# Building Multi Tenant Applications with Django Documentation

*Release 2.0*

**Agiliq and Contributors**

**Aug 28, 2019**

# Table of Contents:

**1 Introduction to multi tenant applications** **3**
- 1.1 What are multi tenant apps? . . . 3
- 1.2 The structure of this book . . . 3
- 1.3 The various approached to multi tenancy . . . 3

**2 Shared database with shared schema** **5**
- 2.1 The base single-tenant app . . . 5
- 2.2 Adding multi tenancy to models . . . 6
- 2.3 Identifying tenants . . . 6
- 2.4 Extracting tenant from request . . . 7
- 2.5 A detour to /etc/hosts . . . 7
- 2.6 Using `tenant_from_request` in the views . . . 7
- 2.7 Isolating the admin . . . 8

**3 Shared database with isolated schema** **9**
- 3.1 Limitations of shared schema and our current method . . . 9
- 3.2 What are database schemas? . . . 10
- 3.3 Managing database migrations . . . 10
- 3.4 Tenant separation in views . . . 11
- 3.5 A middleware to set schemas . . . 12
- 3.6 Beyond the request-response cycle . . . 12

**4 Isolated database with a shared app server** **15**
- 4.1 Multiple database support in Django . . . 15
- 4.2 Database routing in Django . . . 16
- 4.3 Per tenant database routing using middlewares . . . 16
- 4.4 Outside the request response cycle . . . 17

**5 Completely isolated tenants using Docker** **19**
- 5.1 Tools we will use . . . 19
- 5.2 Building a docker image from our app code . . . 19
- 5.3 Using docker-compose to run multi container, multi-tenant apps . . . 20
- 5.4 The final docker-compose.yaml . . . 20

**6 Tying it all together** **25**
- 6.1 Launching new tenants . . . 25
- 6.2 A comparison of trade-offs of various methods . . . 25

6.3 What method should I use? . . . . . . . . . . 26

**7 Third party apps** **27**
7.1 Open source Django multi tenancy apps . . . . . . . . . . 27
7.2 A tour of django-tenant-schemas . . . . . . . . . . 27

**8 Indices and tables** **31**

BUILDING

MULTI TENANT

APPLICATIONS

WITH

DJANGO

UPDATED FOR DJANGO
2.0 AND PYTHON 3.7

django

python™

CHAPTER 1

# Introduction to multi tenant applications

## 1.1 What are multi tenant apps?

Multi tenant applications allow you to serve multiple customers with one install of the application. Each customer has their data completely isolated in such an architecture. Each customer is called a tenant.

Most modern Software as a Service applications are multi tenant. Whether it is Salesforce, Freshbooks, Zoho or Wordpress, most modern cloud based applications are delivered with a multi-tenant architecture.

## 1.2 The structure of this book

In this book we will take a single tenant application and re-architect it to be a multi tenant application. We will use a slightly modified Django polls app as our base.

There are multiple approaches for multi tenancy. We will look at the four most common ones.

## 1.3 The various approached to multi tenancy

- Shared database with shared schema
- Shared database with isolated schema
- Isolated database with a shared app server
- Completely isolated tenants using Docker

### 1.3.1 Shared database with shared schema

A single database keeps every tenant's data. A `ForeignKey` in the tables identifies the tenant.

### 1.3.2 Shared database with isolated schema

A single database keeps every tenant's data. Each tenant's data is in a separate schema within the single database. The schema identifies the tenant and data tabled do no have a FK to the tenant.

### 1.3.3 Isolated database with a shared app server

Every tenant's data is in a separate database. The database identifies the tenant.

### 1.3.4 Completely isolated tenants using Docker

A new set of docker containers are launched for each tenant. Every tenant's data is in a separate database (which may or may not be running in container). A set of containers identifies the tenant.

In the next four chapters, we will look at each architecture in turn. Let's get started.

# CHAPTER 2

## Shared database with shared schema

In this chapter, we will rebuild a slightly modified Django polls app to be multi-tenant. You can download the code from Github.

### 2.1 The base single-tenant app

Our base project has one app called `polls`. The models look something like this.

```
from django.db import models
from django.contrib.auth.models import User


class Poll(models.Model):
    question = models.CharField(max_length=100)
    created_by = models.ForeignKey(User, on_delete=models.CASCADE)
    pub_date = models.DateTimeField(auto_now=True)

    def __str__(self):
        return self.question


class Choice(models.Model):
    poll = models.ForeignKey(Poll, related_name='choices',on_delete=models.CASCADE)
    choice_text = models.CharField(max_length=100)

    def __str__(self):
        return self.choice_text


class Vote(models.Model):
    choice = models.ForeignKey(Choice, related_name='votes', on_delete=models.CASCADE)
    poll = models.ForeignKey(Poll, on_delete=models.CASCADE)
    voted_by = models.ForeignKey(User, on_delete=models.CASCADE)
```

(continues on next page)

(continued from previous page)

```
    class Meta:
        unique_together = ("poll", "voted_by")
```

There are a number of other files which we will look at later.

## 2.2 Adding multi tenancy to models

We will add another app called `tenants`

```
python manage.py startapp tenants
```

Create a model for storing `Tenant` data.

```
class Tenant(models.Model):
    name = models.CharField(max_length=100)
    subdomain_prefix = models.CharField(max_length=100, unique=True)
```

And then create a class `TenantAwareModel` class which other models will subclass from.

```
class TenantAwareModel(models.Model):
    tenant = models.ForeignKey(Tenant, on_delete=models.CASCADE)

    class Meta:
        abstract = True
```

Change the `polls.models` to subclass from `TenantAwareModel`.

```
# ...

class Poll(TenantAwareModel):
    # ...


class Choice(TenantAwareModel):
    # ...


class Vote(TenantAwareModel):
    # ...
```

## 2.3 Identifying tenants

There are many approaches to identify the tenant. One common method is to give each tenant their own subdomain. So if you main website is

`www.example.com`

And each of the following will be a separate tenant.

- thor.example.com
- loki.example.com

- potter.example.com

We will use the same method in the rest of the book. Our `Tenant` model has `subdomain_prefix` which will identify the tenant.

We will use `polls.local` as the main domain and `<xxx>.polls.local` as tenant subdomain.

## 2.4 Extracting tenant from request

Django views always have a `request` which has the `Host` header. This will contain the full subdomain the tenant is using. We will add some utility methods to do this. Create a `utils.py` and add this code.

```
from .models import Tenant


def hostname_from_request(request):
    # split on `:` to remove port
    return request.get_host().split(':')[0].lower()


def tenant_from_request(request):
    hostname = hostname_from_request(request)
    subdomain_prefix = hostname.split('.')[0]
    return Tenant.objects.filter(subdomain_prefix=subdomain_prefix).first()
```

Now wherever you have a `request`, you can use `tenant_from_request` to get the tenant.

## 2.5 A detour to /etc/hosts

To ensure that the `<xxx>.polls.local` hits your development machine, make sure you add a few entries to your `/etc/hosts`

(If you are on windows, use `C:\Windows\System32\Drivers\etc\hosts`). My file looks like this.

```
# ...
127.0.0.1 polls.local
127.0.0.1 thor.polls.local
127.0.0.1 potter.polls.local
```

Also update `ALLOWED_HOSTS` your settings.py. Mine looks like this: `ALLOWED_HOSTS = ['polls.local', '.polls.local']`.

## 2.6 Using `tenant_from_request` in the views

Views, whether they are Django function based, class based or a Django Rest Framework view have access to the request. Lets take the example of `polls.views.PollViewSet` to limit the endpoints to tenant specific `Poll` objects.

```
from tenants.utils import tenant_from_request


class PollViewSet(viewsets.ModelViewSet):
```

(continues on next page)

(continued from previous page)

```
    queryset = Poll.objects.all()
    serializer_class = PollSerializer

    def get_queryset(self):
        tenant = tenant_from_request(self.request)
        return super().get_queryset().filter(tenant=tenant)
```

## 2.7 Isolating the admin

Like the views we need to enforce tenant isolation on the admin. We will need to override two methods.

- `get_queryset`: So that only the current tenant's objects show up.
- `save_model`: So that tenant gets set on the object when the object is saved.

With the changes, your `admin.py` looks something like this.

```
@admin.register(Poll)
class PollAdmin(admin.ModelAdmin):
    fields = ["question", "created_by", "pub_date"]
    readonly_fields = ["pub_date"]

    def get_queryset(self, request, *args, **kwargs):
        queryset = super().get_queryset(request, *args, **kwargs)
        tenant = tenant_from_request(request)
        queryset = queryset.filter(tenant=tenant)
        return queryset

    def save_model(self, request, obj, form, change):
        tenant = tenant_from_request(request)
        obj.tenant = tenant
        super().save_model(request, obj, form, change)
```

With these changes, you have a basic multi-tenant app. But there is a lot more to do as we will see in the following chapters.

The code for this chapter is available at https://github.com/agiliq/building-multi-tenant-applications-with-django/tree/master/shared-db

# CHAPTER 3

# Shared database with isolated schema

## 3.1 Limitations of shared schema and our current method

In the previous chapter we used a `ForeignKey` to separate the tenants. This method is simple but limited due to the following:

- Weak separation of tenant's data
- Tenant isolation code is intermixed with app code
- Duplication of code

### 3.1.1 Weak separation of tenant's data

Because each tenant's data stays in the same schema, there is no way to limit access to a single tenant's data at the DB level.

### 3.1.2 Tenant isolation code is intermixed with app code

You need to litter your code with `.filter(tenant=tenant)` every time you access the database. For example in your `ViewSet` you would be doing this:

```
def get_queryset(self):
    tenant = tenant_from_request(self.request)
    return super().get_queryset().filter(tenant=tenant)
```

If you even miss a `filter`, you would be mixing data from two tenants. This will be a bad security bug.

### 3.1.3 Duplication of code

The tenant separation code of getting the tenant from the request and filtering on it is all over your codebase, rather than a central location.

In this chapter, we will rearchitect our code to use Shared database with isolated schema, which will fix most of these limitations.

## 3.2 What are database schemas?

Schemas in database are a way to group objects. Postgres documentation defines schema as

> A database contains one or more named schemas, which in turn contain tables. Schemas also contain other kinds of named objects, including data types, functions, and operators. The same object name can be used in different schemas without conflict; for example, both schema1 and myschema may contain tables named mytable.

For the rest of the chapter, we will be using Postgres. We will be using one schema per tenant.

We need some way to keeping a mapping on tenants to schemas. There are a number of ways you could do it, for example by keeping a table in public schema to map tenant urls to schemas. In this chapter, for simplicity, we will keep a simple map of tenant urls to schemas.

Add this to your utils.py

```
def get_tenants_map():
    return {
        "thor.polls.local": "thor",
        "potter.polls.local": "potter",
    }
```

Now when we get a request to `thor.polls.local` we need to read from the schema `thor`, and when we get a request to `potter.polls.local` we need to read from schema `potter`.

## 3.3 Managing database migrations

`manage.py migrate` is not schema aware. So we will need to subclass this command so that tables are created in all the schemas. Create the folder structure for a new command following the usual django convention. Then add a file named `migrate_schemas` in there.

```
from django.core.management.commands.migrate import Command as MigrationCommand

from django.db import connection
from ...utils import get_tenants_map


class Command(MigrationCommand):
    def handle(self, *args, **options):
        with connection.cursor() as cursor:
            schemas = get_tenants_map().values()
            for schema in schemas:
                cursor.execute(f"CREATE SCHEMA IF NOT EXISTS {schema}")
                cursor.execute(f"SET search_path to {schema}")
                super(Command, self).handle(*args, **options)
```

To understand what we are doing here, you need to know a few Postgres queries.

- `CREATE SCHEMA IF NOT EXISTS potter` creates a new schema named potter.
- `SET search_path to potter` set the connection to use the given schema.

Now when you run `manage.py migrate_schemas` it loops over the our tenants map, then creates a schema for that tenant and runs the migration for the tenant.

## 3.4 Tenant separation in views

Lets add a few utility methods which will allow us to get and set the schema. Add the following functions to your `utils.py`.

```
def hostname_from_request(request):
    # split on `:` to remove port
    return request.get_host().split(':')[0].lower()


def tenant_schema_from_request(request):
    hostname = hostname_from_request(request)
    tenants_map = get_tenants_map()
    return tenants_map.get(hostname)


def set_tenant_schema_for_request(request):
    schema = tenant_schema_from_request(request)
    with connection.cursor() as cursor:
        cursor.execute(f"SET search_path to {schema}")
```

Now we can separate the tenants in the views using these functions.

```
# apiviews.py
# ...
from tenants.utils import set_tenant_schema_for_request


class PollViewSet(viewsets.ModelViewSet):
    queryset = Poll.objects.all()
    serializer_class = PollSerializer

    def get_queryset(self):
        set_tenant_schema_for_request(self.request)
        tenant = tenant_from_request(self.request)
        return super().get_queryset().filter(tenant=tenant)

    def destroy(self, request, *args, **kwargs):
        set_tenant_schema_for_request(self.request)
        poll = Poll.objects.get(pk=self.kwargs["pk"])
        if not request.user == poll.created_by:
            raise PermissionDenied("You can not delete this poll.")
        return super().destroy(request, *args, **kwargs)

# ...
```

```
# admin.py
# ...
from tenants.utils import tenant_schema_from_request

@admin.register(Poll)
class PollAdmin(admin.ModelAdmin):
```

(continues on next page)

(continued from previous page)

```
    fields = ["question", "created_by", "pub_date"]
    readonly_fields = ["pub_date"]

    def get_queryset(self, request, *args, **kwargs):
        set_tenant_schema_for_request(self.request)
        queryset = super().get_queryset(request, *args, **kwargs)
        tenant = tenant_from_request(request)
        queryset = queryset.filter(tenant=tenant)
        return queryset

    def save_model(self, request, obj, form, change):
        set_tenant_schema_for_request(self.request)
        tenant = tenant_from_request(request)
        obj.tenant = tenant
        super().save_model(request, obj, form, change)
```

## 3.5 A middleware to set schemas

Our naive approach to separate the tenants suffers from a few problems:

- `set_tenant_schema_for_request(self.request)` is duplicated everywhere
- Any third party code, including Django's, ORM accesses will fail because they will try to access the objects from the public schema, which is empty.

Both of these can be fixed by using a middleware. We will set the schema in the middleware before any view code comes in play, so any ORM code will pull and write the data from the tenant's schema.

Create a new middleware like this:

```
from tenants.utils import set_tenant_schema_for_request

class TenantMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        set_tenant_schema_for_request(request)
        response = self.get_response(request)
        return response
```

And add it to your `settings.MIDDLEWARES`

```
MIDDLEWARE = [
    # ...
    'tenants.middlewares.TenantMiddleware',
]
```

## 3.6 Beyond the request-response cycle

We have one more change to make before we are done. You can not use *manage.py createssuperuser* or any Django command, as manage.py will try to use the public schema, and there are no tables in the public schema.

Middleware is only used in the request-response cycle and does not come into play when you run a command. Therefore we need another place to hook our `set_tenant_schema_for_request`. To do this, create a new file `tenant_context_manage.py`. This is similar to `manage.py`, with a few minor changes.

```
#!/usr/bin/env python
import os
import sys


if __name__ == "__main__":
    os.environ.setdefault("DJANGO_SETTINGS_MODULE", "pollsapi.settings")
    try:
        from django.core.management import execute_from_command_line
    except ImportError as exc:
        raise ImportError(
            "Couldn't import Django. Are you sure it's installed and "
            "available on your PYTHONPATH environment variable? Did you "
            "forget to activate a virtual environment?"
        ) from exc
    from django.db import connection
    args = sys.argv
    schema = args[1]
    with connection.cursor() as cursor:

        cursor.execute(f"SET search_path to {schema}")

        del args[1]
        execute_from_command_line(args)
```

This allows setting the tenant schema, which is passed as first argument before running the command.

We will be able to use it like this. `python tenant_context_manage.py thor createsuperuser`.

With this, you can login to any tenant's admin, create some objects, and view the API endpoints. Here is what the polls api endpoint looks like for me.

Api Root / Poll List

Poll List

OPTIONS GET

GET /polls/

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "choices": [],
        "question": "Who is the best avenger?",
        "pub_date": "2018-07-04T17:09:20.945143Z",
        "created_by": 1
    }
]
```

Raw data HTML form

Question

Created by shabda

POST

In the next chapter we will look at separating the tenants to their own databases.

The code for this chapter is available at https://github.com/agiliq/building-multi-tenant-applications-with-django/tree/master/isolated-schema

# CHAPTER 4

# Isolated database with a shared app server

In the previous chapter we used schemas to separate each tenant's data. In this chapter we will keep each tenant's data in a separate DB. For this chapter we will use sqlite, though any DB supported by Django will suffice. Our core architecture will be quite similar to the previous chapter, where we

- Used request header to find the tenant
- Created a mapping of tenants to schemas
- Set the tenant specific schema in middleware

In this chapter, we will

- Use request header to find the tenant
- Create a mapping of tenants to databases
- Set the tenant specific database in middleware.

Let's get rolling.

## 4.1 Multiple database support in Django

Django has descent support for a multi DB apps. You can specify multiple databases in your settings like this.

```
DATABASES = {
    "default": {"ENGINE": "django.db.backends.sqlite3", "NAME": "default.db"},
    "thor": {"ENGINE": "django.db.backends.sqlite3", "NAME": "thor.db"},
    "potter": {"ENGINE": "django.db.backends.sqlite3", "NAME": "potter.db"},
}
```

Then, if you want to read `Polls` from the `thor` db, you can use `Poll.objects.using('thor').all()`.

This sort of works. But if we had to use `using` everywhere, the code duplication would quickly make our code unmanageable. We need a central place to define which database the tenant's DB requests should go to. Enter Django database routers.

## 4.2 Database routing in Django

Django allows hooking into the database routing process using the `DATABASE_ROUTERS` settings.

`DATABASE_ROUTERS` take a list of classes which must implement a few methods. A router class looks like this.

```
class CustomRouter:

    def db_for_read(self, model, **hints):
        return None

    def db_for_write(self, model, **hints):
        return None

    def allow_relation(self, obj1, obj2, **hints):
        return None

    def allow_migrate(self, db, app_label, model_name=None, **hints):
        return None
```

However, none of the methods in a Router class take request as an argument, which means there is no way for a router to call `tenant_db_from_request`. So we will need a way to pass the tenant data to the router.

## 4.3 Per tenant database routing using middlewares

We will use a middleware to calculate the DB to use. We will also need some way to pass it to the router. We are going to use a threadlocal variable to do this.

### 4.3.1 What are threadlocal variables?

Threadlocal variables are variables which you need to be accessible during the whole life-cycle of the thread, but you don't want it to be accessible or to leak between threads. threadlocal variables are discouraged in Django but they are a clean way for us to pass the data down the stack to the routers.

You create a threadlocal variable at the top of the module like this `_threadlocal = threading.local()`.

If you are using Python 3.7, you can also use contextvars instead of threadlocal variables.

### 4.3.2 The middleware class

With this discussion, our middleware class looks like this:

```
import threading

from django.db import connections
from .utils import tenant_db_from_request

THREAD_LOCAL = threading.local()


class TenantMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response
```

(continues on next page)

(continued from previous page)

```
    def __call__(self, request):
        db = tenant_db_from_request(request)
        setattr(THREAD_LOCAL, "DB", db)
        response = self.get_response(request)
        return response


def get_current_db_name():
    return getattr(THREAD_LOCAL, "DB", None)


def set_db_for_router(db):
    setattr(THREAD_LOCAL, "DB", db)
```

We have also added a few utility methods.

Now use these in your `settings.py`.

```
MIDDLEWARE = [
    # ...
    "tenants.middlewares.TenantMiddleware",
]
DATABASE_ROUTERS = ["tenants.router.TenantRouter"]
```

## 4.4 Outside the request response cycle

Our requests requests are now tenant aware, but we still need to run a few commands to finish our setup.

- We need to run migrations for all our databases
- We need to create a superuser to access the admin and create some objects

Most Django commands take a `--database=db_name` option, to specify which DB to run the command against. We can run the migrations like this.

```
python manage.py migrate --database=thor
python manage.py migrate --database=potter
```

However not all commands are multi-db aware, so it worthwhile writing a `tenant_context_manage.py`.

```
#!/usr/bin/env python
import os
import sys

from tenants.middlewares import set_db_for_router

if __name__ == "__main__":
    os.environ.setdefault("DJANGO_SETTINGS_MODULE", "pollsapi.settings")
    try:
        from django.core.management import execute_from_command_line
    except ImportError as exc:
        raise ImportError(
            "Couldn't import Django. Are you sure it's installed and "
            "available on your PYTHONPATH environment variable? Did you "
```

(continues on next page)

(continued from previous page)

```
            "forget to activate a virtual environment?"
        ) from exc
    from django.db import connection

    args = sys.argv
    db = args[1]
    with connection.cursor() as cursor:
        set_db_for_router(db)
        del args[1]
        execute_from_command_line(args)
```

It is slightly modified version of manage.py which takes the dbname as the first argument. We can run like this.

```
python tenant_context_manage.py thor createsuperuser --database=thor
```

With this we can add some `Poll` objects from the admin, and look at the API. It look like this.

Poll List

OPTIONS GET

```
GET /polls/

HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "choices": [],
        "question": "What is the weight of an unladen swallow?",
        "pub_date": "2018-07-05T11:20:55.605247Z",
        "created_by": 1
    }
]
```

Raw data HTML form

Question

Created by shabda

POST

In the next chapter, we will look at separating the tenants in their own docker containers. The code for this chapter is available at https://github.com/agiliq/building-multi-tenant-applications-with-django/tree/master/isolated-db

CHAPTER 5

# Completely isolated tenants using Docker

Until this chapter we have separated the tenant data, but the app server has been common between tenants. In this chapter, we will complete the separation using Docker, each tenant app code runs it own container and the tenant.

## 5.1 Tools we will use

- Docker to build the app code image and run the containers
- Docker-compose to define and run the containers for each tenant
- Nginx to route the requests to correct tenant container
- A separate Postgres database (Running inside a docker container) for each tenant
- A separate app server (Running inside a docker container) for each tenant

## 5.2 Building a docker image from our app code

As the first step we need to convert our app code to Docker image. Create a file named `Dockerfile`, and add this code.

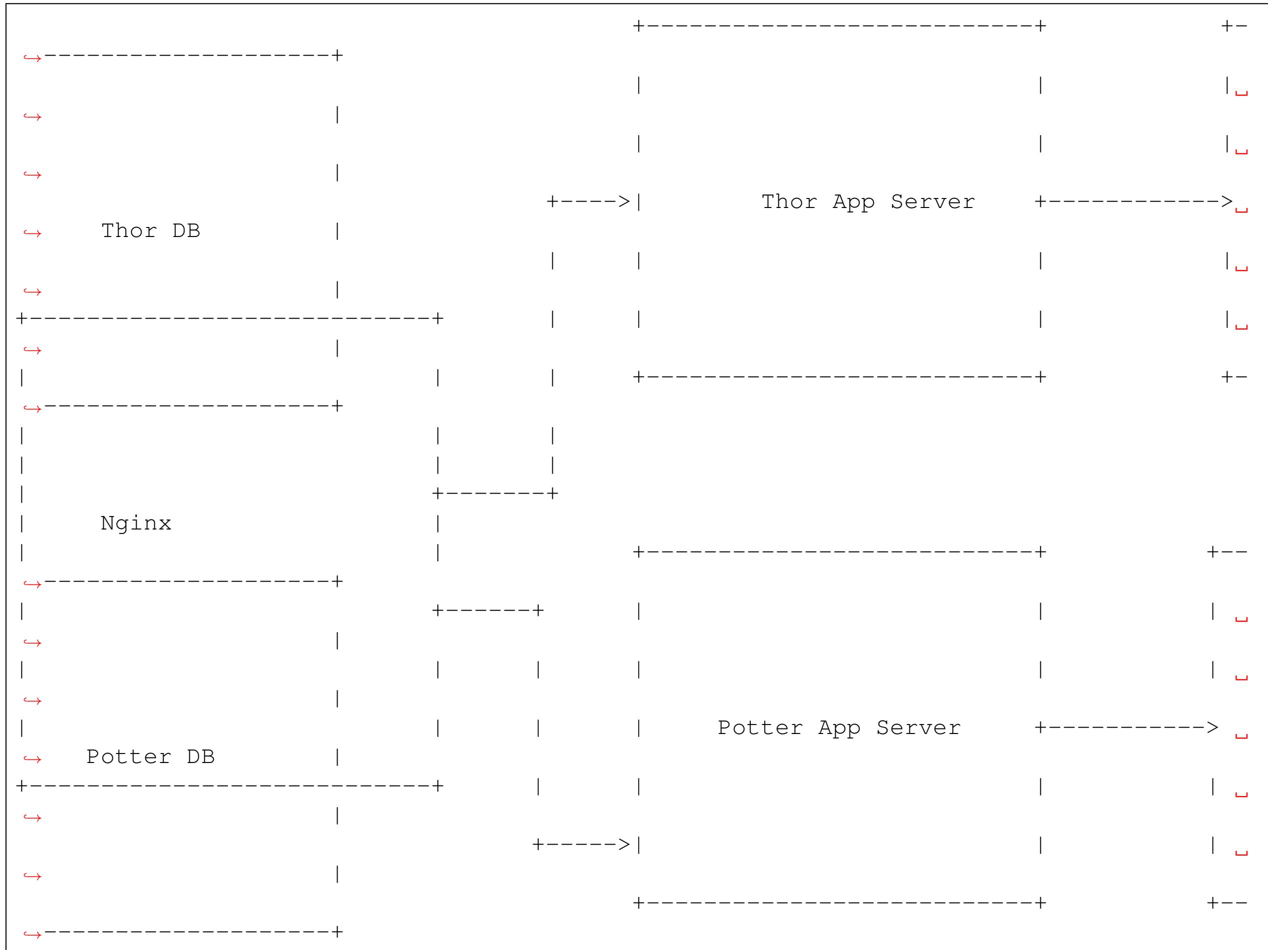
```
FROM python:3
ENV PYTHONUNBUFFERED 1
RUN mkdir /code
WORKDIR /code

# Install requirements
ADD requirements.txt /code/
RUN pip install -r requirements.txt


ADD . /code/
# We will specify the CMD in docker-compose.yaml
```

With this, run `docker build . -t agiliq/multi-tenant-demo`, to create an image and tag it as `agiliq/multi-tenant-demo`.

## 5.3 Using docker-compose to run multi container, multi-tenant apps

As in our previous chapters, we will have two tenants `thor` and `potter` at urls `potter.polls.local` and `thor.polls.local`.

The architecture looks something like this:




The containers we will be running are

- One nginx container
- 2 App servers, one for each tenant
- 2 DB servers, one for each tenant
- Transient containers to run `manage.py migrate`

## 5.4 The final docker-compose.yaml

With our architecture decided, our docker-compose.yaml looks like this

```yaml
version: '3'

services:
  nginx:
    image: nginx:alpine
    volumes:
        - ./nginx.conf:/etc/nginx/nginx.conf
    ports:
        - "8080:80"
    depends_on:
        - thor_web
        - potter_web

  # Thor
  thor_db:
    image: postgres
    environment:
      - POSTGRES_PASSWORD=thor
      - POSTGRES_USER=thor
      - POSTGRES_DB=thor
  thor_web:
    image: agiliq/multi-tenant-demo
    command: python3 manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/code
    depends_on:
      - thor_db
    environment:
      - DATABASE_URL=postgres://thor:thor@thor_db/thor

  thor_migration:
    image: agiliq/multi-tenant-demo
    command: python3 manage.py migrate
    volumes:
      - .:/code
    depends_on:
      - thor_db
    environment:
      - DATABASE_URL=postgres://thor:thor@thor_db/thor

  # Potter
  potter_db:
    image: postgres
    environment:
      - POSTGRES_PASSWORD=potter
      - POSTGRES_USER=potter
      - POSTGRES_DB=potter
  potter_web:
    image: agiliq/multi-tenant-demo
    command: python3 manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/code
    depends_on:
      - potter_db
    environment:
      - DATABASE_URL=postgres://potter:potter@potter_db/potter
```

(continues on next page)

(continued from previous page)

```
  potter_migration:
    image: agiliq/multi-tenant-demo
    command: python3 manage.py migrate
    volumes:
      - .:/code
    depends_on:
      - thor_db
    environment:
      - DATABASE_URL=postgres://potter:potter@potter_db/potter
```

Let's look at each of the components in detail.

### 5.4.1 Nginx

The `nginx` config in our `docker-compose.yaml` looks like this,

```
nginx:
  image: nginx:alpine
  volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf
  ports:
      - "8080:80"
  depends_on:
      - thor_web
      - potter_web
```

And `nginx.conf` look like this

```
events {
    worker_connections  1024;
}

http {
    server {
        server_name  potter.polls.local;
        location / {
            proxy_pass      http://potter_web:8000;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
        }
    }

    server {
        server_name  thor.polls.local;
        location / {
            proxy_pass      http://thor_web:8000;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
        }
    }
}
```

In our nginx config, we are doing a proxypass to appropriate container, using `proxy_pass http://potter_web:8000;`, based on the host header. We also need to set the `Host` header, so Django can enforce its `ALLOWED_HOSTS`.

### 5.4.2 DB and the App containers

Let's look at the app containers we have launched for `thor`. We will launch similar containers for tenants.

```
thor_db:
  image: postgres
  environment:
    - POSTGRES_PASSWORD=thor
    - POSTGRES_USER=thor
    - POSTGRES_DB=thor
thor_web:
  image: agiliq/multi-tenant-demo
  command: python3 manage.py runserver 0.0.0.0:8000
  volumes:
    - .:/code
  depends_on:
    - thor_db
  environment:
    - DATABASE_URL=postgres://thor:thor@thor_db/thor
```

We are launching a standard postgres container with customized DB name and credentials. We are then running our Django code and passing the credentials as DB name to he container using `DATABASE_URL` environment variable. Our app set the db connection using `dj_database_url.config()` which reads from `DATABASE_URL`.

### 5.4.3 Running migrations and creating a superuser

We want to run our migrations for each DB as part of the deployment process, we will add container which does this.

```
thor_migration:
    image: agiliq/multi-tenant-demo
    command: python3 manage.py migrate
    volumes:
      - .:/code
    depends_on:
      - thor_db
    environment:
      - DATABASE_URL=postgres://thor:thor@thor_db/thor
```

This container will terminate as soon as migrations are done.

We also need to create a superuser. You can do this by `docker exec` ing to the running app containers.

Do this *docker exec -it <containet_name> bash*. (You can get the container name by running `docker ps`). Now you have a bash shell inside the container. Create your superuser in the usual way using `manage.py createsuperuser`.

You can now access the thor tenant as `thor.polls.local:8080` and potter at `potter.polls.local:8080`. After adding a `Poll`, my tenant looks like this.

Django REST framework shabda

Api Root / Poll List

Poll List

OPTIONS GET

```
GET /polls/
```

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "choices": [],
        "question": "Who is the best avenger?",
        "pub_date": "2018-07-06T18:09:50.279375Z",
        "created_by": 1
    }
]
```

Raw data HTML form

Question

Created by shabda

POST

The code for this chapter is available at https://github.com/agiliq/building-multi-tenant-applications-with-django/tree/master/isolated-docker

CHAPTER 6

# Tying it all together

## 6.1 Launching new tenants

In the previous chapters, we have worked with a hardcoded list, of two tenants, `thor` and `potter`. Our code looked like this

code-block:: python

> **def get_tenants_map():** return {"thor.polls.local": "thor", "poter.polls.local": "potter"}

In a real scenario, you will need to launch tenants, so the list of tenants can't be part of the code. To be able to launch new tenants, we will create a `Tenant` model.

code-block:: python

> **class Tenant(models.Model):** name = models.CharField(max_length=100) schema_name = models.CharField(max_length=100) subdomain = models.CharField(max_length=1000, unique=True)

And your `get_tenants_map` will change to:

code-block:: python

> **def get_tenants_map():** return dict(Tenant.objects.values_list("subdomain", "schema_name"))

You would need to make similar changes for a multi DB setup, or orchestrate launching new containers and updating nginx config for multi container setup.

## 6.2 A comparison of trade-offs of various methods

Until now, we had looked at four different ways of doing multi tenancy, each with some set of trade-offs.

Depending upon how many tenants you have, how many new tenants you need to launch, and your customization requirements, one of the four architectures will suit you.

| Method | Isolation | Time to launch new tenants | Django DB Compatibility |
|---|---|---|---|
| Shared DB and Schema | Low | Low | High (Supported in all DBs) |
| Isolated Schema | Medium | Low | Medium (DB must support schema) |
| Isolated DB | High | Medium | High (Supported in all DBs) |
| Isolated using docker | Complete | Medium | High (Supported in all DBs) |

## 6.3 What method should I use?

While each method has its pros and cons, for most people, Isolated Schema with shared database is the best method. It provides strong isolation guarantees, customizability with minimal time to launch new tenants.

# CHAPTER 7

# Third party apps

## 7.1 Open source Django multi tenancy apps

There are number of third party Django apps which add multi tenancy to Django.

Some of them are

- Django multitenant: https://github.com/citusdata/django-multitenant (Shared SChema, Shared DB, Tables have `tenant_id`)
- Django tenant schemas: https://github.com/bernardopires/django-tenant-schemas (Isolated Schemas, shared DB)
- Django db multitenant: https://github.com/mik3y/django-db-multitenant (Isolated DB)

We will look in detail at Django tenant schemas, which is our opinion is the most mature of the Django multi tenancy solutions.

## 7.2 A tour of django-tenant-schemas

Install `django-tenant-schemas` using pip. `pip install django-tenant-schemas`. Verify the version of django-tenant-schemas that got installed.

```
$ pip freeze | grep django-tenant-schemas
django-tenant-schemas==1.9.0
```

We will start from our non tenant aware Polls app and add multi tenancy using django-tenant-schemas.

Create a new database, and make sure your Django app picks up the new DB by updating the `DATABASE_URL` environment var.

Update your settings to use the tenant-schemas `DATABASE_BACKEND` and tenant-schemas `DATABASE_ROUTERS`

```
DATABASES["default"]["ENGINE"] = "tenant_schemas.postgresql_backend"
# ...
DATABASE_ROUTERS = ("tenant_schemas.routers.TenantSyncRouter",)
```

The `postgresql_backend` will ensure that the connection has the correct `tenant` set, and the `TenantSyncRouter` will ensure that the migrations run correctly.

Then create a new app called `tenants` with `manage.py startapp`, and create a new `Client` model

```
from tenant_schemas.models import TenantMixin


class Client(TenantMixin):
    name = models.CharField(max_length=100)
```

In your settings, change the middleware and set the `TENANT_MODEL`.

```
TENANT_MODEL = "tenants.Client"
# ...

MIDDLEWARE = [
    "tenant_schemas.middleware.TenantMiddleware",
    # ...
    ]
```

`tenant-schemas` comes with the concept of `SHARED_APPS` and `TENANT_APPS`. The apps in `SHARED_APPS` have their tables in public schema, while the apps in `TENANT_APPS` have their tables in tenant specific schemas.

```
SHARED_APPS = ["tenant_schemas", "tenants"]

TENANT_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "rest_framework",
    "rest_framework.authtoken",
    "polls",
]
```

INSTALLED_APPS = SHARED_APPS + TENANT_APPS

We are almost done. We need to

- Run the migrations in the public schema
- Create the tenants and run migrations in all the tenant schemas
- Create a superuser in tenant schemas

`tenant-schemas` has the `migrate_schemas` which replaces the `migrate` command. It is tenant aware and will sync `SHARED_APPS` to public schema, and `TENANT_APPS` to tenant specific schemas.

Run `python manage.py migrate_schemas --shared` to sync the public tables.

The run a python shell using `python manage.py shell`, and create the two tenants, using

```
Client.objects.create(name="thor",
    schema_name="thor", domain_url="thor.polls.local")
Client.objects.create(name="potter",
    schema_name="potter", domain_url="potter.polls.local")
```

This will create the schemas in the table and run the migrations. You now need to create the superuser in the tenant schema so that you can access the admin. The `tenant_command` command allow running any Django command in the context of any tenant.

```
   python manage.py tenant_command createsuperuser

And we are done. We can now access the tenant admins, create polls and view the␣
↪tenant specific API endpoints.
```

The code for this chapter is available at https://github.com/agiliq/building-multi-tenant-applications-with-django/tree/master/tenant-schemas-demo .

CHAPTER 8

# Indices and tables

- genindex
- modindex
- search