# What is PostgREST?

## 1 Motivation

1.1 Declarative Programming  
1.2 Leakproof Abstraction  
1.3 Embracing the Relational Model  
1.4 One Thing Well  
1.5 Shared Improvements

## 2 Myths

2.1 You have to make tons of stored procs and triggers  
2.2 Exposing the database destroys encapsulation

## 3 Conventions

## 4 Installation

4.1 Installing from Pre-Built Release  
4.2 Building from Source  
4.3 Running the Server  
4.4 Install via Homebrew (Mac OS X)  
4.5 Installing PostgreSQL

## 5 Ecosystem

5.1 Client-Side Libraries  
5.2 Extensions  
5.3 Example Apps  
5.4 In Production  
5.5 Commercial

## 6 Requesting Information

6.1 Tables and Views  
6.2 Stored Procedures  
6.3 Filtering  
6.4 Ordering  
6.5 Limiting and Pagination  
6.6 Embeding Foreign Entities  
6.7 Response Format  
6.8 Singular vs Plural  
6.9 Data Schema  
6.10 CORS
### 7 Updating Data
- 7.1 Record Creation ........................................... 21
- 7.2 Record Update ........................................... 21
- 7.3 Bulk Insertion ........................................... 21
- 7.4 Multiple Tables Insertion or Update ......................... 22
- 7.5 Bulk Updates ........................................... 23
- 7.6 Deletion ............................................... 23
- 7.7 Protecting Dangerous Actions ......................... 24

### 8 Security
- 8.1 Database Roles ....................................... 25
- 8.2 Row-Level Security ................................ 26
- 8.3 Building Auth on top of JWT .......................... 26
- 8.4 SSL .................................................. 26

### 9 API Versioning
- 9.1 Schema Search Path ................................ 27
- 9.2 Changing a Resource ................................ 27
- 9.3 Removing a Resource ................................ 27
- 9.4 Avoiding DB and Client Coupling .................. 27

### 10 Data Migration
- 10.1 Sqitch .............................................. 29
- 10.2 Test-Driven Migrations ............................. 29

### 11 Deployment
- 11.1 Heroku .............................................. 31
- 11.2 Debian ............................................... 43

### 12 Performance
- 12.1 Benchmarks .......................................... 45
- 12.2 Caching ............................................. 45
- 12.3 Quality of Service ................................ 45
- 12.4 Tips ................................................ 45

### 13 Getting Started
- 13.1 Your First (simple) API ......................... 47
- 13.2 Populating Data ................................ 48
- 13.3 Getting and Embedding Data .................. 49
- 13.4 Singular Responses ................................ 52
- 13.5 Conclusion ......................................... 53

### 14 User Management
- 14.1 Storing Users and Passwords .................... 55
- 14.2 Password Reset .................................... 57
- 14.3 Email Validation ................................ 59
- 14.4 Editing Own User ................................ 59
- 14.5 Generating JWT ................................... 61
- 14.6 Same-Role Users .................................. 62
- 14.7 Permissions .......................................... 62
- 14.8 Conclusion ......................................... 63

### 15 Multi-Tenant Blog
- 15.1 Adding Blog-Specific Tables .................... 65
- 15.2 Permissions .......................................... 66
PostgREST is a standalone web server that turns your database directly into a RESTful API. The structural constraints and permissions in the database determine the API endpoints and operations.

This guide explains how to install the software and provides practical examples of its use. You’ll learn how to build a fast, versioned, secure API and how to deploy it to production.

The project has a friendly and growing community. Here are some ways to get help or get involved:

- The project chat room
- Report or search issues
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.

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 set constraints than to litter code with sanity checks.

Leakproof 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.

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.

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.
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 codebases.
You have to make tons of stored procs and triggers

Modern PostgreSQL features like auto-updatable views and computed columns make this mostly unnecessary. Triggers do play a part, but generally not for irksome boilerplate. When they are required triggers are preferable to ad-hoc app code anyway, since the former work reliably for any codepath.

Exposing the database destroys encapsulation

PostgREST does versioning through database schemas. This allows you to expose tables and views without making the app brittle. Underlying tables can be superseded and hidden behind public facing views. The chapter about versioning shows how to do this.
This guide contains highlighted notes and tangential information interspersed with the text.
Installation

Installing from Pre-Built Release

The release page has precompiled binaries for Mac OS X, Windows, and several Linux distros. Extract the tarball and run the binary inside with no arguments to see usage instructions:

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

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

# Try running it
$ ./postgrest

# You should see a usage help message

Building from Source

When a prebuilt 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

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ubuntu/Debian</td>
<td>libpq-dev</td>
</tr>
<tr>
<td>CentOS/Fedora/Red Hat</td>
<td>postgresql-devel, zlib-devel</td>
</tr>
<tr>
<td>BSD</td>
<td>postgresql95-server</td>
</tr>
</tbody>
</table>

- Build & install in one step

  git clone https://github.com/begriffs/postgrest.git
  cd postgrest
  stack build --install-ghc
  sudo stack install --allow-different-user --local-bin-path /usr/local/bin

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

If you want to run the test suite, stack can do that too: stack test.
Running the Server

```
postgrest postgres://user:pass@host:port/db -a anon_user [other flags]
```

The user in the connection string is the “authenticator role,” i.e. a role which is used temporarily to switch into other roles depending on the authentication request JWT. For simple APIs you can use the same role for authenticator and anonymous.

The complete list of options:

When running `postgrest` on the same machine as PostgreSQL, it is also possible to connect to the database using the Unix socket and the Peer Authentication method as an alternative to TCP/IP communication and authentication with a password.

The Peer Authentication grants access to the database to any Unix user who connects as a user of the same name in the database. Since the empty host resolves to the Unix socket and the password can be omitted in this case, the command line is reduced to:

```
sudo -u user postgrest postgres://user@/db [flags]
```

where the `sudo -u user` command runs the following command as given `user`.

If you create a Unix user `postgrest` and a database user `postgrest` for example, the command becomes:

```
sudo -u postgrest postgrest postgres://postgrest@/db [flags]
```

The first `postgrest` is the Unix user name, the second `postgrest` is the name of the executable, the third `postgrest` is the name of the database user.

Install via Homebrew (Mac OS X)

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 (see the `Installing PostgreSQL` section for setup instructions). 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
```

Installing PostgreSQL

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
Client-Side Libraries

- calebmer/postgrest-client - Advanced JS client for the PostgREST API
- mithril.postgrest - Mithril plugin to create and authenticate requests
- lewisjared/postgrest-request - node interface to postgrest instances
- JarvusInnovations/jarvus-postgrest-apikit - Sencha framework package for binding models/stores/proxies to PostgREST tables
- davidthewatson/postgrest_python_requests_client - python client featuring JWT auth and pagination of result sets

Extensions

- diogob/postgrest-ws - expose web sockets for PostgreSQL’s LISTEN/NOTIFY
- srid/spas - allow file uploads and basic auth

Example Apps

- CodeforAustralia/heritage-near-me - Elm and PostgREST with PostGIS
- benoror/ember-postgrest-dynamic-ui - generating Ember forms to edit data
- rulantalpa/blogdemo - blog api demo in a vagrant image
- timwis/ext-postgrest-crud - browser-based spreadsheet
- srid/chronicle - tracking a tree of personal memories
- begriffs/postgrest-example - how to configure a db for use as an API
- marmelab/ng-admin-postgrest - automatic database admin panel
- tyrchen/goodfilm - example film api
In Production

- Catarse
- Drip Depot

Commercial

- Sub0 - Automated GraphQL & REST API for your PostgreSQL database with built-in caching
CHAPTER 6

Requesting Information

Tables and Views

- Cacheable, prefetchable
- Idempotent

The list of accessible tables and views is provided at

GET /

Every view and table accessible by the active db role is exposed in a one-level deep route. For instance the full contents of a table people is returned at

GET /people

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

Stored Procedures

- Cannot necessarily be cached or prefetched
- Not necessarily idempotent

Every stored procedure is accessible under the /rpc prefix. The API endpoint supports only POST which executes the function.

POST /rpc/proc_name

PostgREST supports calling procedures with named arguments. 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 PostgREST client can call it by posting an object like { "a": 1, "b": 2} to /rpc/add_them. 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) ....
Filtering

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:

```plaintext
GET /people?age=lt.13
```

Adding multiple parameters conjoins the conditions:

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

These operators are available:

<table>
<thead>
<tr>
<th>abbreviation</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq</td>
<td>equals</td>
</tr>
<tr>
<td>gte</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>gt</td>
<td>greater than</td>
</tr>
<tr>
<td>lte</td>
<td>less than or equal</td>
</tr>
<tr>
<td>lt</td>
<td>less than</td>
</tr>
<tr>
<td>neq</td>
<td>not equal</td>
</tr>
<tr>
<td>like</td>
<td>LIKE operator (use * in place of %)</td>
</tr>
<tr>
<td>ilike</td>
<td>ILIKE operator (use * in place of %)</td>
</tr>
<tr>
<td>in</td>
<td>one of a list of values e.g. ?a=in.1,2,3</td>
</tr>
<tr>
<td>notin</td>
<td>not one of a list of values e.g. ?a=notin.1,2,3</td>
</tr>
<tr>
<td>is</td>
<td>checking for exact equality (null,true,false)</td>
</tr>
<tr>
<td>isnot</td>
<td>checking for exact inequality (null,true,false)</td>
</tr>
<tr>
<td>@@</td>
<td>full-text search using to_tsquery</td>
</tr>
<tr>
<td>@&gt;</td>
<td>contains e.g. ?tags=@&gt;.{example, new}</td>
</tr>
<tr>
<td>&lt;@</td>
<td>contained in e.g. values=&lt;@{1,2,3}</td>
</tr>
<tr>
<td>not</td>
<td>negates another operator, see below</td>
</tr>
</tbody>
</table>

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

For more complicated filters (such as those involving condition 1 OR condition 2) you will have to create a new view in the database.

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.

Filtering Columns

You can customize which columns are returned by using the `select` parameter:

```plaintext
GET /people?select=age,height,weight
```

To cast the column types, add a double colon

```plaintext
GET /people?select=age::text,height,weight
```

Not all type coercions are possible, and you will get an error describing any problems from selection or type casting.

The `select` keyword is reserved. You thus cannot filter rows based on a column named `select`. Then again it is a reserved SQL keyword too, hence an unlikely column name.
**Inside JSONB**

PostgreSQL >=9.4.2 supports native JSON columns and can even index them by internal keys using the `jsonb` column type. PostgREST allows you to filter results by internal JSON object values. Use the single- and double-arrows to path into and obtain values, e.g.

```
GET /stuff?json_col->a->>b=eq.2
```

This query finds rows in `stuff` where `json_col->'a'->>'b'` is equal to 2 (or “2” – it coerces as needed). The final arrow must be the double kind, `->>`, or else PostgREST will not attempt to look inside the JSON.

**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
```

If no direction is specified it defaults to ascending order:

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

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

```
GET /people?order=age.nullsfirst
GET /people?order=age.desc.nullslast
```

You can order by `jsonb` object values with the same criteria:

```
GET /people?order=json_col->a->>b.desc.nullslast
```

To order the embedded items, you need to specify the tree path for the order param like so.

```
GET /projects?select=id,name,tasks{id,name}&order=id.asc&tasks.order=name.asc
```

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

**Limiting and Pagination**

**Pagination by Limit-Offset**

PostgREST uses HTTP range headers for limiting and describing the size of results. Every response contains the current range and total results:

```
Range-Unit: items
Content-Range → 0-14/15
```

This means items zero through fourteen are returned out of a total of fifteen – i.e. all of them. This information is available in every response and can help you render pagination controls on the client. This is a RFC7233-compliant solution that keeps the response JSON cleaner.

The client can set the limit and offset of a request by setting the `Range` header. Translate the limit and offset into a range. To request the first five elements, include these request headers:
You can also use open-ended ranges for an offset with no limit: Range: 10-.

In addition to the Range header, you can use &limit and &offset parameters to achieve the same result.

You can also set a limit (but not offset) for the embedded items like so

/posts?select=id,title,body,comments(id,email,body)&limit=10&comments.limit=3

The above request will return the first 10 posts and for each of the posts, 3 comments at most

**Suppressing Counts**

Sometimes knowing the total row count of a query is unnecessary and only adds extra cost to the database query. So you can skip the count total using a Prefer header as:

Prefer: count=none

With count suppressed the PostgREST response will look like:

```
Range-Unit: items
Content-Range → 0-14/*
```

**Embedding Foreign Entities**

To help you make fewer requests, PostgREST allows the embedding of traditional SQL relationships into a response. Suppose you have a projects table which references clients through a foreign key called client_id. When listing projects through the API you can have it embed the client within each project response. For example,

GET /projects?id=eq.1&select=id, name, clients{*}

Notice this is the same select keyword which is used to choose which columns to include. When a column name is followed by parentheses that means to fetch the entire record and nest it. You include a list of columns inside the parens, or asterisk to request all columns.

The embedding works for 1-N, N-1, and N-N relationships. That means you could also ask for a client and all their projects:

GET /clients?id=eq.42&select=id, name, projects{*}

In the examples above we asked for all columns in the embedded resource but the the select query is recursive. You could for instance specify

GET /foo?select=x, y, bar[z, w, baz{*}]

You can select not only using table names, but also foreign key column names! This is especially needed when you have a table with two foreign keys pointing to the same table, for example billing_address_id and shipping_address_id. To embed the same foreign key row from our client example earlier you could do the following:

GET /projects?id=eq.1&select=id, name, client_id{*}

In the response there will be a client_id object containing all the data for that row.

However, a client_id object doesn’t make a lot of sense, so you could do one of two things. Tell PostgREST that you want the key renamed by using the alias feature like so clientid:client_id{*}, or just try client{*}
in the select parameter! PostgREST supports smart ducktype checking for common foreign key names, so if your column name ends with \_id, \_fk, or any variation of the two (including camelcase) you can embed a row with just the name’s beginning.

So for a complete example:

```plaintext
GET /projects?id=eq.1&select=id, name, client{*}
```

Would embed in the client key the row referenced with client\_id.

The alias feature works for embedded entities and also for regular columns. This is useful in situations where for example you use different naming conventions in the database and front-end.

The following request will produce the output below:

```plaintext
GET /orders?id=eq.1&select=orderId:id, customer:customer_id{customerId:id, customerName:name}
```

```json
[
{
  "orderId": 1,
  "customer": {
    "customerId": 1,
    "customerName": "John Smith"
  }
}
]
```

If you want to apply filters to the embedded items, you can do that like so:

```plaintext
GET /clients?id=eq.42&select=id,name,projects{id,name,is_active}&projects.is_active=eq.true
```

The above request will return the client with id=42 and all the projects for that client that are still active

**Response Format**

Query responses default to JSON but you can get them in CSV as well. Just make your request with the header

```
Accept: text/csv
```

**Singular vs Plural**

Many APIs distinguish plural and singular resources, e.g./stories vs /stories/1. Why do we use /stories?id=eq.1? It is because a single 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 common urls come from a degenerate case of simple (and overwhelmingly numeric) primary keys often introduced automatically be Object Relational Mapping.

For consistency’s sake all these endpoints return a JSON array, /stories, /stories?genre=eq.mystery, /stories?id=eq.1. They are all filtering a bigger array. However you might want the last one to return a single JSON object, not an array with one element. To request a singular response send the header Prefer: plurality=singular.
Data Schema

As well as issuing a `GET /` to obtain a list of the tables, views, and stored procedures available, you can get more information about any particular endpoint.

```plaintext
OPTIONS /my_view
```

This will include the row names, their types, primary key information, and foreign keys for the given table or view.

CORS

PostgREST sets highly permissive cross origin resource sharing. It accepts Ajax requests from any domain.
CHAPTER 7

Updating Data

Record Creation

- Cannot be cached or prefetched
- Not idempotent

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 keys will be set to default values when applicable.

```plaintext
POST /table_name
{ "col1": "value1", "col2": "value2" }
```

The response will include a Location header describing where to find the new object. If you would like to get the full object back in the response to your request, include 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.

Record Update

To update a single existing record, you will need to make a PATCH request, and use filtering via parameters to specify which record to update.

```plaintext
PATCH /table_name?myPKey=eq.thePKey
{ "col1":"update-value1", "col2":"update-value2" }
```

Bulk Insertion

- Cannot be cached or prefetched
- Not idempotent

You can POST a JSON array or CSV to insert multiple rows in a single HTTP request. Note that using CSV requires less parsing on the server and is much faster.

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

```plaintext
POST /people
name,age,height
```
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.

Example of JSON bulk insert. Send an array:

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

If you would like to get the full object back in the response to your request, include the header `Prefer: return=representation`. Chances are you only want certain information back, though, like created ids. You can pass a `select` parameter to affect the shape of the response (further documented in the reading page). For instance

```sql
POST /people?select=id
[...]
```

returns something like

```json
[ { "id": 1 }, { "id": 2 } ]
```

# Multiple Tables Insertion or Update

The cleanest way to insert or update data into multiple tables using only one POST/PATCH request is to create a view that will join all target tables and present a single endpoint. In our example let's assume one users table and one companies table. In this case, we want a signup endpoint to create the first user within a company. And for this endpoint we want to insert with one request both user and company.

```sql
CREATE TABLE companies {
  id serial primary key,
  name text unique
};

CREATE TABLE users {
  id serial primary key,
  name text not null,
  pass text,
  company_id integer not null references companies
};
```

Having both tables created we create a view that joins them to be used as a /signup endpoint.

```sql
CREATE VIEW signup AS
SELECT  
  c.name AS company_name,
  u.name AS user_name,
  u.pass 
FROM    
  public.users u 
JOIN public.companies c ON c.id = u.company_id;
```
After the signup view creation, we can issue GET requests to read data from users and companies, but any attempt to POST or PATCH data will fail. PostgreSQL won’t allow any data change on views that have a JOIN clause in their FROM without a proper INSTEAD OF trigger. So in the example below we create a trigger to allow insertion of data in the signup view. The trigger is a simple PL/pgSQL function that first inserts into the companies table and uses the newly create company_id to create its first user.

```sql
CREATE FUNCTION signup()
RETURNS trigger
LANGUAGE plpgsql
AS $$
DECLARE
  vcompany_id int;
BEGIN
  INSERT INTO companies (name) VALUES (new.company_name) RETURNING id INTO vcompany_id;
  INSERT INTO users (name, pass, company_id) VALUES (new.user_name, new.pass, vcompany_id);
  RETURN new;
END;
$$;
CREATE TRIGGER signup
INSTEAD OF INSERT ON signup
FOR EACH ROW
EXECUTE PROCEDURE signup();
```

After the trigger creation we can issue a normal POST request to our signup endpoint:

```json
POST /signup
{ "company_name": "foo", "user_name": "bar" }
```

For an endpoint such as signup it's usually not desirable to have a PATCH route for updates, and we will skip this example for the sake of brevity. But it would be implemented in a very similar way to our POST example.

### Bulk Updates

- Cannot be cached or prefetched
- Not idempotent

To change parts of a resource or resources use the PATCH verb. For instance, here is how to mark all young people as children.

```json
PATCH /people?age=lt.13
{
  "person_type": "child"
}
```

This affects any rows matched by the url param filters, overwrites any fields specified in in the payload JSON and leaves the other fields unaffected. Note that although the payload is not in the JSON patch format specified by RFC6902, HTTP does not specify which patch format to use. Our format is more pleasant, meant for basic field replacements, and not at all “incorrect.”

### Deletion

- Cannot be cached or prefetched
- Idempotent
Simply use the DELETE verb. All records that match your filter will be removed. For instance deleting inactive users:

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

### Protecting Dangerous Actions

Notice that it is very easy to delete or update many records at once. In fact forgetting a filter will affect an entire table!
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.

We use JSON Web Tokens to authenticate API requests. As you’ll recall a JWT contains a list of cryptographically signed claims. PostgREST cares specifically about a claim called role. When 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. If the client included no (or an invalid) JWT then PostgREST selects the “anonymous role” which is specified by a command line arguments to the server on startup.

```
{
   "role": "jdoe123"
}
```

// Encoded as JWT with a secret of "secret" this becomes
// eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJyb2xlIjoiamRvZTEyMyJ9.X_ZeWSS9qsKDCDczv8C-GE2fccrPQjOh_ALM

Using JWT allows us to authenticate with external services. A login service needs merely to share a JWT encryption secret with the PostgREST server. The secret is also a server command line option.

It is even possible to generate JWT from inside a stored procedure in your database. Any SQL stored procedure that returns a type whose name ends in jwt_claims will have its return value encoded into JWT. See the User Management example for details.

Database Roles

Suppose you start the server like this:

```
postgrest postgres://foo@localhost:5432/mydb --anonymous anon
```

This means that foo is the so-called authenticator role and anon is the anonymous role. When a new HTTP request arrives at the server the latter is connected to the database as user foo. If no JWT is present, or if it is invalid, or if it does not contain the role claim then the server changes to the anonymous role with the query

```
SET LOCAL ROLE anon;
```

Otherwise it sets the role to that specified by JWT. For security your authenticator role should have access to nothing except the ability to become other users. Supposing you have three roles, one for anonymous users, one for authors, and another for the authenticator, you would set it up like this
CREATE ROLE authenticator NOINHERIT LOGIN;
CREATE ROLE anon;
CREATE ROLE author;

GRANT anon, author TO authenticator;

Row-Level Security

Simulated - PostgreSQL <9.5

Real - PostgreSQL >=9.5

Building Auth on top of JWT

Basic Auth

Github Sign-in

SSL
CHAPTER 9

API Versioning

Schema Search Path
Changing a Resource
Removing a Resource
Avoiding DB and Client Coupling
Data Migration

Sqitch

Test-Driven Migrations

Structural Tests

Value Tests with pgTAP
Heroku

Account setup on Heroku

Setting up a simple Heroku “Free Account” is easy and should take no more than a couple of minutes. For this guide I will be using a fictitious email account “postgrest@kismail.ru” to set up the account on Heroku. You can use whatever email account you desire. NOTE: During signup you will be asked to “Pick your primary development language”. It is safe here to choose any language or simply “I use another language”.

1. Heroku signup input

2. Heroku signup waiting
Thanks for signing up with Heroku! You must follow this link to activate your account:

https://id.heroku.com/account/accept/3818474/9ce5ce6374191364e8ee92eaf9f44427

Have fun, and don’t hesitate to contact us with your feedback.

The Heroku Team
https://heroku.com

Heroku is the cloud platform for rapid deployment and scaling of web applications. Get up and running in minutes, then deploy instantly via Git.

To learn more about Heroku and all its features, check out the Dev Center:
https://devcenter.heroku.com/articles/quickstart

3. Heroku signup confirmation

4. Heroku signup password
5. Heroku signup success

After initial signup, you will receive a confirmation email. You have to open the link provided in the email to activate your Heroku Free Account.

**New “App” setup in Heroku**

Creating an “App” can be seen as creating an environment, complete with hardware resources, to run any code you wish. It is here that the PostgreSQL database and the PostgREST instance will reside.

Select “New App” and then select a new name for the “App”. It can be anything as long as it is available. The “Run-time Selection” region can be anywhere, it only specifies where the app data is stored.
2. Heroku create new app.

3. Heroku app creation success

**Setup of PostgreSQL add-on in newly setup “App”**

At this point you should have your new “App” environment setup and running. It is now time to add a PostgreSQL database to your “App”. In your “App” screen, you will have several tabs with names such as “Overview”, “Resources”, “Deploy”, etc.

1. Choose “Resources”. This will show you the resources currently running in your “App”, which should be empty as you have yet to deploy any resources.

2. On the “Resources” page, under the header “Add-ons” you will find a search bar. Type in “PostgreSQL”, which will show you “Heroku PostgreSQL”. Click on the name to show a deployment
3. After deployment, you will see a line in your “Add-ons” named “Heroku PostgreSQL”. You have now added a PostgreSQL database to your “App”. Now to setup the database. NOTE: It might take a few minutes for Heroku to initialize the db, so please wait 1-2 minutes before moving on to step 4.
4. Click on the newly added “Heroku PostgreSQL” in your “Add-ons” and you will be taken to the database “Overview” page. Here you will find all the information necessary to find and connect to your PostgreSQL DB. This information will be very important when setting up PostgREST.

```
postgresqlguide:database::postgresql-emorphous-79539
```

Here are a variety of ways to connect to a Heroku PostgreSQL database. Find out how to do so via `sql`, `Java`, `Ruby`, `Python`, or `Node.js`. 

11.1. Heroku
### Statistics

<table>
<thead>
<tr>
<th>Resource name</th>
<th>postgresql-amorphous-79519</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plan</td>
<td>Hobby-dev</td>
</tr>
<tr>
<td>Status</td>
<td>Available</td>
</tr>
<tr>
<td>Primary</td>
<td>Yes</td>
</tr>
<tr>
<td>Connections</td>
<td>0/20</td>
</tr>
<tr>
<td>Data Size</td>
<td>7.1 MB</td>
</tr>
<tr>
<td>Tables</td>
<td>0</td>
</tr>
<tr>
<td>PG Version</td>
<td>9.5.2</td>
</tr>
<tr>
<td>Created</td>
<td>2016-06-04T11:12:00Z</td>
</tr>
<tr>
<td>Rollback</td>
<td>Unsupported</td>
</tr>
<tr>
<td>Rows</td>
<td>0/10000 (In compliance)</td>
</tr>
</tbody>
</table>
5. At this point, you have full access to the PostgreSQL database and can create tables, schemas, functions, etc. 

NOTE: The PostgREST setup will only work against a single Schema. This means that tables outside of the selected Schema will not be directly accessible via API calls.

**Testing of newly setup PostgreSQL**

Before we move on to deploying PostgREST, we could perform a quick test to see that our DB is up and running. This can be done in many different ways, one of which is the pgAdmin GUI. It is also possible to connect with psql for simple testing.

NOTE: When using pgAdmin to connect to the DB, you will see many other inaccessible DBs. Don’t be alarmed. Find your DB by the “Database” name provided in the Overview.

**Setup of PostgREST deployment**

Now it is time to deploy PostgREST. The PostgREST deployment on Heroku is not an “Add-on”, it is an “Element”. It can be found by opening the menu at the top right of the Heroku screen and selecting “Elements”. Search for “PostgREST” and find it under “Buttons”. Opening the “PostgREST” page, a “Deploy to Heroku” button will be present on the page. This button will lead to the initial setup and configuration page for the PostgREST deployment on your “App”.

1. Heroku find PostgREST 1
2. Heroku find PostgREST 2

PostgREST serves a fully RESTful API from any existing PostgreSQL database. It provides a cleaner, more standards-compliant, faster API than you are likely to write from scratch.

Demo postgrest.herokuapp.com | Read Docs | Watch Video

Try making requests to the live demo server with an HTTP client such as postman. The structure of the demo database is defined by begrifts/postgres-example. You can use it as inspiration for test-driven server migrations in your own projects.

Also try other tools in the PostgREST ecosystem like the rcmd/admin demo.
4. Heroku PostgREST install

Explanations for each field in the PostgREST setup page:

- **App Name** (optional) : The name chosen for the PostgREST deployment. NOTE: The name you choose will be the beginning of your specific API url. So if you choose “bugbunny”, your API url will be “bugbunny.herokuapp.com”.

- **Run-time Selection** : Simply select the region the deployment (basically data) will be stored.

- **BUILDPACK_URL** : Do not change this. This points to the pack that will be used for setup, which cannot be changed if you want a functioning PostgREST.

- **POSTGREST_VER** : Do not change this. Version of PostgREST used for installation.

PostgREST configuration (Information from the PostgreSQL Overview page):

- **DB_NAME** : Here you must provide the name of the database you setup earlier, which can be found on the “Overview” page of the PostgreSQL add-on. It should be something like “d41vontrqm2qo”.

- **AUTH_ROLE** : This is the role of an authenticated user. All authenticated calls to the API will use this user role when attempting to perform actions against the PostgreSQL database. User authentication is done using the procedure described on the Security page. For the purpose of this demo, you can simply input the data in the “User” field of the Overview page of the PostgreSQL add-on. It should be something like “nhvhzryphzmcp”.

- **AUTH_PASS** : Simply the password for the authenticated user, also found on the Overview page of the PostgreSQL add-on. It should be something like “5YDLXynT2ZWqUA6c2c_D12ToFA”.

- **ANONYMOUS_ROLE** : This is the role for any unauthenticated call to the API. In a regular PostgreSQL database this role could be for calls that for example don’t alter data, but simply retrieve data. Unfortunately in Heroku Hobby-dev account, it is not possible to add any roles to your PostgreSQL database, which means you will have to use the only role provided when PostgreSQL is setup, the admin role. So input the same data provided in the “AUTH_ROLE” field. Please keep in mind that this admin role has full access and can do anything, including deleting from the db.

- **DB_HOST** : This is the URL to the database, which should be copied directly from the Overview page of the PostgreSQL add-on.

- **DB_PORT** : The port to used when connecting to the database. Usually not necessary to change unless some specific manual changes have been made to the DB connection.

- **DB_POOL** : As the description states, “Maximum number of connections in database pool”. This parameter is not necessary to change.
• **JWT_SECRET**: Information regarding this parameter can be found in the Security page. For testing purposes, there is no need to change this parameter, but it is important to read and fully understand how JWT works if and when moving on from testing only.

• **SCHEMA**: This parameter is very important when it comes to being able to access the information on the PostgreSQL DB via PostgREST, as it indicates what Schema on the DB should be made available through the API calls. Only a single Schema can be selected and the tables and views in that Schema will be accessible through the API, so please make sure that the correct Schema name is input here.

When all the above fields are filled in, the PostgREST configuration should be ready for deployment. Pressing the “Deploy for free” button at the bottom of the screen will initiate the deployment process, which should take no more than a 1-2 minutes to finish.

When all the above fields are filled in, the PostgREST configuration should be ready for deployment. Pressing the “Deploy for free” button at the bottom of the screen will initiate the deployment process, which should take no more than a 1-2 minutes to finish.

---

## Testing the PostgREST deployment

At this point, you should have the PostgreSQL database and the PostgREST Heroku element up and running. Time to test the setup. The easiest way to see if things are up and running is to go to your “Personal Apps” in your Heroku dashboard, select the name you selected when deploying PostgREST (like “bugbunny”) and...
then pressing the “Open app” button at the top, right corner of the screen. What should open is a new web page, with a JSON representation of the tables in your PostgreSQL database. (https://bugbunny.herokuapp.com/)

```json
[
  - {
    "schema": "testing",
    "name": "people",
    "insertable": true
  }
]
```

An additional test to perform is to add the name of any table inside you database Schema to the end of the PostgREST Heroku app URL, such as `https://bugbunny.herokuapp.com/people`, where “people” is the name of the Table in the database Schema. This will retrieve, in JSON form, all the rows in the “People” table.

Heroku LINKS:
- Heroku PostgreSQL Plans
- Heroku PostgREST

Debian

(To be written)
[  
  - {  
      id: 1,  
      name: "Bruce ",  
      surname: "Scott ",  
      age: 65  
    },  
  - {  
      id: 2,  
      name: "John ",  
      surname: "Doe ",  
      age: 40  
    },  
  - {  
      id: 3,  
      name: "Jane ",  
      surname: "Doe ",  
      age: 35  
    }  
] 

Fig. 11.1: Heroku postgrest test 1
CHAPTER 12

Performance

Benchmarks
Caching
Quality of Service
Tips
Getting Started

Your First (simple) API

Let’s start with the simplest thing possible. We will expose some tables directly for reading and writing by anyone.

Start by making a database

createdb demo1

We’ll set it up with a film example (courtesy of Jonathan Harrington). Copy the following into your clipboard:

BEGIN;

CREATE TABLE director
(
  name text NOT NULL PRIMARY KEY
);

CREATE TABLE film
(
  id serial PRIMARY KEY,
  title text NOT NULL,
  year date NOT NULL,
  director text REFERENCES director (name)
    ON UPDATE CASCADE ON DELETE CASCADE,
  rating real NOT NULL DEFAULT 0,
  language text NOT NULL
);

CREATE TABLE festival
(
  name text NOT NULL PRIMARY KEY
);

CREATE TABLE competition
(
  id serial PRIMARY KEY,
  name text NOT NULL,
  festival text NOT NULL REFERENCES festival (name)
    ON UPDATE CASCADE ON DELETE CASCADE,
  year date NOT NULL
);

CREATE TABLE film_nomination


```sql
{
  id serial PRIMARY KEY,
  competition integer NOT NULL REFERENCES competition (id)
      ON UPDATE NO ACTION ON DELETE NO ACTION,
  film integer NOT NULL REFERENCES film (id)
      ON UPDATE CASCADE ON DELETE CASCADE,
  won boolean NOT NULL DEFAULT true
};

COMMIT;
```

Apply it to your new database by running

```
# On OS X
pbpaste | psql demo1

# Or Linux
# xclip -selection clipboard -o | psql dem01
```

Start the PostgREST server and point it at the new database. (See the installation instructions.)

```
postgrest postgres://postgres:@localhost:5432/demo1 -a postgres --schema public
```

## Populating Data

Let’s use PostgREST to populate the database. Install a REST client such as Postman. Now let’s insert some data as a bulk post in CSV format:

```
POST http://localhost:3000/festival
Content-Type: text/csv

name
Venice Film Festival
Cannes Film Festival
```

In Postman it will look like this

Notice that the post type is raw and that `Content-Type: text/csv` set in the Headers tab.

The server returns HTTP 201 Created. Because we inserted more than one item at once there is no `Location` header in the response. However sometimes you want to learn more about items which you just inserted. To have the server include the full results, include the header `Prefer: return=representation`.

At this point if you send a GET request to `/festival` it should return

```
[
  {
    "name": "Venice Film Festival"
  },
  {
    "name": "Cannes Film Festival"
  }
]
```

Now that you’ve seen how to do a bulk insert, let’s do some more and fully populate the database.

Post the following to `/competition`:

Now `/director`:  

```
And /film:

Finally /film_nomination:

## Getting and Embedding Data

First let’s review which films are stored in the database:

```
GET http://localhost:3000/film
```

It gives us back a list of JSON objects. What if we care only about the film titles? Use `select` to shape the output:

```
```

```
[
    {
        "title": "Chuang ru zhe"
    },
    {
        "title": "The Look of Silence"
    },
    {
        "title": "Fires on the Plain"
    },
    ...
]
```

Here is where it gets cool. PostgREST can embed objects in its response through foreign key relationships. Earlier we created a join table called `film_nomination`. It joins films and competitions. We can ask the server about the structure of this table:

```
OPTIONS http://localhost:3000/film_nomination
```
{"pkey": [
  "id"
],
"columns": [
  {
    "references": null,
    "default": "nextval('film_nomination_id_seq'::regclass)",
    "precision": 32,
    "updatable": true,
    "schema": "public",
    "name": "id",
    "type": "integer",
    "maxLen": null,
    "enum": [],
    "nullable": false,
    "position": 1
  },
  {
    "references": {
      "schema": "public",
      "column": "id",
      "table": "competition"
    },
    "default": null,
    "precision": 32,
    "updatable": true,
    "schema": "public",
    "name": "competition",
    "type": "integer",
    "maxLen": null,
    "enum": [],
    "nullable": false,
    "position": 2
  },
  {
    "references": {
      "schema": "public",
      "column": "id",
      "table": "film"
    },
    "default": null,
    "precision": 32,
    "updatable": true,
    "schema": "public",
    "name": "film",
    "type": "integer",
    "maxLen": null,
    "enum": [],
    "nullable": false,
    "position": 3
  },
  {"references": null,
   "default": "true",
   "precision": null,
   "updatable": true,
   "schema": "public",
From this you can see that the columns `film` and `competition` reference their eponymous tables. Let's ask the server for each film along with names of the competitions it entered. You don't have to do any custom coding. Send this query:

```
GET http://localhost:3000/film?select=title,competition{name}
```

```
[ 
  { 
    "title": "Chuang ru zhe", 
    "competition": [ 
    { 
      "name": "Golden Lion" 
    } 
  }, 
  { 
    "title": "The Look of Silence", 
    "competition": [ 
    { 
      "name": "Golden Lion" 
    } 
  }, 
  ... 
]
```

The relation flows both ways. Here is how to get the name of each competition’s name and the movies shown at it.

```
GET http://localhost:3000/competition?select=name,film{title}
```

```
[ 
  { 
    "name": "Golden Lion", 
    "film": [ 
    { 
      "title": "Chuang ru zhe" 
    }, 
    { 
      "title": "The Look of Silence" 
    }, 
    ... 
    ], 
  }, 
  { 
    "name": "Palme d'Or", 
    "film": [ 
    { 
      "title": "The Wonders" 
    }
```

13.3. Getting and Embedding Data
Why not learn about the directors too? There is a many-to-one relation directly between films and directors. We can alter our previous query to include directors in its results.

GET http://localhost:3000/competition?select=name,film{title,director[*]}

```
[
  {
    "name": "Golden Lion",
    "film": [
      {
        "title": "Manglehorn",
        "director": {
          "name": "David Gordon Green"
        }
      },
      {
        "title": "Belye nochi pochtalona Alekseya Tryapitsyna",
        "director": {
          "name": "Andrey Konchalovskiy"
        }
      },
      ...
    ]
  },
  ...
]
```

Singular Responses

How do we ask for a single film, for instance the second one we inserted?

GET http://localhost:3000/film?id=eq.2

It returns

```
[
  {
    "id": 2,
    "title": "The Look of Silence",
    "year": "2014-01-01",
    "director": "Joshua Oppenheimer",
    "rating": 8.3,
    "language": "Indonesian"
  }
]
```

Like any query, it gives us a result set, in this case an array with one element. However you and I know that id is a primary key, it will never return more than one result. We might want it returned as a JSON object, not an array. To express this preference include the header Prefer: plurality=singular. It will respond with
Conclusion

This tutorial showed how to create a database with a basic schema, run PostgREST, and interact with the API. The next tutorial will show how to enable security for a multi-tenant blogging API.
API clients authenticate with JSON Web Tokens. PostgREST does not support any other authentication mechanism directly, but they can be built on top. In this demo we will build a username and password system on top of JWT using only plpgsql.

Future examples such as the multi-tenant blogging platform will use the results from this example for their auth. We will build a system for users to sign up, log in, manage their accounts, and for admins to manage other people’s accounts. We will also see how to trigger outside events like sending password reset emails.

Before jumping into the code, a little more about how the tokens work. Every JWT contains cryptographically signed claims. PostgREST cares specifically about a claim called role. When a client includes a role claim PostgREST executes their request using that database role.

How would a client include a role claim, or claims in general? Without knowing the server JWT secret a client cannot create a claim. The only place to get a JWT is from the PostgREST server or from another service sharing the secret and acting on its behalf. We’ll use a stored procedure returning type jwt_claims which is a special type causing the server to encrypt and sign the return value.

Storing Users and Passwords

We create a database schema especially for auth information. We’ll also need the PostgreSQL extension pgcrypto.

```
create extension if not exists pgcrypto;

-- 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;
```

Next a table to store the mapping from usernames and passwords to database roles. The code below includes triggers and functions to encrypt the password and ensure the role exists.

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

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();

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 passwords. 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 $$
begin
return (select role from basic_auth.users
    where users.email = user_role.email
    and users.pass = crypt(user_role.pass, users.pass)
);
end;
$$;
Password Reset

When a user requests a password reset or signs up we create a token they will use later to prove their identity. The tokens go in this table.

```sql
drop type if exists token_type_enum cascade;
create type token_type_enum as enum ('validation', 'reset');
create table if not exists
  basic_auth.tokens 
  (token uuid primary key,
   token_type token_type_enum not null,
   email text not null references basic_auth.users (email)
     on delete cascade on update cascade,
   created_at timestamptz not null default current_date);
```

In the main schema (as opposed to the `basic_auth` schema) we expose a password reset request function. HTTP clients will call it. The function takes the email address of the user.

```sql
create or replace function
  request_password_reset(email text) returns void
  language plpgsql
  as $$
  declare
tok uuid;
bEGIN
  delete from basic_auth.tokens
  where token_type = 'reset'
  and tokens.email = request_password_reset.email;

  select gen_random_uuid() into tok;
  insert into basic_auth.tokens (token, token_type, email)
    values (tok, 'reset', request_password_reset.email);
  perform pg_notify('reset',
    json_build_object('email', request_password_reset.email,
      'token', tok,
      'token_type', 'reset'))::text
  end;
$$;
```

This function does not send any emails. It sends a PostgreSQL NOTIFY command. External programs such as a mailer listen for this event and do the work. The most robust way to process these signals is by pushing them onto work queues. Here are two programs to do that:

1. aweber/pgsql-listen-exchange for RabbitMQ
2. SpiderOak/skeeter for ZeroMQ

For experimentation you don’t need that though. Here’s a sample Node program that listens for the events and logs them to stdout.

```javascript
var PS = require('pg-pubsub');

if(process.argv.length !== 3) {
  console.log("USAGE: DB_URL");
}
```
process.exit(2);
}
var url = process.argv[2],
    ps = new PS(url);

// password reset request events
ps.addChannel('reset', console.log);
// email validation required event
ps.addChannel('validate', console.log);

// modify me to send emails

Once the user has a reset token they can use it as an argument to the password reset function, calling it through the PostgREST RPC interface.

crate or replace function
reset_password(email text, token uuid, pass text)
    returns void
language plpgsql
as $$
declare
tok uuid;
begin
    if exists(select 1 from basic_auth.tokens
        where tokens.email = reset_password.email
        and tokens.token = reset_password.token
        and token_type = 'reset') then
        update basic_auth.users set pass=reset_password.pass
        where users.email = reset_password.email;
    delete from basic_auth.tokens
        where tokens.email = reset_password.email
        and tokens.token = reset_password.token
        and token_type = 'reset';
    else
        raise invalid_password using message =
            'invalid user or token';
    end if;
    delete from basic_auth.tokens
        where token_type = 'reset'
        and tokens.email = reset_password.email;
    select gen_random_uuid() into tok;
    insert into basic_auth.tokens (token, token_type, email)
        values (tok, 'reset', reset_password.email);
    perform pg_notify('reset',
        json_build_object(
            'email', reset_password.email,
            'token', tok
        )::text
    );
end;
$$;
Email Validation

This is similar to password resets. Once again we generate a token. It differs in that there is a trigger to send validations when a new login is added to the users table.

```sql
create or replace function basic_auth.send_validation() returns trigger
    language plpgsql
    as $$
begin
    select gen_random_uuid() into tok;
    insert into basic_auth.tokens (token, token_type, email)
    values (tok, 'validation', new.email);
    perform pg_notify('validate',
        json_build_object('email', new.email,
                         'token', tok,
                         'token_type', 'validation'
        )::text
    );
    return new;
end
$$;

drop trigger if exists send_validation on basic_auth.users;
cREATE TRIGGER send_validation
    AFTER INSERT ON basic_auth.users
    FOR EACH ROW
    EXECUTE PROCEDURE basic_auth.send_validation();
```

Editing Own User

We’ll construct a redacted view for users. It hides passwords and shows only those users whose roles the currently logged in user has db permission to access.

```sql
create or replace view users as
select actual.role as role,
    '***':text as pass,
    actual.email as email,
    actual.verified as verified
from basic_auth.users as actual,
    (select rolname
    from pg_authid
    where pg_has_role(current_user, oid, 'member')
    ) as member_of
where actual.role = member_of.rolname;
-- can also add restriction that current_setting('postgrest.claims.email')
-- is equal to email so that user can only see themselves
```

Using this view clients can see themselves and any other users with the right db roles. This view does not yet support inserts or updates because not all the columns refer directly to underlying columns. Nor do we want it to be auto-updatable because it would allow an escalation of privileges. Someone could update their own row and change their role to become more powerful.
We’ll handle updates with a trigger, but we’ll need a helper function to prevent an escalation of privileges.

```sql
create or replace function
basic_auth.clearance_for_role(u name) returns void as
$$
declare
ok boolean;
begin
select exists (
    select rolname
    from pg_authid
    where pg_has_role(current_user, oid, 'member')
    and rolname = u
) into ok;
if not ok then
    raise invalid_password using message =
    'current user not member of role ' || u;
end if;
end
$$ LANGUAGE plpgsql;
```

With the above function we can now make a safe trigger to allow user updates.

```sql
create or replace function
update_users() returns trigger
language plpgsql
AS $$
begin
if tg_op = 'INSERT' then
    perform basic_auth.clearance_for_role(new.role);

    insert into basic_auth.users
      (role, pass, email, verified)
     values
      (new.role, new.pass, new.email,
       coalesce(new.verified, false));
    return new;
elsif tg_op = 'UPDATE' then
    -- no need to check clearance for old.role because
    -- an ineligible row would not have been available to update (http 404)
    perform basic_auth.clearance_for_role(new.role);

    update basic_auth.users set
      email = new.email,
      role = new.role,
      pass = new.pass,
      verified = coalesce(new.verified, old.verified, false)
    where email = old.email;
    return new;
elsif tg_op = 'DELETE' then
    -- no need to check clearance for old.role (see previous case)

    delete from basic_auth.users
    where basic_auth.email = old.email;
    return null;
end if;
end
$$;
```
drop trigger if exists update_users on users;
cREATE TRIGGER update_users
    instead of insert or update or delete on
    users for each row execute procedure update_users();

Finally add a public function people can use to sign up. You can hard code a default db role in it. It alters the underlying basic_auth.users so you can set whatever role you want without restriction.

cREATE OR REPLACE FUNCTION
  signup(email text, pass text) returns void
as $$
  insert into basic_auth.users (email, pass, role) values
    (signup.email, signup.pass, 'hardcoded-role-here');
$$ language sql;

Generating JWT

As mentioned at the start, clients authenticate with JWT. PostgREST has a special convention to allow your sql functions to return JWT. Any function that returns a type whose name ends in jwt_claims will have its return value encoded. For instance, let’s make a login function which consults our users table.

First create a return type:

drop type if exists basic_auth.jwt_claims cascade;
cREATE TYPE basic_auth.jwt_claims AS (role text, email text, exp integer);

The exp field is expiration time of the token expressed in seconds since the epoch. In the function below we choose to make the token valid for one hour.

cREATE OR REPLACE FUNCTION
  login(email text, pass text) returns basic_auth.jwt_claims
  language plpgsql
  as $$
  declare
    _role name;
    _verified boolean;
    _email text;
    result basic_auth.jwt_claims;
  begin
    -- check email and password
    select basic_auth.user_role(email, pass) into _role;
    if _role is null then
      raise invalid_password using message = 'invalid user or password';
    end if;
    -- check verified flag whether users
    -- have validated their emails
    _email := email;
    select verified from basic_auth.users as u where u.email=_email limit 1 into _verified;
    if not _verified then
      raise invalid_authorization_specification using message = 'user is not verified';
    end if;
    select _role as role, login.email as email,
    extract(epoch from now())::integer + 60*60 as exp
    into result;
    return result;
  end;
$$;
An API request to login would look like this.

```
POST /rpc/login

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

Response

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6ImZvb0BiYXIuY29tIiwicm9sZSI6ImF1dGhvciJ9.KHwYdK9dAMAg-MGCQXuDiFuvbmW-y8FjfYIcMrETnto"
}
```

Try decoding the token at jwt.io. (It was encoded with a secret of secret which is the default.) To use this token in a future API request include it in an Authorization request header.

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6ImZvb0BiYXIuY29tIiwicm9sZSI6ImF1dGhvciJ9.KHwYdK9dAMAg-MGCQXuDiFuvbmW-y8FjfYIcMrETnto
```

### Same-Role Users

You may not want a separate db role for every user. You can distinguish one user from another in SQL by examining the JWT claims which PostgREST makes available in the SQL variable `postgrest.claims`. Here's a function to get the email of the currently authenticated user.

```
-- Prevent current_setting('postgrest.claims.email') from raising
-- an exception if the setting is not present. Default it to ''.
ALTER DATABASE your_db_name SET postgrest.claims.email TO ' provisions;

cREATE OR REPLACE FUNCTION basic_auth.current_email() RETURNS TEXT
  LANGUAGE plpgsql
AS $$
BEGIN
  RETURN current_setting('postgrest.claims.email');
END;
$$;
```

Remember that the `login` function set the claims `email` and `role`. You can modify `login` to set other claims as well if they are useful for your other SQL functions to reference later.

### Permissions

Basic table-level permissions. We'll add an the authenticator role which can’t do anything itself other than switch into other roles as directed by JWT.

```
CREATE ROLE anon;
CREATE ROLE authenticator NOINHERIT;
GRANT anon TO authenticator;

GRANT USAGE ON SCHEMA public, basic_auth TO anon;
-- anon can create new logins
GRANT INSERT ON TABLE basic_auth.users, basic_auth.tokens TO anon;
GRANT SELECT ON TABLE pg_authid, basic_auth.users TO anon;
GRANT EXECUTE ON FUNCTION login(text, text),
```
request_password_reset(text),
reset_password(text, uuid, text),
signup(text, text)
to anon;

Conclusion

This section explained the implementation details for building a password based authentication system in pure sql. The next example will put it to work in a multi-tenant blogging API.
Multi-Tenant Blog

In our blog app there will be anonymous users and authors. Each author can create and edit their own posts, and read (but not edit) the posts of other authors. Anonymous users cannot edit anything but can sign up for author accounts. Authors can also post comments on articles.

This example builds off the previous User Management one. We had previously created a signup and login system on top of JWT. We’ll use this auth system for the blog. Run the SQL in the previous example first, before continuing with this example.

For your convenience, the complete sql for the blog demo is here. You can try it out in this vagrant image as well.

Adding Blog-Specific Tables

Storing the posts and comments is this simple. The comments do not form a tree, they are linear under a post.

```sql
create table if not exists
  posts (id bigserial primary key,
         title text not null,
         body text not null,
         author text not null references basic_auth.users (email)
            on delete restrict on update cascade
            default basic_auth.current_email(),
         created_at timestamptz not null default current_date);

create table if not exists
  comments (id bigserial primary key,
            body text not null,
            author text not null references basic_auth.users (email)
                on delete restrict on update cascade
                default basic_auth.current_email(),
            post bigint not null references posts (id)
                on delete cascade on update cascade,
            created_at timestamptz not null default current_date);
```
Permissions

On top of the authenticator and anon access granted in the previous example, blogs have an author role with extra permissions.

```sql
create role author;
grant author to authenticator;
grant usage on schema public, basic_auth to author;
grant select, insert, update, delete
  on basic_auth.tokens, basic_auth.users to author;
grant select, insert, update, delete
  on table users, posts, comments to author;
grant usage, select on sequence posts_id_seq, comments_id_seq to author;

-- authors can edit comments/posts
grant select, insert, update, delete
  on basic_auth.tokens, basic_auth.users to author;
grant select, insert, update, delete
  on table users, posts, comments to author;
grant usage, select on sequence posts_id_seq, comments_id_seq to author;
```

To ensure that authors cannot edit each others’ posts and comments we’ll use row-level security. Note that it requires PostgreSQL 9.5 or later.

```sql
grant select on posts, comments to anon;

-- Enable row-level security
ALTER TABLE posts ENABLE ROW LEVEL SECURITY;
ALTER TABLE comments ENABLE ROW LEVEL SECURITY;

drop policy if exists posts_select_unsecure on posts;
cREATE POLICY posts_select_unsecure on posts for select
  using (true);

drop policy if exists comments_select_unsecure on comments;
cREATE POLICY comments_select_unsecure on comments for select
  using (true);

drop policy if exists authors_eigencreate on posts;
cREATE POLICY authors_eigencreate on posts for insert
  with check (author = basic_auth.current_email());

drop policy if exists authors_eigencreate on comments;
cREATE POLICY authors_eigencreate on comments for insert
  with check (author = basic_auth.current_email());

drop policy if exists authors_eigenedit on posts;
cREATE POLICY authors_eigenedit on posts for update
  using (author = basic_auth.current_email())
  with check (author = basic_auth.current_email());

drop policy if exists authors_eigenedit on comments;
cREATE POLICY authors_eigenedit on comments for update
  using (author = basic_auth.current_email())
  with check (author = basic_auth.current_email());
```
Finally we need to modify the users view from the previous example. This is because all authors share a single db role. We could have chosen to assign a new role for every author (all inheriting from author) but we choose to tell them apart by their email addresses. The addition below prevents authors from seeing each others’ info in the users view.

```sql
create or replace view users as
select actual.role as role,
      '***':::text as pass,
      actual.email as email,
      actual.verified as verified
from basic_auth.users as actual,
     (select rolname
      from pg_authid
      where pg_has_role(current_user, oid, 'member')
     ) as member_of
where actual.role = member_of.rolname
  + and (actual.role <> 'author'
         + or email = basic_auth.current_email()
         + );
```

Example client queries

- **Top ten most recent posts**
  
  GET /posts?order=created_at.desc
  Range: 0–9

- **Single post (randomly chose id=1) with its comments**
  
  GET /posts?id=eq.1&select=*,comments{*}

- **Add a new post**
  
  POST /posts
  Authorization: Bearer [JWT TOKEN]

  ```
  
  "title": "My first post",
  "body": "Meh, forgot what I wanted to say."
  ```

Conclusion

Voilà, a blog API. Most of the code ended up being for defining security. Once you have set up an authentication system, the code to do application specific things like blog posts and comments is short. All the front-end routes and
verbs are created automatically for you.
API clients authenticate with JSON Web Tokens. PostgREST does not support any other authentication mechanism directly, but they can be built on top. In this demo we will build a system that works with an external authentication server and integrates with a PostgREST server by sharing the same JWT secret.

For a better understanding of JWT and PostgREST authentication system you should read the User Management example as well.

I’ll use a Rails application using Devise just to make the example more concrete, but this could be replicated for any other external authentication system using the same principles. In case Rails is not your cup of tea you can continue reading and just skip the Ruby code samples. I’ll also assume the use of JQuery for some client-side code samples for the sake of simplicity.

I won’t delve into Devise authentication details, for this would require a tutorial on its own, so I’m assuming that the reader’s authentication system is already working.

Sharing the JWT Secret

Allowing a third party to generate valid JWTs for your PostgREST API is just a matter of sharing a secret. So you need to give your authenticator software the same secret that was used in your API server under the --jwt-secret parameter.

This could be done easily using environment variables. You set a JWT_SECRET variable in the environment where you run your rails app and it will be accessible in the global variable ENV['JWT_SECRET'].

User Model

We will map each user in this example to two database roles. So our application users are either admin or customer. If they are just visitors (not logged in) to our website they will be anonymous. One way of mapping users is to add a field in our users table indicating their database role. I’ll add a text field called role to my users table:

```
ALTER TABLE users ADD role text NOT NULL DEFAULT 'customer';
```

Besides the main user that PostgREST uses to connect to PostgreSQL and the anonymous user, we will need two additional roles for our example:

- admin - to be used by users that access all the system rows.
- customer - to be used when user has restricted access to database rows.

Bellow we have the commands to create all roles that will be used:
CREATE USER authenticator NOINHERIT;
CREATE ROLE anonymous;
CREATE ROLE admin;
CREATE ROLE customer;

GRANT customer, admin, anonymous TO authenticator;

## Generating a JWT

Several libraries are available to generate JWT, you will find a very handy list in their website under Libraries. To continue our Rails example I’ll use the ruby library json_web_token.

In order to make the gem available in my Rails project I add the following line to my Gemfile:

```
gem 'json_web_token'
```

Then we create a Rails controller to serve JWTs for my authenticated users. For this I just open a file app/controllers/api_tokens_controller.rb with the content:

```ruby
class ApiTokensController < ApplicationController
  TOKEN_TTL = 1.hour

  def show
    unless ENV['JWT_SECRET'].present?
      return render json: {error: "you need to have JWT_SECRET configured to get an API token"}, status: 500
    end

    unless current_user.present?
      return render json: {error: "only authenticated users can request the API token"}, status: 401
    end

    expires_in TOKEN_TTL, public: false
    render json: {token: jwt}, status: 200
  end

  private
  def jwt
    JsonWebToken.sign(claims, key: ENV['JWT_SECRET'])
  end

  def claims
    # This token will expire 1 hour after being issued
    { role: current_user.role,
      user_id: current_user.id.to_s,
      exp: (Time.now + TOKEN_TTL).to_i
    }
  end
end
```

We also need to create a route in the config/routes.rb file:

```ruby
resource :api_token, only: [:show]
```

Now, any authenticated user in our rails application can request an api_token making a GET request to /api_token. This endpoint will return a json object with one property whose value is the token the API requests should use.
Orders Endpoint

Here is how to create a view to generate an endpoint /orders filtered by the logged in user:

```sql
ALTER DATABASE mydb SET postgrest.claims.user_id TO '';

CREATE OR REPLACE FUNCTION current_user_id()
RETURNS integer
STABLE
LANGUAGE SQL
AS $$
    SELECT nullif(current_setting('postgrest.claims.user_id'), '')::integer;
$$;

CREATE SCHEMA private;

CREATE TABLE private.orders (
    id serial primary key,
    user_id int references users,
    created_at timestamp not null default current_timestamp,
    updated_at timestamp not null default current_timestamp
);

CREATE VIEW orders AS
    SELECT id, user_id, created_at, updated_at
    FROM private.orders o
    WHERE current_user = 'admin' OR o.user_id = current_user_id();
```

Using the JWT

Now whenever you are authenticated in your Rails application you can use some Javascript code to get the token and use it:

```javascript
$.getJSON('/api_token').done(function(data){
    $.ajax('/orders', {'Authorization': 'Bearer ' + data.token}).done(function(data){
        console.log('Visible Orders: ', data);
    })
}).fail(function(){
    console.log('Error fetching API token');
})
```

We could also store the token to avoid having to fetch it again in the same page.

Conclusion

This section explained the implementation details for building an external authentication system working with PostgREST. With the previous User Management example this should give a clearer idea of how to set up authentication for your API.