: FQDN + '/login', // Replace with your redirect url
    responseType: 'token'
  }
})
```

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.

SSL

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 SSL. 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.

CHAPTER 29

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.

CHAPTER 30

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
  language plpgsql
  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
$$;

drop trigger if exists ensure_user_role_exists on basic_auth.users;
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
  language plpgsql
  as $$
begin
  if tg_op = 'INSERT' or new.pass <> old.pass then
    new.pass = crypt(new.pass, gen_salt('bf'));
  end if;
  return new;
end
$$;

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 secret of your choosing.

```
create or replace function
login(email text, pass text) returns basic_auth.jwt_token
  language plpgsql
  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), 'mysecret'
    ) 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;
$$;
```

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 `mysecret` as specified in the SQL code above. You'll want to change this secret in your app!)

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
↪eyJlbWFpbCI6ImZvb0BiYXIuY29tIiwicm9sZSI6ImF1dGhvciJ9.fpf3_
↪ERi5qbWOE5NPzvauJgvulm0zkIG9xSm2w5zmdw"
}
```

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.