tenantschemaDocumentation

Release dev

Thomas Turner

Aug 01, 2022

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

ONE

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

TWO

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

THREE

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.

Important: Tenant's domain name and schema name are usually the same or similar but they don't have to be! For example the tenant at http://acme.example.com could be backed by the `acme` schema, while http://looney-tunes.tld could be backed by the `tenant2` schema! Notice that domain names are not related in any way to schema names! There is also no restriction whether or not you should use sub-domains or top-level domains.

Warning: Schema names and domain names have different validation rules. Underscores (_) and capital letters are permitted in schema names but they are illegal for domain names! On the other hand domain names may contain a dash (-) which is illegal for schema names!

You must be careful if using schema names and domain names interchangeably in your multi-tenant applications! The tenant and domain model classes, creation and validation of input data are something that you need to handle yourself, possibly imposing additional constraints to the acceptable values!

CHAPTER

FOUR

SHARED AND TENANT-SPECIFIC APPLICATIONS

4.1 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.

4.2 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

FIVE

CONTENTS

5.1 Installation

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

```
pip install django-tenants
```

5.1.1 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': 'django_tenants.postgresql_backend',
        # ..
    }
}
```

Add *django_tenants.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 = (
    'django_tenants.routers.TenantSyncRouter',
)
```

Add the middleware `django_tenants.middleware.main.TenantMainMiddleware` to the top of `MIDDLEWARE`, so that each request can be set to use the correct schema.

```
MIDDLEWARE = (
    'django_tenants.middleware.main.TenantMainMiddleware',
    #...
)
```

Make sure you have `django.template.context_processors.request` listed under the `context_processors` option of `TEMPLATES` otherwise the tenant will not be available on `request`.

```
TEMPLATES = [
    {
        #...
        'OPTIONS': {
```

(continues on next page)

(continued from previous page)

```
            'context_processors': [
                'django.template.context_processors.request',
                #...
            ],
        },
    },
]
```

5.1.2 The Tenant & Domain 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 one field `schema_name` which is required. You also have to have a table for your domain names for this you **must** inherit from `DomainMixin` .

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 django_tenants.models import TenantMixin, DomainMixin

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

class Domain(DomainMixin):
    pass
```

5.1.3 Admin Support

TenantAdminMixin is available in order to register the tenant model. Here's an example (following the example above), we want to register the `Client` model, so we create a the related admin class `ClientAdmin`. The mixin disables save and delete buttons when not in current or public tenant (preventing Exceptions).

```
from django.contrib import admin
from django_tenants.admin import TenantAdminMixin

from myapp.models import Client

@admin.register(Client)
class ClientAdmin(TenantAdminMixin, admin.ModelAdmin):
        list_display = ('name', 'paid_until')
```

5.1.4 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 = (
    'django_tenants',  # mandatory
    '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 = (
    # your tenant-specific apps
    'myapp.hotels',
    'myapp.houses',
)

INSTALLED_APPS = list(SHARED_APPS) + [app for app in TENANT_APPS if app not in SHARED_
↪APPS]
```

You also have to set where your tenant & domain models are located.

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

TENANT_DOMAIN_MODEL = "customers.Domain"  # app.Model
```

Now run `migrate_schemas --shared`, this 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
```

Note: If you use `migrate` migrations will be applied to both shared and tenant schemas!

Warning: You might need to run `makemigrations` and then `migrate_schemas --shared` again for your `app.Models` to be created in the database.

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.

Warning: By default, a check is performed to validate that the additional extension schemas do not conflict with the tenant schemas. If you want to ignore this check, you can set SKIP_PG_EXTRA_VALIDATION to True. We do not recommend disabling this validation, **it is at your own risk**.

5.1.5 Sub-folder Support

Currently in beta.

There is a option that allows you to run Django-Tenants with sub-folder instead of sub-domains.

Note: e.g. http://www.mydomain.local/r/schemaname/ instead of http://schemaname.mydomain.local/

Warning: The schemaname value from the URL path has to match domain value which is set when creating a new tenant, as described https://django-tenants.readthedocs.io/en/latest/use.html#creating-a-tenant

TENANT_SUBFOLDER_PREFIX needs to be added to the settings file. This is the url prefix for the tenant this can't be left blank.

```
TENANT_SUBFOLDER_PREFIX = "clients"

In the example given above, the prefixed path ``/r`` will become ``/clients``.
e.g.  http://www.mydomain.local/clients/schemaname/ instead of http://www.mydomain.
↪local/r/schemaname/
```

The middleware is different to the standard middleware. The middleware required is

```
MIDDLEWARE = (
    'django_tenants.middleware.TenantSubfolderMiddleware',
    #...
)
```

Tip: There is an example project for this in the examples folder

5.1.6 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_CREATION_FAKES_MIGRATIONS

Default `'False'`

Sets if the schemas will be copied from an existing "template" schema instead of running migrations. Useful in the cases where migrations can not be faked and need to be ran individually, or when running migrations takes a long time. Be aware that setting this to *True* may significantly slow down the process of creating tenants.

When using this option, you must also specify which schema to use as template, under `TENANT_BASE_SCHEMA`.

TENANT_BASE_SCHEMA

Default `None`

The name of the schema to use as a template for creating new tenants. Only used when `TENANT_CREATION_FAKES_MIGRATIONS` is enabled.

TENANT_SYNC_ROUTER

Default `django_tenants.routers.TenantSyncRouter`

The name of the database router that `ready()` checks for when the django_tenant app checks for. If set then place this in `DATABASE_ROUTERS`.

```
DATABASE_ROUTERS = [
    # ..
    TENANT_SYNC_ROUTER
    # ..
]
```

TENANT_MIGRATION_ORDER

Default `None`

A list of fields to order the tenant queryset by when migrating schemas.

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 file 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.

5.1.7 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': 'django_tenants.cache.make_key',
        'REVERSE_KEY_FUNCTION': 'django_tenants.cache.reverse_key',
    },
}
```

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

5.1.8 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.

5.1.9 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`.

5.2 Using django-tenants

5.2.1 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, Domain

# create your public tenant
tenant = Client(schema_name='public',
                name='Schemas Inc.',
                paid_until='2016-12-05',
                on_trial=False)
tenant.save()

# Add one or more domains for the tenant
domain = Domain()
domain.domain = 'my-domain.com' # don't add your port or www here! on a local server␣
↪you'll want to use localhost here
domain.tenant = tenant
domain.is_primary = True
domain.save()
```

Now we can create our first real tenant.

```
from customers.models import Client, Domain

# create your first real tenant
tenant = Client(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!

# Add one or more domains for the tenant
domain = Domain()
domain.domain = 'tenant.my-domain.com' # don't add your port or www here!
domain.tenant = tenant
domain.is_primary = True
domain.save()
```

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.

5.2.2 Deleting a tenant

You can delete tenants by just deleting the entry via the Django ORM. There is a flag that can set on the tenant model called `auto_drop_schema`. The default for `auto_drop_schema` is False. WARNING SETTING `AUTO_DROP_SCHEMA` TO TRUE WITH DELETE WITH TENANT!

5.2.3 Utils

There are several utils available in *django_tenants.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`. (with statement)

```
from django_tenants.utils import schema_context

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

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

You can also use *schema_context* as a decorator.

```
from django_tenants.utils import schema_context

@schema_context(schema_name)
def my_func():
  # All commands in this function are ran under the schema `schema_name`
```

tenant_context(*tenant_object*)

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

```
from django_tenants.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
```

You can also use *tenant_context* as a decorator.

```
from django_tenants.utils import tenant_context

@tenant_context(tenant)
def my_func():
  # All commands in this function are ran under the schema from the `tenant` object
```

5.2.4 Signals

There are number of signals

`post_schema_sync` will get called after a schema gets created from the save method on the tenant class.

`schema_needs_to_be_sync` will get called if the schema needs to be migrated. `auto_create_schema` (on the tenant model) has to be set to False for this signal to get called. This signal is very useful when tenants are created via a background process such as celery.

`schema_migrated` will get called once migrations finish running for a schema.

`schema_migrate_message` will get called after each migration with the message of the migration. This signal is very useful when for process / status bars.

Example

```
@receiver(schema_needs_to_be_sync, sender=TenantMixin)
def created_user_client_in_background(sender, **kwargs):
    client = kwargs['tenant']
    print ("created_user_client_in_background %s" % client.schema_name)
    from clients.tasks import setup_tenant
    task = setup_tenant.delay(client)

@receiver(post_schema_sync, sender=TenantMixin)
def created_user_client(sender, **kwargs):

    client = kwargs['tenant']

    # send email to client to as tenant is ready to use

@receiver(schema_migrated, sender=run_migrations)
def handle_schema_migrated(sender, **kwargs):
    schema_name = kwargs['schema_name']

    # recreate materialized views in the schema

@receiver(schema_migrate_message, sender=run_migrations)
def handle_schema_migrate_message(**kwargs):
    message = kwargs['message']
    # recreate materialized views in the schema
```

5.2.5 Multi-types tenants

It is also possible to have different types of tenants. This is useful if you have two different types of users for instance you might want customers to use one style of tenant and suppliers to use another style. There is no limit to the amount of types however once the tenant has been set to a type it can't easily be convert to another type. To enable multi types you need to change the setting file and add an extra field onto the tenant table.

In the setting file `SHARED_APPS`, `TENANT_APPS` and `PUBLIC_SCHEMA_URLCONF` needs to be removed.

The following needs to be added to the setting file

```
HAS_MULTI_TYPE_TENANTS = True
MULTI_TYPE_DATABASE_FIELD = 'type'  # or whatever the name you call the database field

TENANT_TYPES = {
    "public": {  # this is the name of the public schema from get_public_schema_name
```

(continues on next page)

(continued from previous page)

```
        "APPS": ['django_tenants',
                 'django.contrib.admin',
                 'django.contrib.auth',
                 'django.contrib.contenttypes',
                 'django.contrib.sessions',
                 'django.contrib.messages',
                 'django.contrib.staticfiles',
                  # shared apps here
                 ],
        "URLCONF": "tenant_multi_types_tutorial.urls_public", # url for the public␣
↪type here
    },
    "type1": {
        "APPS": ['django.contrib.contenttypes',
                 'django.contrib.auth',
                 'django.contrib.admin',
                 'django.contrib.sessions',
                 'django.contrib.messages',
                 # type1 apps here
                 ],
        "URLCONF": "tenant_multi_types_tutorial.urls_type1",
    },
    "type2": {
        "APPS": ['django.contrib.contenttypes',
                 'django.contrib.auth',
                 'django.contrib.admin',
                 'django.contrib.sessions',
                 'django.contrib.messages',
                 # type1 apps here
                 ],
        "URLCONF": "tenant_multi_types_tutorial.urls_type2",
    }
}
```

Now you need to change the install app line in the settings file

```
INSTALLED_APPS = []
for schema in TENANT_TYPES:
    INSTALLED_APPS += [app for app in TENANT_TYPES[schema]["APPS"] if app not in␣
↪INSTALLED_APPS]
```

You also need to make sure that `ROOT_URLCONF` is blank

The tenant tables needs to have the following field added to the model

```
from django_tenants.utils import get_tenant_type_choices

class Client(TenantMixin):
    type = models.CharField(max_length=100, choices=get_tenant_type_choices())
```

That's all you need to add the multiple types.

There is an example project called `tenant_multi_types`

5.2.6 Other settings

By default if no tenant is found it will raise an error Http404 however you add `SHOW_PUBLIC_IF_NO_TENANT_FOUND` to the setting it will display the the public tenant. This will not work for subfolders.

Admin

By default if you look at the admin all the tenant apps will be colored dark green you can disable this by doing.

```
TENANT_COLOR_ADMIN_APPS = False
```

Reverse

You can get the tenant domain name by calling a method on the tenant model called `reverse`.

5.2.7 Management commands

Every command except tenant_command runs by default on all tenants. You can also create your own commands that run on every tenant by inheriting `BaseTenantCommand`. To run only a particular schema, there is an optional argument called `--schema`.

Custom command example:

```
from django_tenants.management.commands import BaseTenantCommand
# rest of your imports

class Command(BaseTenantCommand):
    COMMAND_NAME = 'awesome command'
    # rest of your command
```

```
./manage.py migrate_schemas --schema=customer1
```

migrate_schemas

We've also packed the django migrate command in a compatible way with this app. It will also respect the `SHARED_APPS` and `TENANT_APPS` settings, so if you're migrating the `public` schema it will only migrate `SHARED_APPS`. If you're migrating tenants, it will only migrate `TENANT_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
```

Or

```
./manage.py migrate_schemas myapp 0001_initial --fake
```

in case you're just switching your `myapp` application to use South migrations.

migrate_schemas in Parallel

You can run tenant migrations in parallel like this:

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

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

The `multiprocessing` executor accepts the following settings:

- `TENANT_MULTIPROCESSING_MAX_PROCESSES` (default: 2) - maximum number of processes for migration pool (this is to avoid exhausting the database connection pool)
- `TENANT_MULTIPROCESSING_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
```

all_tenants_command

To run any command on an every schema, you can use the special `all_tenants_command`, which creates a wrapper around your command so that it run on every schema. For example

```
./manage.py all_tenants_command loaddata
```

create_tenant_superuser

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

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

create_tenant

The command `create_tenant` creates a new schema

```
./manage.py create_tenant --domain-domain=newtenant.net --schema_name=new_tenant --
↪name=new_tenant --description="New tenant"
```

The argument are dynamic depending on the fields that are in the `TenantMixin` model. For example if you have a field in the `TenantMixin` model called company you will be able to set this using –company=MyCompany. If no argument are specified for a field then you be promted for the values. There is an additional argument of -s which sets up a superuser for that tenant.

delete_tenant

The command `delete_tenant` deletes a schema

```
./manage.py delete_tenant
```

Warning this command will delete a tenant and PostgreSQL schema regardless if `auto_drop_schema` is set to False.

clone_tenant

The command `clone_tenant` clones a schema.

```
./manage.py clone_tenant
```

There are some options to that can be set. You can view all the options by running

```
./manage.py clone_tenant -h
```

Credits to pg-clone-schema.

rename_schema

The command `rename_schema` renames a schema in the db and updates the Client associated with it.

```
./manage.py rename_schema
```

It will prompt you for the current name of the schema, and what it should be renamed to.

You can provide them with these arguments:

```
./manage.py rename_schema --rename_from old_name --rename_to new_name
```

create_missing_schemas

The command `create_missing_schemas` checks the tenant table against the list of schemas. If it find a schema that doesn't exist it will create it.

```
./manage.py create_missing_schemas
```

5.2.8 PostGIS

If you want to run PostGIS add the following to your Django settings file

```
ORIGINAL_BACKEND = "django.contrib.gis.db.backends.postgis"
```

5.2.9 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:

```
#in settings.py:
TENANT_LIMIT_SET_CALLS = True
```

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

5.2.10 Extra Set Tenant Method

Sometime you might want to do something special when you switch to another schema / tenant such as read replica. Add `EXTRA_SET_TENANT_METHOD_PATH` to the settings file and point a method.

```
EXTRA_SET_TENANT_METHOD_PATH = 'tenant_multi_types_tutorial.set_tenant_utils.extra_
↪set_tenant_stuff'
```

The method

The method takes 2 arguments the first is the database wrapper class and the second is the tenant. example

```
def extra_set_tenant_stuff(wrapper_class, tenant):
    pass
```

5.2.11 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': {
            '()': 'django_tenants.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 ...
```

5.2.12 Running in Development

If you want to use django-tenant in development you need to use a fake a domain name. All domains under the TLD `.localhost` will be routed to your local machine, so you can use things like `tenant1.localhost` and `tenant2.localhost`.

5.2.13 Migrating Single-Tenant to Multi-Tenant

Warning: The following instructions may or may not work for you. Use at your own risk!

- Create a backup of your existing single-tenant database, presumably non PostgreSQL:

```
./manage.py dumpdata --all --indent 2 > database.json
```

- Edit `settings.py` to connect to your new PostrgeSQL database
- Execute `manage.py migrate` to create all tables in the PostgreSQL database
- Ensure newly created tables are empty:

```
./manage.py sqlflush | ./manage.py dbshell
```

- Load previously exported data into the database:

```
./manage.py loaddata --format json database.json
```

- Create the `public` tenant:

```
./manage.py create_tenant
```

At this point your application should be multi-tenant aware and you may proceed creating more tenants.

5.2.14 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
```

(continues on next page)

(continued from previous page)

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

5.2.15 Useful information

Running code across every tenant

If you want to run some code on every tenant you can do the following

```
from django_tenants.utils import tenant_context, get_tenant_model

for tenant in get_tenant_model().objects.all():
    with tenant_context(tenant):
        pass
        # do whatever you want in that tenant
```

5.3 Examples

5.3.1 Tenant Tutorial

This app comes with an interactive tutorial to teach you how to use `django-tenants` 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.

5.3.2 Running the example projects with Docker Compose

To run the example projects with docker-compose. You will need.

1. Docker
2. Docker Compose

Then you can run

```
docker-compose run web bash

cd examples/tenant_tutorial

python manage.py migrate

python manage.py create_tenant

python manage.py runserver 0.0.0.0:8088
```

All other steps will be explained by following the tutorial, just open `http://127.0.0.1:8088` on your browser.

5.4 Tenant-aware file handling

The default Django behaviour is for all tenants to share **one** set of templates and static files between them. This can be changed so that each tenant will have its own:

- Static files (like cascading stylesheets and JavaScript)
- Location for files uploaded by users (usually stored in a */media* directory)
- Django templates

The process for making Django's file handling tenant-aware generally consists of the following steps:

1. Using a custom tenant-aware `finder` for locating files
2. Specifying where `finders` should look for files
3. Using a custom tenant-aware file `storage` handler for collecting and managing those files
4. Using a custom tenant-aware `loader` for finding and loading Django templates

We'll cover the configuration steps for each in turn.

5.4.1 Project layout

This configuration guide assumes the following Django project layout (loosely based on django-cookiecutter):

```
absolute/path/to/your_project_dir
    ...
    static              # System-wide static files
    templates           # System-wide templates
    # Tenant-specific files below will override pre-existing system-wide files with␣
↪same name.
    tenants
        tenant_1        # Static files / templates for tenant_1
            templates
            static
        tenant_2        # Static files / templates for tenant_2
            templates
            static
    media               # Created automatically when users upload files
        tenant_1
        tenant_2
staticfiles             # Created automatically when collectstatic_schemas is run
    tenant_1
    tenant_2
    ...
```

The configuration details may differ depending on your specific requirements for your chosen layout. Fortunately, django-tenants makes it easy to cater for a wide range of project layouts as illustrated below.

Configuring the static file finders

Start by inserting django-tenants' `django_tenants.staticfiles.finders.TenantFileSystemFinder` at the top of the list of available `STATICFILES_FINDERS` in your Django configuration file:

```
# in settings.py

STATICFILES_FINDERS = [
    "django_tenants.staticfiles.finders.TenantFileSystemFinder",  # Must be first
    "django.contrib.staticfiles.finders.FileSystemFinder",
    "django.contrib.staticfiles.finders.AppDirectoriesFinder",
    "compressor.finders.CompressorFinder",
]

# or this way

STATICFILES_FINDERS.insert(0, "django_tenants.staticfiles.finders.
↪TenantFileSystemFinder")
```

By adding `TenantFileSystemFinder` at the top, we ensure that Django will look for the tenant-specific files first, before reverting to the standard search path. This makes it possible for tenants to *override* any static files (e.g. stylesheets or javascript files) that are specific to that tenant, and use the standard static files for the rest.

Next, add `MULTITENANT_STATICFILES_DIRS` to the configuration file in order to let `TenantFileSystemFinder` know where to look for tenant-specific static files:

```
# in settings.py

MULTITENANT_STATICFILES_DIRS = [
    os.path.join( "absolute/path/to/your_project_dir", "tenants/%s/static" ),
]
```

For the path provided above, `%s` will be replaced with the current tenant's `schema_name` during runtime (see *Specifying a different target directory* for details).

Configuring the static files storage

By default, Django uses `StaticFilesStorage` for collecting static files into a dedicated folder on the server when the `collectstatic` management command is run. The location that the files are written to is specified in the `STATIC_ROOT` setting (usually configured to point to *'staticfiles'*).

django-tenants provides a replacement tenant-aware `TenantStaticFilesStorage` than can be configured by setting:

```
# in settings.py

STATICFILES_STORAGE = "django_tenants.staticfiles.storage.TenantStaticFilesStorage"

MULTITENANT_RELATIVE_STATIC_ROOT = ""  # (default: create sub-directory for each␣
↪tenant)
```

The path specified in `MULTITENANT_RELATIVE_STATIC_ROOT` tells `TenantStaticFilesStorage` where in `STATIC_ROOT` the tenant's files should be saved. The default behaviour is to just create a sub-directory for each tenant in `STATIC_ROOT`.

The command to collect the static files for all tenants is `collectstatic_schemas`. The optional `--schema` argument can be used to only collect files for a single tenant.

```
./manage.py collectstatic_schemas --schema=your_tenant_schema_name
```

Note: If you have configured an HTTP server, like nginx, to serve static files instead of the Django built-in server, then you also need to set `REWRITE_STATIC_URLS = True`. This tells django-tenants to rewrite `STATIC_URL` to include `MULTITENANT_RELATIVE_STATIC_ROOT` when static files are requested so that these files can be found and served directly by the external HTTP server.

Configuring media file storage

The default Django behavior is to store all files that are uploaded by users in one folder. The path for this upload folder can be configured via the standard `MEDIA_ROOT` setting.

The above behaviour can be changed for multi-tenant setups so that each tenant will have a dedicated sub-directory for storing user-uploaded files. To do this simply change `DEFAULT_FILE_STORAGE` so that `TenantFileSystemStorage` replaces the standard `FileSystemStorage` handler:

```
# in settings.py

DEFAULT_FILE_STORAGE = "django_tenants.files.storage.TenantFileSystemStorage"

MULTITENANT_RELATIVE_MEDIA_ROOT = ""  # (default: create sub-directory for each␣
↪tenant)
```

The path specified in `MULTITENANT_RELATIVE_MEDIA_ROOT` tells `TenantFileSystemStorage` where in `MEDIA_ROOT` the tenant's files should be saved. The default behaviour is to just create a sub-directory for each tenant in `MEDIA_ROOT`.

Configuring the template loaders

django-tenants provides a tenant-aware template loader that uses the current tenant's `schema_name` when looking for templates.

It can be configured by inserting the custom `Loader` at the top of the list in the `TEMPLATES` setting, and specifying the template search path to be used in the `MULTITENANT_TEMPLATE_DIRS` setting, as illustrated below:

```
TEMPLATES = [
    {
        ...
        "DIRS": ["absolute/path/to/your_project_dir/templates"],  # -> Dirs used by␣
↪the standard template loader
        "OPTIONS": {
            ...
            "loaders": [
                "django_tenants.template.loaders.filesystem.Loader",  # Must be first
                "django.template.loaders.filesystem.Loader",
                "django.template.loaders.app_directories.Loader",
            ],
            ...
        ...
    }
```

(continues on next page)

(continued from previous page)

```
]

MULTITENANT_TEMPLATE_DIRS = [
    "absolute/path/to/your_project_dir/tenants/%s/templates"
]
```

Notice that `django_tenants.template.loaders.filesystem.Loader` is added at the top of the list. This will cause Django to look for the tenant-specific templates first, before reverting to the standard search path. This makes it possible for tenants to *override* individual templates as required.

Just like with standard Django, the first template found will be returned.

Attention: If the template contains any include tags, then all of the included templates need to be located in the tenant's template folder as well. It is not currently possible to include templates from sources outside of the tenant's template folder.

Specifying a different target directory

django-tenants supports simple Python string formatting for configuring the various path strings used throughout the configuration steps. any occurances of `%s` in the path string will be replaced with the current tenant's `schema_name` during runtime.

This makes it possible to cater for more elaborate folder structures. Some examples are provided below:

```
# in settings.py

STATIC_ROOT = "absolute/path/to/your_project_dir/staticfiles"

MULTITENANT_RELATIVE_STATIC_ROOT = "tenants/%s
```

Static files will be collected into -> absolute/path/to/your_project_dir/staticfiles/tenants/`schema_name`.

. . . and for media files:

```
# in settings.py

MEDIA_ROOT = "absolute/path/to/your_project_dir/apps_dir/media/"

MULTITENANT_RELATIVE_MEDIA_ROOT = "%s/other_dir"
```

Media files will be uploaded at -> absolute/path/to/your_project_dir/apps_dir/media/`schema_name`/other_dir

5.5 Tests

5.5.1 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 django_tenants.tests
```

If you want to run with custom migration executor then do

```
EXECUTOR=multiprocessing ./manage.py test django_tenants.tests
```

You can also run the tests with docker-compose

```
docker-compose run django-tenants-test
```

5.5.2 Updating your app's tests to work with django_tenants

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 django_tenants.test.cases import TenantTestCase
from django_tenants.test.client import TenantClient

class BaseSetup(TenantTestCase):

    def setUp(self):
        super().setUp()
        self.c = TenantClient(self.tenant)

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

5.5.3 Additional information

You may have other fields on your tenant or domain model which are required fields. If you have there are two routines to look at `setup_tenant` and `setup_domain`

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

class BaseSetup(TenantTestCase):

    @classmethod
    def setup_tenant(cls, tenant):
        """
        Add any additional setting to the tenant before it get saved. This is␣
↪required if you have
        required fields.
        """
        tenant.required_value = "Value"
        return tenant

    def setup_domain(self, domain):
        """
        Add any additional setting to the domain before it get saved. This is␣
↪required if you have
        required fields.
        """
```

(continues on next page)

(continued from previous page)

```
        domain.ssl = True
        return domain

    def setUp(self):
        super().setUp()
        self.c = TenantClient(self.tenant)

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

You can also change the test domain name and the test schema name by using `get_test_schema_name` and `get_test_tenant_domain`. by default the domain name is `tenant.test.com` and the schema name is `test`.

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

class BaseSetup(TenantTestCase):
    @staticmethod
    def get_test_tenant_domain():
        return 'tenant.my_domain.com'


    @staticmethod
    def get_test_schema_name():
        return 'tester'
```

You can set the the verbosity by overriding the `get_verbosity` method.

5.5.4 Running tests faster

Using the `TenantTestCase` can make running your tests really slow quite early in your project. This is due to the fact that it drops, recreates the test schema and runs migrations for every `TenantTestCase` you have. If you want to gain speed, there's a `FastTenantTestCase` where the test schema will be created and migrations ran only one time. The gain in speed is noticiable but be aware that by using this you will be perpertraiting state between your test cases, please make sure your they wont be affected by this.

Running tests using `TenantTestCase` can start being a bottleneck once the number of tests grow. 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 django_tenants.test.cases import FastTenantTestCase
```

There are some extra method that you can use for FastTenantTestCase. They are.

`flush_data` default is True which means is will empty the table after each run. False will keep the data

```
class FastTenantTestCase(TenantTestCase):

    @classmethod
    def flush_data(cls):
        return True
```

`use_existing_tenant` Gets run if the setup doesn't need to create a new database `use_new_tenant` Get run is an new database is created

```
class FastTenantTestCase(TenantTestCase):
    @classmethod
    def use_existing_tenant(cls):
        pass

    @classmethod
    def use_new_tenant(cls):
        pass
```

5.6 Useful links

5.6.1 Tom's Youtube Channel

An example of using Django Tenants and other Python / Django related video Click Here

5.6.2 SaaSy maps

SaaSy maps - using django-tenants and geodjango to provide web-gis software-as-a-service http://www.slideshare.net/AnushaChickermane/saasy-maps

5.6.3 django-tenant-users

An application expands the django users and permissions frameworks https://github.com/Corvia/django-tenant-users

5.7 Get Involved!

5.7.1 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 <https://github.com/django-tenants/django-tenants>*. :)

5.8 Credits

5.8.1 django-tenant-schemas

I would like to thank the original author of this project Bernardo Pires Carneiro under the name django-tenant-schemas. I forked this project as I wanted to add enhancements to make it work better with Django 1.8

If you are using Django 1.7 or below please use his project.

INDEX