2020 GSoC PostgreSQL Project Proposal

Performance Farm Benchmarks and Website Development

Contact Information

- Name: Pengyu Zhou (Eric Zhou)
- Email Address: ericzpy1998@outlook.com
- LinkedIn: Eric Pengyu Zhou

Personal Background

I finished my third year of electrical and computer engineering at Queen's University in Canada. In classes, I learned concepts in computer engineering and implemented them in small scale class projects. By the end of May last year, I had a chance to start my internship at IBM Canada Lab. The most helpful thing to me is to get familiar with a large scale, enterprise-level product. In development, I mainly worked on testing and web development. To me, learning computer science knowledge, or how to code overall, is to find a way of resolving real-world problems. Therefore, by the time I learnt about GSoC, I'm very excited as it is a good platform that can offer me a chance to resolve people's problems with other professionals.

Why interested

The first time I closely interacted with the open-source community was back in May 2019. A friend of mine worked on OpenJ9, she shared with me about the interesting moments during her work which inspired me to contribute as well. In Feb, My friends and I started a project that uses both the Django rest framework and PostgreSQL in the backend service. I did database related work and realized how different types of the database should be chosen, how the performance testing should be evaluated among kinds of designs, based on our business logic layer. This project experience gave me a chance to look deeply into PostgreSQL and made me more interested in contributing to the community. One day when I saw the pgperffarm project, I felt so excited that I explored the codebase, designed and implemented my unique feature (described in later sections).

Introduction

Since my proposal has a few large sections, I feel it would be helpful to give a brief summary of the structure:

- The **Overall Project Information** covers the description of the project from both users' and developers' perspective.
- The **My Implementation of a Feature with Demo** part where I showcase the feature I implemented during this period.
- Designs of the features that will be implemented soon are covered in **2020 GSoC Feature Design and Implementation.**
- The **Deliver with time table**.

Overall Project Information -- PgperfFarm

PgperfFarm intro and 2020 GSoC targets

PostgreSQL Performance Farm is a project that aims to collect and review performance data based on PgBench as changes are pushed to PostgreSQL. The whole infrastructure contains Vue.js as the frontEnd framework, python script to collect performance results, and DRF as the backend one to build an user-friendly website.

Until now, the website is hosted with a set of base APIs running on the server. Machines are displayed and two records are shown with links to the commits. In this proposal, the aim is to improve the usability, extendability and stability of the APIs, even that of the whole project as the APIs right now only cover a part of features.

Another main purpose is to hook the authentication module in the performance farm to the one used in postgresql.org so that the community login system can be enabled. Currently, only users that are registered in Django admin can be authenticated by the simpleJWT module, which is what's used right now. Based on the doc provided in pgweb repo[1], I would deliver a customized authentication module based on the existing one that would authenticate users in the central system.

Last but not least, being able to customize test scripts will improve the user experience to another level, also in this proposal, I will demonstrate an approach to achieve it.

Current technical stage/components

Here we have information about where we are at in this project from a technical perspective. Currently the project can be divided into three sections.

The essential part is the Client that fetches the config params from settings and then runs pre-defined scripts. With the finish of the test running, the result is then fed into the server side to process. Server side handles all kinds of responses, most part of its job is to run ORM operations that communicate with the database to store/get information; UI is now for displaying most of the time. **Fig.1** can help us better see the structure.

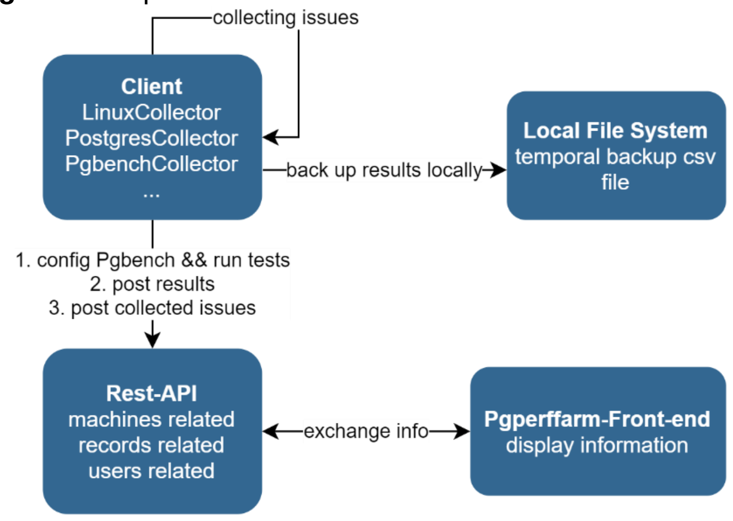


Fig.1 how UI, client and rest-api three parts communicate with each other.

Practically speaking, in the rest-api, the implementation is almost done in records, as well as the machine that contains basic post/get operations. Additionally, the users related operations are written based on the built-in Django auth module.

There are still some features and improvements that mainly focus on all three parts, which I will elaborate on in the following sections.

My Implementation of a Feature with Demo

Allowing custom tests to be added with configurable parameters

As we know the test case running on pgperffarm now is the built-in TPC-B script which covers all basic operations[2].

There can be two phases of this feature. The first is to let users configure pgbench params. an object based on the existing PGBENCH_CONFIG in client/settings.py can be created to include more potential params.

The second is to create a folder named ‘custom_scripts’ being put inside the client folder. Then by having a new type of collector called ‘scriptCollector’ containing the following methods in **Tbl.1**.

Method Name	**Definition**
has_script	check whether the dest folder has .sql files
run_custom_scripts	assemble a pgbench command that uses custom scripts to do tests and run it

Tbl.1 The basic methods needed in scriptCollector

A basic flow in the client side is shown in **Fig.2** below.

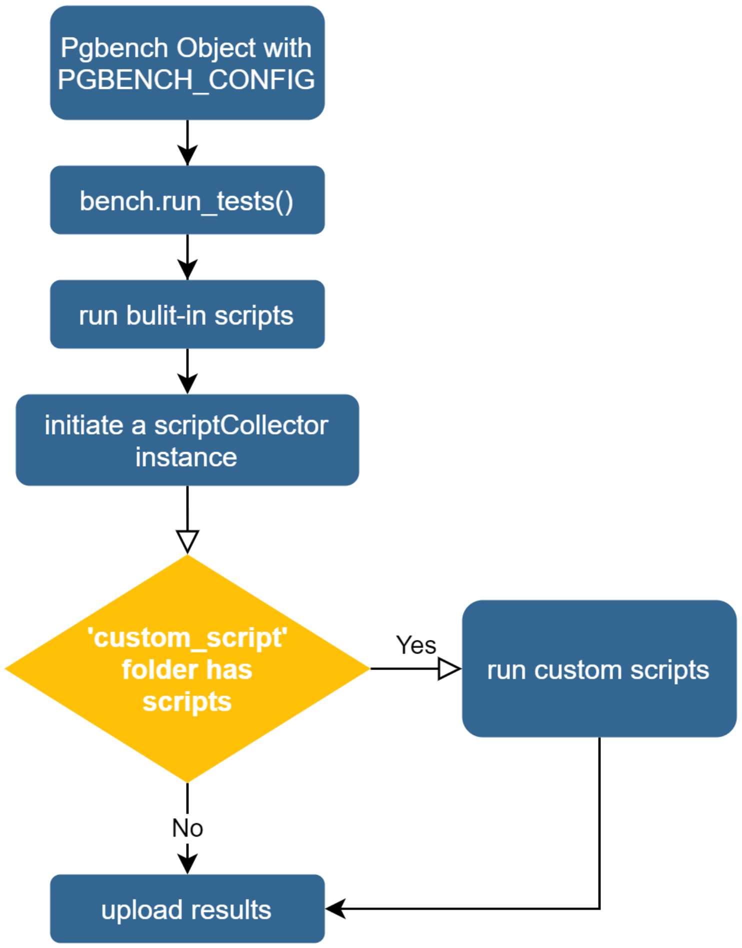

Pgbench Object with PGBENCH_CONFIG
bench.run_tests()
run bulit-in scripts
initiate a scriptCollector instance
'custom_script' folder has scripts
Yes
run custom scripts
No
upload results

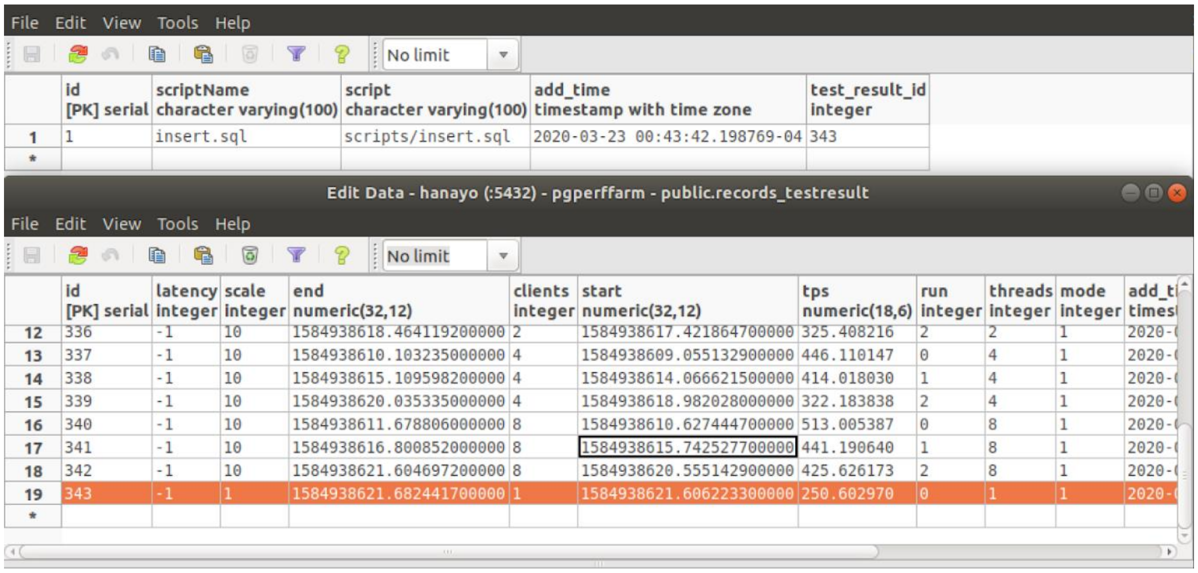
Fig.2 a general flow of getting results of custom scripts in the client.

On the server-side, a new model including script name and name should be created so that in the future, the custom scripts can be fetched by UI and displayed with the result. To do so, the new model named ‘‘records_customscripts’’ will have at least the following fields, as shown in **Tbl.2**:

Field Name	**Type**
TestResult	Foreign Key
script_name	CharField
script_content	FileField
add_date_time	DateTimeField

Tbl.2 The new model ‘records_customScripts’ containing fields details

Fig.3 is the uploaded result having the custom script. As you can see, ‘insert.sql’ is a custom script sitting in a newly created table named ‘records_testscripts’, its test_result_id is referenced in the already existing table ‘records_testresult’ . See highlighted in orange.



File Edit View Tools Help

	id [PK] serial	scriptName character varying(100)	script character varying(100)	add_time timestamp with time zone	test_result_id integer
1	1	insert.sql	scripts/insert.sql	2020-03-23 00:43:42.198769-04	343
*					

Edit Data - hanayo (:5432) - pgperffarm - public.records_testresult

File Edit View Tools Help

	id
	[PK] serial
		latency
		integer
			scale
			integer
				end
				numeric(32,12)
					clients
					integer
						start
						numeric(32,12)
							tps
							numeric(18,6)
								run
								integer
									threads
									integer
										mode
										integer
											add_ti
											times

12	336	-1	10	1584938618.464119200000	2	1584938617.421864700000	325.408216	2	2	1	2020-(
13	337	-1	10	1584938610.103235000000	4	1584938609.055132900000	446.110147	0	4	1	2020-(
14	338	-1	10	1584938615.109598200000	4	1584938614.066621500000	414.018030	1	4	1	2020-(
15	339	-1	10	1584938620.035335000000	4	1584938618.982028000000	322.183838	2	4	1	2020-(
16	340	-1	10	1584938611.678806000000	8	1584938610.627444700000	513.005387	0	8	1	2020-(
17	341	-1	10	1584938616.800852000000	8	1584938615.742527700000	441.190640	1	8	1	2020-(
18	342	-1	10	1584938621.604697200000	8	1584938620.555142900000	425.626173	2	8	1	2020-(
19	343	-1	1	1584938621.682441700000	1	1584938621.606223300000	250.602970	0	1	1	2020-(
*											

Fig.3 custom script is referenced records_testresult and stored in records_customscripts shown in pgadmin.

2020 Gsoc Feature Design and Implementation

PgperfFarm stability improvement

The collection down below is a subset of problems I encountered during the local distribution development. There are some other improvements in terms of code cleanness and readability.

Collection of issues

- There is no way to test uploading in the UI. which missed the key feature that should be supported in Pgperffarm.

- The handling of uploading results need extensive debugging since having a try-catch block for all operations is too general.

```
try:
        secret = request.META.get("HTTP_AUTHORIZATION")
        ret = Machine.objects.filter(machine_secret=secret, state='A').get()
        test_machine = ret.id
        if test_machine <= 0:
                raise TestDataUploadError("The machine is unavailable.")
```

Taking fetching a Machine object shown above as an example, if the operation throws an error of objects not found, the if statement wouldn't be invoked. Instead, the error would be caught by the catch block at the end of the method, which is a general one that prints status code and error msg of ‘object not found’. the point is that there are some similar object fetch operations existing in the try block, which makes it harder to to debug and spot the error.

- API_URL (http://<IP addr>/upload/) is put in the local client/settings_local.py for specifying the endpoint of uploading results. The hard coded url path is less efficient compared to having a method that dynamically combines the root url and target path.
- In the development, the test results were not stored in tables as the statisticals there are all -1. The reason for it is that a superuser role of postgres that matches the name of the user in OS should be created. However, the reason was not explicit and required proper debuggings. Either this should be fixed into a way of using the default user ‘postgres’, or explicitly ask developers to create a proper user in the contribution guide.
- Ideally, a temporal csv file containing results is stored in the file system every time when tests run. The reason is to backup the results so that it won’t get lost if the handling on the server side fails. In the codebase, the switch of whether or not collecting results is in settings, controlled by the constant *csv* in PGBENCH_CONFIG. However, the implementation of storing results in a csv file still can not be run through as the argument ‘mode’ is not implemented yet, which is also mentioned in the TODO comment in pgbench.py.

Plan to Address

The bullet points below follow the exact order of that in the preceding collection of Issue section. Each of them offers a way of either fixing the issue or creating a related feature.

- There should be a button vertically aligned with ‘upload a machine button’. The design reference is below. After clicking on it, a page with input fields should be displayed.
- Inside the original big try block, there should be small nested try blocks that catch each ORM operation so that errors can be reported precisely.
- Instead of writing request.post() inside upload_results method currently used in the repo:

```
def _upload_results(self, results):
       postdata = results
       post = []
       post.append(postdata)

       headers = {'Content-Type', 'application/json; charset=utf-8',
                  'Authorization: self._secret'}
       r = requests.post(self._url.encode('utf-8'),
                         data=json.dumps(post).encode('utf-8'), headers=headers)
```

The API invocation should be abstracted into a util method that receives headers, data, url_path as arguments. On top of that, each api call can be rewritten into a method and put together inside a ‘api-invocation’ module so that it is easier for developers to organize the api section. The interface can be in a format like this:

```
def _call_rest_api(self, headers, postdata):
       # The base_url is defined in the settings_local.py
       return requests.post(self._url.encode('utf-8'),
                            data=json.dumps(postdata).encode('utf-8'),headers=headers)
```

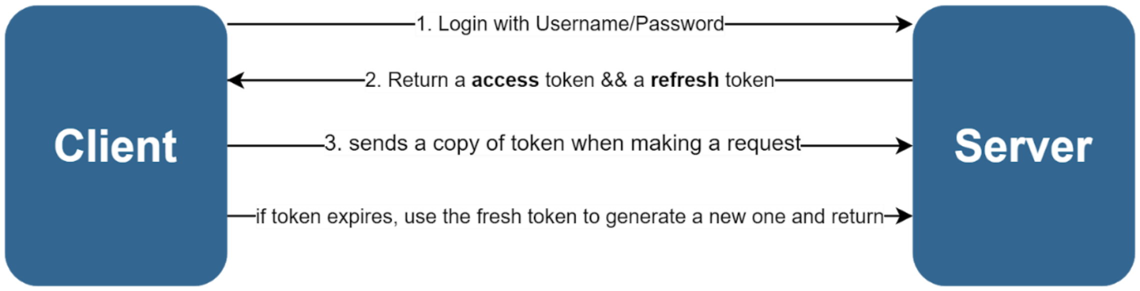
Then with the abstracted method, our upload result method can be then refactored into an example below, and the same rule applies to all the other methods that invokes rest api:

```
def _upload_results(self, results):
        path = 'upload/'
        postdata = results
        …
        headers = {'Content-Type', 'application/json; charset=utf-8', 'Authorization:
                          self._secret'}
        _call_rest_api(path, headers, postdata)
```

- Lastly, for collecting results in a csv file, the first job is to debug based on the current implementation as the basic methods are written already, for example, the method _ call _rest_api client/benchmarks/runner.py.

Authentication module of PgperfFarm

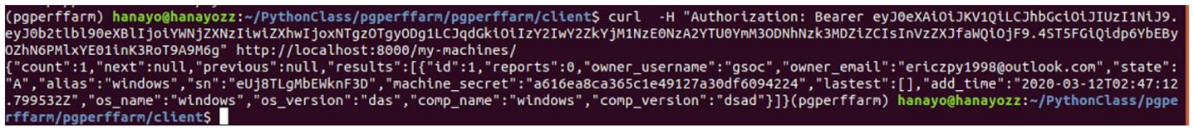
SimpleJWT -- Current stage



Client
Server
1. Login with Username/Password
2. Return a **access** token && a **refresh** token
3. sends a copy of token when making a request
if token expires, use the fresh token to generate a new one and return

Fig.4 a very basic flow of how JWT works

Fig.4 shows how the JWT works in general with token exchange. The login module in the project has not fully implemented this feature, yet we can still manually generate access token and refresh token to access limited information as shown in **Fig.5**. To make it work, token generation action should be integrated in the rest_api side and return tokens in response.



```
(pgperffarm) hanayo@hanayozz:~/PythonClass/pgperffarm/pgperffarm/client$ curl  -H "Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.
eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNTgzOTgyODg1LCJqdGkiOiIzY2IwY2ZkYjM1NzE0NzA2YTU0YmM3ODNhNzk3MDZiZCIsInVzZXJfaWQiOjF9.4ST5FGiQidp6YbEBy
OZhN6PMlxYE01inK3RoT9A9M6g" http://localhost:8000/my-machines/
{"count":1,"next":null,"previous":null,"results":[{"id":1,"reports":0,"owner_username":"gsoc","owner_email":"ericzpy1998@outlook.com","state":
"A","alias":"windows","sn":"eUj8TLgMbEWknF3D","machine_secret":"a616ea8ca365c1e49127a30df6094224","lastest":[],"add_time":"2020-03-12T02:47:12
.799532Z","os_name":"windows","os_version":"das","comp_name":"windows","comp_version":"dsad"}]}(pgperffarm) hanayo@hanayozz:~/PythonClass/pgpe
rffarm/pgperffarm/client$
```

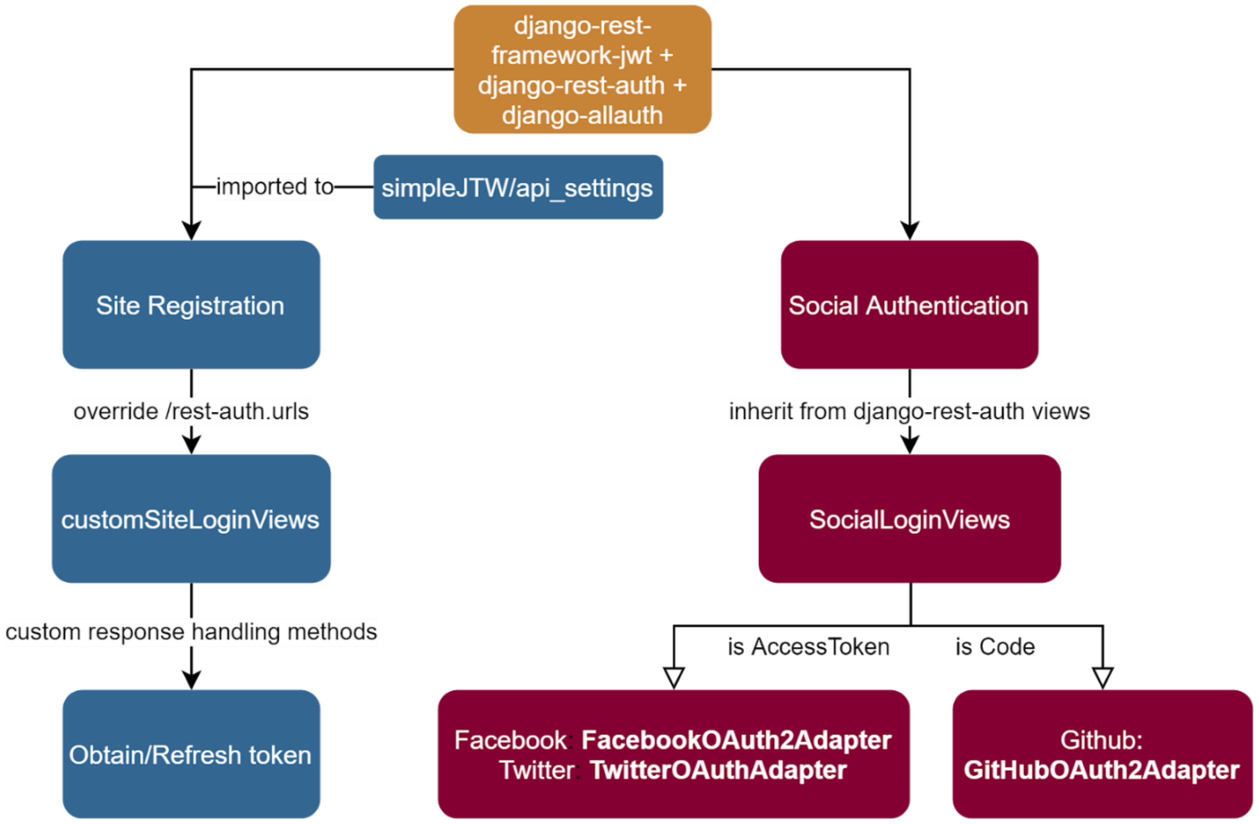
Fig.5 using the token generated by SimpleJWT module to access information that requires login permission.

Admin Registration/Login && Social Authentication -- Future Stage

There will be two apps getting involved with the feature, *Django-rest-framework-simplejwt* and *Django-rest-auth*. Several factors are being considered here such as Blacklist, support for refresh/access token, user registration with activation, social media authentication, etc.

In Django-rest-auth, there are views that can be modified to accommodate to return the JWT token in the response: LoginView/SocialLoginView/LogoutView. We can inherit and customize these view classes with our own implementation. For example, a class JWTSocialLoginView can be created that is inherited from SocialLoginView, by overriding the get_response method with a one that returns a token.

The overall flow looks like this:



django-rest-framework-jwt + django-rest-auth + django-allauth
imported to
simpleJTW/api_settings
Site Registration
Social Authentication
override /rest-auth.urls
inherit from django-rest-auth views
customSiteLoginViews
SocialLoginViews
custom response handling methods
is AccessToken
is Code
Obtain/Refresh token
Facebook FacebookOAuth2Adapter
Twitter TwitterOAuthAdapter
Github: GitHubOAuth2Adapter

Fig.6 an overall flow of how the auth module will be in the design.

The detailed Interface can look like the following:

```
# override the /rest-auth/login/ path with the login view from django-rest-framework-simplejwt
urlpatterns = [
       …
       url(r'^rest-auth/login/$', TokenLoginView.as_view(), name='token_obtain_pair'),
       url(r'^rest-auth/', include('rest_auth.urls')),
]
# override SocialLoginView and its get_response method as mentioned precedingly.
from rest_framework_simplejwt.views import TokenObtainPairView
rom django.contrib.auth.views import LoginView

# defined outside the TokenLoginView to work as a util method.
def get_jwt_token(self, login_user):
       # generate and return both access token and refresh token for a user.

class TokenLoginView(LoginView):
       def post(self):
              # login user and return both access token and refresh token in response.
              login()
```

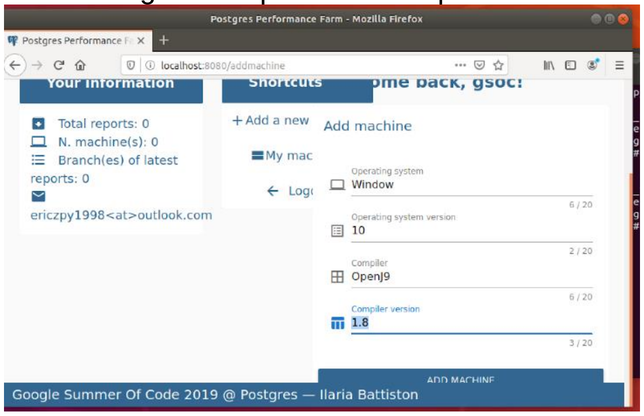
```
            return get_reponse();

    def login(self):
            # check if the request header contains a valid token
            # if YES, login the user
            login_after();
            # if NO, create a pair for the user.
            get_jwt_token(login_user)
            login_after();

    def get_response(self):
            # get the token and return it.
```

Or there is a newly forked repo named dj-rest-auth[3] as well from Django-rest-auth. It was recently opened and replaced the rest_framwwork_jwt with django_rest-framework-simplejwt[4]. Both ways are worth trying and seem promising, but focusing on the social auth should be placed at first considering username/password login is easier to be compromised compared to it when facing web attacks.

Last but not least, integrating the community sign-in feature is part of a long-term plan down the road. To start it, a conversion should be engaged with the PgInfra team.

Responsive web design

The **Fig.7** below is an example of how UI components overlap with each other when starting to shrink window size. By making a website responsive, a Flexbox layout[5] can be applied with a different design that can be leveraged to improve user experience.



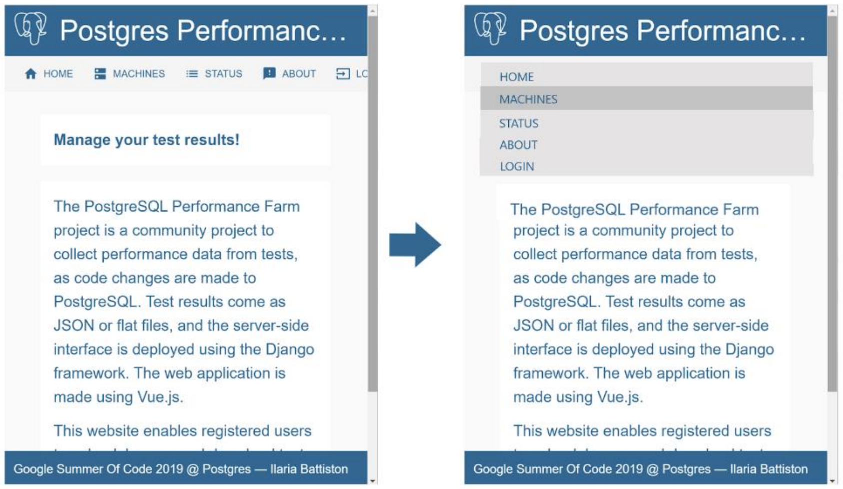
Total reports: 0
N. machine(s): 0
Branch(es) of latest reports: 0
ericzpy1998<at>outlook.com

Operating system: Window 6 / 20
Operating system version: 10 2 / 20
Compiler: OpenJ9 6 / 20
Compiler version: 1.8 3 / 20

Google Summer Of Code 2019 @ Postgres — Ilaria Battiston

Fig.7 the user page after login is not responsive to the window change

Taking the tabs as an example shown in **Fig.8**, a long dropdown menu can be rendered instead when the width of the window is getting too small.



Manage your test results!

The PostgreSQL Performance Farm project is a community project to collect performance data from tests, as code changes are made to PostgreSQL. Test results come as JSON or flat files, and the server-side interface is deployed using the Django framework. The web application is made using Vue.js.

This website enables registered users

Google Summer Of Code 2019 @ Postgres — Ilaria Battiston

The PostgreSQL Performance Farm project is a community project to collect performance data from tests, as code changes are made to PostgreSQL. Test results come as JSON or flat files, and the server-side interface is deployed using the Django framework. The web application is made using Vue.js.

This website enables registered users

Google Summer Of Code 2019 @ Postgres — Ilaria Battiston

Fig.8 a comparison between the original design (left) and the new one (right). The icons should be kept in the right dropdown menu as well.

An usual way of starting this is to use *@media* to apply certain CSS rules when meeting the condition specified, for example, when the width is smaller than a predefined size.

Migration from Django 1.11 to Django 2.2

Last but not least, the support of Django 1.11, which is what we are using right now, will be stopped by the first quarter in 2020. An upgrade to 2.2 can keep the support being offered until April 2022.
Some key new features that were introduced will be the plan-to-do[6][7]. After doing all steps layout below, we install Django 2.2, which the 1.11 version will be uninstalled automatically.

- **is_authenticated()** is now **is_authenticated**, which the code snippet below needs to be modified to is_authenticated.

```
def logout(self, results):
        if request.user.is_authenticated():
                django_logout(request)
        return HttpResponseRedirect("%slogout/" % settings.PGAUTH_REDIRECT)
```

- The **on_delete** argument for **ForeignKey** and **OneToOneField** is now required in models and migrations. as shown in **Fig.11**, ForeignKeys in the models of the record that don't have on_delete argument are the examples that should be changed:

```
class TestDataSet(models.Model):
        test_record = models.ForeignKey(TestRecord, verbose_name="test record id",
                                        help_text="test record id")
        test_cate = models.ForeignKey(TestCategory, verbose_name="test cate id",
                                     help_text="test cate id")
```

- The change in the **URL pattern system**. Taking the comparison below as an example:

```
url(r'^articles/(?P<year>[0-9]{4})/$', views.year_archive)
```

could be written as:

```
path(r'^articles/<int:year>/$', views.year_archive)
```

One thing to be noticed that the rewritten one is less constrained compared to the first one because the regular expression not only requires integer type but also the number of digits.

Deliverables with Time Table

Date Period	Duration	Goal
Before May 6	~	• Review the local contribution and refactor the code based on feedback.
May 6 - May 12	1 week	• Discuss with mentors about the opened PR for the feature of enabling custom testing scripts, and submit code.
May 13 - May 20	1 week	• Finalize the migration plan from 1.1 ITS to 2.2 LTS.
May 21 - May 27	1 week	• Implement the migration.
May 27 - June 3	1 week	• Finalize the solutions for ○ how to handle the ORM operation properly ○ how to dynamically generate the URL path to improve reusability and maintainability.
June 3 - June 17	2 weeks	• Work on refactoring views mentioned in the design. • Work on building the structure of generating API endpoints.
June 17 - June 29	2 weeks	• Send reviews and optimize the code with feedback (I intentionally give it a longer time as it mostly depends on the overall architecture of the project.).
Evaulation June 29 - July 3	5 days	• Submit evaluations.
July 3 - July 9	1 week	• Finalize the design of the authentication module, discuss with mentors about its validity.

July 10 - July 23	2 weeks	• Work on the social authentication module first, and then the site registration/login.
July 23 - July 29	1 week	• Send reviews and fix the code with feedback from mentors.
Evaulation July 27 - July 31	5 days	• Submit second evaluations
July 31 - Aug 6	1 week	• Finalize the design of UI including responsive web design and the new layout of user page containing a button for upload results
Aug 7 - Aug 19	2 weeks	• Implement the logic between the UI component and the client. • Implement the new design layout for responsive web design.
Aug 20 - Aug 28	1 week	• Submit code based on reviews, project summaries, and final evaluation

Reference

[1] Community authentication 2.0. https://github.com/postgres/pgweb/blob/master/docs/authentication.rst
[2] Pgbench Documentation. https://www.postgresql.org/docs/11/pgbench.html
[3] Dj-Rest-Auth. https://github.com/jazzband/dj-rest-auth
[4] Pull Request for replacing rest_framework_jwt. https://github.com/jazzband/dj-rest-auth/pull/3/
[5] FlexBox guide. https://css-tricks.com/snippets/css/a-guide-to-flexbox/
[6] Django 2.0 release notes. https://docs.djangoproject.com/en/3.0/releases/2.0/
[7] Django 2.2 release noteshttps://docs.djangoproject.com/en/3.0/releases/2.2/