tenantschemaDocumentation

Release dev

Bernardo Pires Carneiro

Jun 05, 2017

Contents

This application enables Django powered websites to have multiple tenants via PostgreSQL schemas. A vital feature for every Software-as-a-Service website.

Django provides currently no simple way to support multiple tenants using the same project instance, even when only the data is different. Because we don't want you running many copies of your project, you'll be able to have:

- Multiple customers running on the same instance
- Shared and Tenant-Specific data
- Tenant View-Routing

CHAPTER 1

What are schemas?

A schema can be seen as a directory in an operating system, each directory (schema) with it's own set of files (tables and objects). This allows the same table name and objects to be used in different schemas without conflict. For an accurate description on schemas, see PostgreSQL's official documentation on schemas.

CHAPTER 2

Why schemas?

There are typically three solutions for solving the multitenancy problem.

1. Isolated Approach: Separate Databases. Each tenant has it's own database.
2. Semi Isolated Approach: Shared Database, Separate Schemas. One database for all tenants, but one schema per tenant.
3. Shared Approach: Shared Database, Shared Schema. All tenants share the same database and schema. There is a main tenant-table, where all other tables have a foreign key pointing to.

This application implements the second approach, which in our opinion, represents the ideal compromise between simplicity and performance.

- Simplicity: barely make any changes to your current code to support multitenancy. Plus, you only manage one database.
- Performance: make use of shared connections, buffers and memory.

Each solution has it's up and down sides, for a more in-depth discussion, see Microsoft's excellent article on Multi-Tenant Data Architecture.

CHAPTER 3

How it works

Tenants are identified via their host name (i.e tenant.domain.com). This information is stored on a table on the `public` schema. Whenever a request is made, the host name is used to match a tenant in the database. If there's a match, the search path is updated to use this tenant's schema. So from now on all queries will take place at the tenant's schema. For example, suppose you have a tenant `customer` at http://customer.example.com. Any request incoming at `customer.example.com` will automatically use `customer`'s schema and make the tenant available at the request. If no tenant is found, a 404 error is raised. This also means you should have a tenant for your main domain, typically using the `public` schema. For more information please read the [setup](#setup) section.

CHAPTER 4

Shared and Tenant-Specific Applications

Tenant-Specific Applications

Most of your applications are probably tenant-specific, that is, its data is not to be shared with any of the other tenants.

Shared Applications

An application is considered to be shared when its tables are in the `public` schema. Some apps make sense being shared. Suppose you have some sort of public data set, for example, a table containing census data. You want every tenant to be able to query it. This application enables shared apps by always adding the `public` schema to the search path, making these apps also always available.

CHAPTER 5

Contents

Installation

Assuming you have django installed, the first step is to install `django-tenant-schemas`.

```
pip install django-tenant-schemas
```

Basic Settings

You'll have to make the following modifications to your `settings.py` file.

Your `DATABASE_ENGINE` setting needs to be changed to

```
DATABASES = {
    'default': {
        'ENGINE': 'tenant_schemas.postgresql_backend',
        # ..
    }
}
```

Add *tenant_schemas.routers.TenantSyncRouter* to your *DATABASE_ROUTERS* setting, so that the correct apps can be synced, depending on what's being synced (shared or tenant).

```
DATABASE_ROUTERS = (
    'tenant_schemas.routers.TenantSyncRouter',
)
```

Add the middleware `tenant_schemas.middleware.TenantMiddleware` to the top of `MIDDLEWARE_CLASSES`, so that each request can be set to use the correct schema.

If the hostname in the request does not match a valid tenant `domain_url`, a HTTP 404 Not Found will be returned.

If you'd like to raise `DisallowedHost` and a HTTP 400 response instead, use the `tenant_schemas.middleware.SuspiciousTenantMiddleware`.

If you'd like to serve the public tenant for unrecognised hostnames instead, use `tenant_schemas.middleware.DefaultTenantMiddleware`. To use a tenant other than the public tenant, create a subclass and register it instead.

If you'd like a different tenant selection technique (e.g. using an HTTP Header), you can define a custom middleware. See *Advanced Usage*.

```
from tenant_schemas.middleware import DefaultTenantMiddleware

class MyDefaultTenantMiddleware(DefaultTenantMiddleware):
    DEFAULT_SCHEMA_NAME = 'default'
```

```
MIDDLEWARE_CLASSES = (
    'tenant_schemas.middleware.TenantMiddleware',
    # 'tenant_schemas.middleware.SuspiciousTenantMiddleware',
    # 'tenant_schemas.middleware.DefaultTenantMiddleware',
    # 'myproject.middleware.MyDefaultTenantMiddleware',
    #...
)
```

```
TEMPLATES = [
    {
        'BACKEND': # ...
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                # ...
                'django.template.context_processors.request',
                # ...
            ]
        }
    }
]
```

```
TEMPLATE_CONTEXT_PROCESSORS = (
    'django.core.context_processors.request',
    #...
)
```

The Tenant Model

Now we have to create your tenant model. Your tenant model can contain whichever fields you want, however, you **must** inherit from `TenantMixin`. This Mixin only has two fields (`domain_url` and `schema_name`) and both are required. Here's an example, suppose we have an app named `customers` and we want to create a model called `Client`.

```
from django.db import models
from tenant_schemas.models import TenantMixin

class Client(TenantMixin):
    name = models.CharField(max_length=100)
    paid_until =  models.DateField()
    on_trial = models.BooleanField()
    created_on = models.DateField(auto_now_add=True)
```

```
    # default true, schema will be automatically created and synced when it is saved
    auto_create_schema = True
```

Before creating the migrations, we must configure a few specific settings.

Configure Tenant and Shared Applications

To make use of shared and tenant-specific applications, there are two settings called `SHARED_APPS` and `TENANT_APPS`. `SHARED_APPS` is a tuple of strings just like `INSTALLED_APPS` and should contain all apps that you want to be synced to `public`. If `SHARED_APPS` is set, then these are the only apps that will be synced to your `public` schema! The same applies for `TENANT_APPS`, it expects a tuple of strings where each string is an app. If set, only those applications will be synced to all your tenants. Here's a sample setting

```
SHARED_APPS = (
    'tenant_schemas',  # mandatory, should always be before any django app
    'customers', # you must list the app where your tenant model resides in

    'django.contrib.contenttypes',

    # everything below here is optional
    'django.contrib.auth',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.admin',
)

TENANT_APPS = (
    'django.contrib.contenttypes',

    # your tenant-specific apps
    'myapp.hotels',
    'myapp.houses',
)

INSTALLED_APPS = (
    'tenant_schemas',  # mandatory, should always be before any django app

    'customers',
    'django.contrib.contenttypes',
    'django.contrib.auth',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.admin',
    'myapp.hotels',
    'myapp.houses',
)
```

You also have to set where your tenant model is.

```
TENANT_MODEL = "customers.Client" # app.Model
```

Now you must create your app migrations for `customers`:

```
python manage.py makemigrations customers
```

The command `migrate_schemas --shared` will create the shared apps on the `public` schema. Note: your database should be empty if this is the first time you're running this command.

```
python manage.py migrate_schemas --shared
```

Warning: Never use `migrate` as it would sync *all* your apps to `public`!

Lastly, you need to create a tenant whose schema is `public` and it's address is your domain URL. Please see the section on *use*.

You can also specify extra schemas that should be visible to all queries using `PG_EXTRA_SEARCH_PATHS` setting.

```
PG_EXTRA_SEARCH_PATHS = ['extensions']
```

`PG_EXTRA_SEARCH_PATHS` should be a list of schemas you want to make visible globally.

Tip: You can create a dedicated schema to hold postgresql extensions and make it available globally. This helps avoid issues caused by hiding the public schema from queries.

Optional Settings

PUBLIC_SCHEMA_NAME

Default `'public'`

The schema name that will be treated as `public`, that is, where the `SHARED_APPS` will be created.

Tenant View-Routing

PUBLIC_SCHEMA_URLCONF

Default `None`

We have a goodie called `PUBLIC_SCHEMA_URLCONF`. Suppose you have your main website at `example.com` and a customer at `customer.example.com`. You probably want your user to be routed to different views when someone requests `http://example.com/` and `http://customer.example.com/`. Because django only uses the string after the host name, this would be impossible, both would call the view at `/`. This is where `PUBLIC_SCHEMA_URLCONF` comes in handy. If set, when the `public` schema is being requested, the value of this variable will be used instead of ROOT_URLCONF. So for example, if you have

```
PUBLIC_SCHEMA_URLCONF = 'myproject.urls_public'
```

When requesting the view `/login/` from the public tenant (your main website), it will search for this path on `PUBLIC_SCHEMA_URLCONF` instead of `ROOT_URLCONF`.

Separate projects for the main website and tenants (optional)

In some cases using the `PUBLIC_SCHEMA_URLCONF` can be difficult. For example, Django CMS takes some control over the default Django URL routing by using middlewares that do not play well with the tenants. Another example

would be when some apps on the main website need different settings than the tenants website. In these cases it is much simpler if you just run the main website *example.com* as a separate application.

If your projects are ran using a WSGI configuration, this can be done by creating a filed called `wsgi_main_website.py` in the same folder as `wsgi.py`.

```
# wsgi_main_website.py
import os
os.environ.setdefault("DJANGO_SETTINGS_MODULE", "project.settings_public")

from django.core.wsgi import get_wsgi_application
application = get_wsgi_application()
```

If you put this in the same Django project, you can make a new `settings_public.py` which points to a different `urls_public.py`. This has the advantage that you can use the same apps that you use for your tenant websites.

Or you can create a completely separate project for the main website.

Caching

To enable tenant aware caching you can set the KEY_FUNCTION setting to use the provided `make_key` helper function which adds the tenants `schema_name` as the first key prefix.

```
CACHES = {
    "default": {
        ...
        'KEY_FUNCTION': 'tenant_schemas.cache.make_key',
        'REVERSE_KEY_FUNCTION': 'tenant_schemas.cache.reverse_key',
    },
}
```

The `REVERSE_KEY_FUNCTION` setting is only required if you are using the django-redis cache backend.

Configuring your Apache Server (optional)

Here's how you can configure your Apache server to route all subdomains to your django project so you don't have to setup any subdomains manually.

```
<VirtualHost 127.0.0.1:8080>
    ServerName mywebsite.com
    ServerAlias *.mywebsite.com mywebsite.com
    WSGIScriptAlias / "/path/to/django/scripts/mywebsite.wsgi"
</VirtualHost>
```

Django's Deployment with Apache and mod_wsgi might interest you too.

Building Documentation

Documentation is available in `docs` and can be built into a number of formats using Sphinx. To get started

```
pip install Sphinx
cd docs
make html
```

This creates the documentation in HTML format at `docs/_build/html`.

Using django-tenant-schemas

Supported versions

You can use `django-tenant-schemas` with currently maintained versions of Django – see the Django's release process and the present list of Supported Versions.

It is necessary to use a PostgreSQL database. `django-tenant-schemas` will ensure compatibility with the minimum required version of the latest Django release. At this time that is PostgreSQL 9.3, the minimum for Django 1.11.

Creating a Tenant

Creating a tenant works just like any other model in Django. The first thing we should do is to create the `public` tenant to make our main website available. We'll use the previous model we defined for `Client`.

```
from customers.models import Client

# create your public tenant
tenant = Client(domain_url='my-domain.com', # don't add your port or www here! on a␣
↪local server you'll want to use localhost here
                schema_name='public',
                name='Schemas Inc.',
                paid_until='2016-12-05',
                on_trial=False)
tenant.save()
```

Now we can create our first real tenant.

```
from customers.models import Client

# create your first real tenant
tenant = Client(domain_url='tenant.my-domain.com', # don't add your port or www here!
                schema_name='tenant1',
                name='Fonzy Tenant',
                paid_until='2014-12-05',
                on_trial=True)
tenant.save() # migrate_schemas automatically called, your tenant is ready to be used!
```

Because you have the tenant middleware installed, any request made to `tenant.my-domain.com` will now automatically set your PostgreSQL's `search_path` to `tenant1, public`, making shared apps available too. The tenant will be made available at `request.tenant`. By the way, the current schema is also available at `connection.schema_name`, which is useful, for example, if you want to hook to any of django's signals.

Any call to the methods `filter`, `get`, `save`, `delete` or any other function involving a database connection will now be done at the tenant's schema, so you shouldn't need to change anything at your views.

Management commands

By default, base commands run on the public tenant but you can also own commands that run on a specific tenant by inheriting `BaseTenantCommand`.

For example, if you have the following `do_foo` command in the `foo` app:

`foo/management/commands/do_foo.py`

```
from django.core.management.base import BaseCommand

class Command(BaseCommand):
    def handle(self, *args, **options):
        do_foo()
```

You could create a wrapper command by using `BaseTenantCommand`:

`foo/management/commands/tenant_do_foo.py`

```
from tenant_schemas.management.commands import BaseTenantCommand

class Command(BaseTenantCommand):
    COMMAND_NAME = 'do_foo'
```

To run the command on a particular schema, there is an optional argument called `--schema`.

```
./manage.py tenant_command do_foo --schema=customer1
```

If you omit the `schema` argument, the interactive shell will ask you to select one.

migrate_schemas

`migrate_schemas` is the most important command on this app. The way it works is that it calls Django's `migrate` in two different ways. First, it calls `migrate` for the `public` schema, only syncing the shared apps. Then it runs `migrate` for every tenant in the database, this time only syncing the tenant apps.

Warning: You should never directly call `migrate`. We perform some magic in order to make `migrate` only migrate the appropriate apps.

```
./manage.py migrate_schemas
```

The options given to `migrate_schemas` are also passed to every `migrate`. Hence you may find handy

```
./manage.py migrate_schemas --list
```

`migrate_schemas` raises an exception when an tenant schema is missing.

migrate_schemas in parallel

Once the number of tenants grow, migrating all the tenants can become a bottleneck. To speed up this process, you can run tenant migrations in parallel like this:

```
python manage.py migrate_schemas --executor=parallel
```

In fact, you can write your own executor which will run tenant migrations in any way you want, just take a look at `tenant_schemas/migration_executors`.

The `parallel` executor accepts the following settings:

- `TENANT_PARALLEL_MIGRATION_MAX_PROCESSES` (default: 2) - maximum number of processes for migration pool (this is to avoid exhausting the database connection pool)
- `TENANT_PARALLEL_MIGRATION_CHUNKS` (default: 2) - number of migrations to be sent at once to every worker

tenant_command

To run any command on an individual schema, you can use the special `tenant_command`, which creates a wrapper around your command so that it only runs on the schema you specify. For example

```
./manage.py tenant_command loaddata
```

If you don't specify a schema, you will be prompted to enter one. Otherwise, you may specify a schema preemptively

```
./manage.py tenant_command loaddata --schema=customer1
```

createsuperuser

The command `createsuperuser` is already automatically wrapped to have a `schema` flag. Create a new super user with

```
./manage.py createsuperuser --username=admin --schema=customer1
```

list_tenants

Prints to standard output a tab separated list of schema:domain_url values for each tenant.

```
for t in $(./manage.py list_tenants | cut -f1);
do
    ./manage.py tenant_command dumpdata --schema=$t --indent=2 auth.user > ${t}_users.
↪json;
done
```

Storage

The `storage` API will not isolate media per tenant. Your `MEDIA_ROOT` will be a shared space between all tenants.

To avoid this you should configure a tenant aware storage backend - you will be warned if this is not the case.

```
# settings.py

MEDIA_ROOT = '/data/media'
MEDIA_URL = '/media/'
DEFAULT_FILE_STORAGE = 'tenant_schemas.storage.TenantFileSystemStorage'
```

We provide `tenant_schemas.storage.TenantStorageMixin` which can be added to any third-party storage backend.

In your reverse proxy configuration you will need to capture use a regular expression to identify the `domain_url` to serve content from the appropriate directory.

```
# illustrative /etc/nginx/cond.d/tenant.conf

upstream web {
    server localhost:8080 fail_timeout=5s;
}

server {
    listen 80;
```

```
    server_name ~^(www\.)?(.+)$;

    location / {
        proxy_pass http://web;
        proxy_redirect off;
        proxy_set_header Host $host;
    }

    location /media/ {
        alias /data/media/$2/;
    }
}
```

Utils

There are several utils available in *tenant_schemas.utils* that can help you in writing more complicated applications.

schema_context(*schema_name*)

This is a context manager. Database queries performed inside it will be executed in against the passed `schema_name`.

```
from tenant_schemas.utils import schema_context

with schema_context(schema_name):
    # All comands here are ran under the schema `schema_name`

# Restores the `SEARCH_PATH` to its original value
```

tenant_context(*tenant_object*)

This context manager is very similiar to the `schema_context` function, but it takes a tenant model object as the argument instead.

```
from tenant_schemas.utils import tenant_context

with tenant_context(tenant):
    # All commands here are ran under the schema from the `tenant` object

# Restores the `SEARCH_PATH` to its original value
```

schema_exists(*schema_name*)

Returns `True` if a schema exists in the current database.

```
from django.core.exceptions import ValidationError
from django.utils.text import slugify

from tenant_schemas.utils import schema_exists

class TenantModelForm:
    # ...

    def clean_schema_name(self)
        schema_name = self.cleaned_data["schema_name"]
        schema_name = slugify(schema_name).replace("-", "")
        if schema_exists(schema_name):
            raise ValidationError("A schema with this name already exists in the␣
↪database")
```

```
        else:
            return schema_name
```

get_tenant_model()

Returns the class of the tenant model.

get_public_schema_name()

Returns the name of the public schema (from settings or the default `public`).

get_limit_set_calls()

Returns the `TENANT_LIMIT_SET_CALLS` setting or the default (`False`). See below.

Logging

The optional `TenantContextFilter` can be included in `settings.LOGGING` to add the current `schema_name` and `domain_url` to the logging context.

```
# settings.py

LOGGING = {
    'filters': {
        'tenant_context': {
            '()': 'tenant_schemas.log.TenantContextFilter'
        },
    },
    'formatters': {
        'tenant_context': {
            'format': '[%(schema_name)s:%(domain_url)s] '
            '%(levelname)-7s %(asctime)s %(message)s',
        },
    },
    'handlers': {
        'console': {
            'filters': ['tenant_context'],
        },
    },
}
```

This will result in logging output that looks similar to:

```
[example:example.com] DEBUG 13:29 django.db.backends: (0.001) SELECT ...
```

Performance Considerations

The hook for ensuring the `search_path` is set properly happens inside the `DatabaseWrapper` method `_cursor()`, which sets the path on every database operation. However, in a high volume environment, this can take considerable time. A flag, `TENANT_LIMIT_SET_CALLS`, is available to keep the number of calls to a minimum. The flag may be set in `settings.py` as follows:

```
# settings.py:

TENANT_LIMIT_SET_CALLS = True
```

When set, `django-tenant-schemas` will set the search path only once per request. The default is `False`.

Third Party Apps

Celery

Support for Celery is available at tenant-schemas-celery.

django-debug-toolbar

django-debug-toolbar routes need to be added to `urls.py` (both public and tenant) manually.

```
from django.conf import settings
from django.conf.urls import include
if settings.DEBUG:
    import debug_toolbar

    urlpatterns += patterns(
        '',
        url(r'^__debug__/', include(debug_toolbar.urls)),
    )
```

Advanced Usage

Custom tenant strategies (custom middleware support)

By default, `django-tenant-schemas`'s strategies for determining the correct tenant involve extracting it from the URL (e.g. `mytenant.mydomain.com`). This is done through a middleware, typically `TenantMiddleware`.

In some situations, it might be useful to use **alternative tenant selection strategies**. For example, consider a website with a fixed URL. An approach for this website might be to pass the tenant through a special header, or to determine it in some other manner based on the request (e.g. using an OAuth token mapped to a tenant). `django-tenant-schemas` offer an **easily extensible way to provide your own middleware** with minimal code changes.

To add custom tenant selection strategies, you need to **subclass the** `BaseTenantMiddleware` **class and implement its** `get_tenant` **method**. This method accepts the current `request` object through which you can determine the tenant to use. In addition, for backwards-compatibility reasons, the method also accepts the tenant model class (`TENANT_MODEL`) and the `hostname` of the current request. **You should return an instance of your** `TENANT_MODEL` **class** from this function. After creating your middleware, you should make it the top-most middleware in your list. You should only have one subclass of `BaseTenantMiddleware` per project.

Note that you might also wish to extend the other provided middleware classes, such as `TenantMiddleware`. For example, you might want to chain several strategies together, and you could do so by subclassing the original strategies and manipulating the call to `super`'s `get_tenant`.

Example: Determine tenant from HTTP header

Suppose you wanted to determine the current tenant based on a request header (`X-DTS-SCHEMA`). You might implement a simple middleware such as:

```
class XHeaderTenantMiddleware(BaseTenantMiddleware):
    """
    Determines tenant by the value of the ``X-DTS-SCHEMA`` HTTP header.
    """
```

```
    def get_tenant(self, model, hostname, request):
        schema_name = request.META.get('HTTP_X_DTS_SCHEMA', get_public_schema_name())
        return model.objects.get(schema_name=schema_name)
```

Your application could now specify the tenant with the `X-DTS-SCHEMA` HTTP header. In scenarios where you are configuring individual tenant websites by yourself, each with its own `nginx` configuration to redirect to the right tenant, you could use a configuration such as the one below:

```
# /etc/nginx/conf.d/multitenant.conf

upstream web {
    server localhost:8000;
}

server {
    listen 80 default_server;
    server_name _;

    location / {
        proxy_pass http://web;
        proxy_set_header Host $host;
    }
}

server {
    listen 80;
    server_name example.com www.example.com;

    location / {
        proxy_pass http://web;
        proxy_set_header Host $host;
        proxy_set_header X-DTS-SCHEMA example; # triggers XHeaderTenantMiddleware
    }
}
```

Examples

Tenant Tutorial

This app comes with an interactive tutorial to teach you how to use `django-tenant-schemas` and to demonstrate its capabilities. This example project is available under examples/tenant_tutorial. You will only need to edit the `settings.py` file to configure the `DATABASES` variable and then you're ready to run

```
./manage.py runserver
```

All other steps will be explained by following the tutorial, just open `http://127.0.0.1:8000` on your browser.

Specializing templates based on tenants

Multitenant aware filesystem template loader

The classical Django filesystem template loader cannot make the search path for template vary based on the current tenant so it's needed to use a special one which adapt the search path based on the tenant. For using it add it to your `TEMPLATE_LOADERS` setting.

```
TEMPLATE_LOADERS = (
    'tenant_schemas.template_loaders.FilesystemLoader',
    'django.template.loaders.filesystem.Loader',
    'django.template.loaders.app_directories.Loader'
)
```

This loader is looking for templates based on the setting `MULTITENANT_TEMPLATE_DIRS` instead of the path in `TEMPLATE_DIRS`. Templates are not searched directly in each directory `template_dir` but in the directory `os.path.join(template_dir, tenant.domain_url)`. If `template_dir` contains a `%s` formatting placeholder the directory used is `template_dir % tenant.domain_url` so that you can store your templates in a subdirectory of your tenant directory. Like with the Django `FilesystemLoader` the first found file is returned.

Multitenant aware cached template loader

If you are using template caching with the multitenant filesystem loader it is not gonna work as the cache is ignoring the tenant. So the first template loaded for any tenant will be returned for all other tenants. To remediate to this problem you can use a new loader whose cache is based on the template path and the primary key of the tenant model.

The multitenant cached loader works exactly like the Django cached loader but is tenant aware.

```
TEMPLATE_LOADERS = (
    ('tenant_schemas.template_loaders.CachedLoader', (
      'tenant_schemas.template_loaders.FilesystemLoader',
      'django.template.loaders.filesystem.Loader',
      'django.template.loaders.app_directories.Loader')),
)
```

Tests

Running the tests

Run these tests from the project `dts_test_project`, it comes prepacked with the correct settings file and extra apps to enable tests to ensure different apps can exist in `SHARED_APPS` and `TENANT_APPS`.

```
./manage.py test tenant_schemas.tests
```

To run the test suite outsite of your application you can use tox to test all supported Django versions.

```
tox
```

Updating your app's tests to work with tenant-schemas

Because django will not create tenants for you during your tests, we have packed some custom test cases and other utilities. If you want a test to happen at any of the tenant's domain, you can use the test case `TenantTestCase`. It will automatically create a tenant for you, set the connection's schema to tenant's schema and make it available at `self.tenant`. We have also included a `TenantRequestFactory` and a `TenantClient` so that your requests will all take place at the tenant's domain automatically. Here's an example

```
from tenant_schemas.test.cases import TenantTestCase
from tenant_schemas.test.client import TenantClient

class BaseSetup(TenantTestCase):
    def setUp(self):
        self.c = TenantClient(self.tenant)

    def test_user_profile_view(self):
        response = self.c.get(reverse('user_profile'))
        self.assertEqual(response.status_code, 200)
```

Running tests faster

Running tests using `TenantTestCase` can start being a bottleneck once the number of tests grow. `TenantTestCase` drops, recreates and executes migrations for the test schema every time for every `TenantTestCase` you have. If you do not care that the state between tests is kept, an alternative is to use the class `FastTenantTestCase`. Unlike `TenantTestCase`, the test schema and its migrations will only be created and ran once. This is a significant improvement in speed coming at the cost of shared state.

```
from tenant_schemas.test.cases import FastTenantTestCase
```

Get Involved!

Suggestions, bugs, ideas, patches, questions

Are **highly** welcome! Feel free to write an issue for any feedback you have or send a pull request on GitHub. :)

Index