Deployment Guide: Django Multi-Tenant Chatbot on Render

1. Architecture Overview

This SaaS chatbot platform uses **PostgreSQL schemas per client** with `django-tenants` so each customer's data is logically isolated while sharing one codebase and one database instance.[1]
The **public schema** stores shared objects (tenant metadata, auth tables, global configuration), while each **tenant schema** (one per client) stores chatbot configuration, conversations, and analytics for that specific tenant.[1]

In production, requests pass through `HeaderTenantMiddleware`, which reads an identifier (for example, an `X-Client-ID` header) to select the correct tenant and adjust the PostgreSQL `search_path` to that schema.[2]
This approach avoids separate databases per customer, simplifies scaling, and keeps tenant-aware logic in the database router and middleware instead of in every view.[1]

2. Production Settings Refactor

This section shows the **exact settings changes** needed to move from local development to a Render-ready production configuration, based on `settings-copy.py` and the documentation brief.[3][2]

2.1 Security, Hosts, and Core Settings

```
from pathlib import Path
import os
import dj_database_url

BASE_DIR = Path(__file__).resolve().parent.parent

# ---------------------------------------------------------------------
# Security & Debug
# ---------------------------------------------------------------------

# Never hard-code the secret key in production
SECRET_KEY = os.environ.get("SECRET_KEY")
```

```
# Toggle debug via environment variable
DEBUG = os.environ.get("DEBUG", "False") == "True"

# Allow Render’s domains plus local development
ALLOWED_HOSTS = [
    ".onrender.com",
    "localhost",
    "127.0.0.1",
]
```

Why this matters

- Moving `SECRET_KEY` and `DEBUG` to environment variables prevents leaking secrets in the repo and allows different behavior between local and production environments.[3]
- `ALLOWED_HOSTS` must include the Render URL (e.g. `your-app-name.onrender.com`) or Django will return 400 errors for valid requests.[3]

2.2 Installed Apps, Tenants, and Middleware (for context)

```
SHARED_APPS = [
    "django_tenants",
    "corsheaders",
    "tenants",
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "rest_framework",
    "rest_framework.authtoken",
    "drf_spectacular",
    "dj_rest_auth",
    "dashboard",
]

TENANT_APPS = [
    "chatbot",
]
```

```
INSTALLED_APPS = list(set(SHARED_APPS + TENANT_APPS))

TENANT_MODEL = "tenants.Client"
TENANT_DOMAIN_MODEL = "tenants.Domain"

DATABASE_ROUTERS = ("django_tenants.routers.TenantSyncRouter",)

MIDDLEWARE = [
    "corsheaders.middleware.CorsMiddleware",
    "tenants.middleware.HeaderTenantMiddleware",
    "django.middleware.security.SecurityMiddleware",
    "django.contrib.sessions.middleware.SessionMiddleware",
    "django.middleware.common.CommonMiddleware",
    "django.middleware.csrf.CsrfViewMiddleware",
    "django.contrib.auth.middleware.AuthenticationMiddleware",
    "django.contrib.messages.middleware.MessageMiddleware",
    "django.middleware.clickjacking.XFrameOptionsMiddleware",
]
```

These are consistent with your current architecture and ensure `django-tenants` and the header-based routing are active in production.[2]

2.3 Database Configuration with `dj_database_url`

```
# Default local URL for development
DEFAULT_DB_URL = "postgres://postgres:postgres@localhost:5432/chatterplatform"

DATABASES = {
    "default": dj_database_url.config(
        default=os.environ.get("DATABASE_URL", DEFAULT_DB_URL),
        conn_max_age=600,
    )
}

# Ensure the correct engine for django-tenants
DATABASES["default"]["ENGINE"] = "django_tenants.postgresql_backend"
```

Why switch from hard-coded DATABASES to `dj_database_url`

- Render exposes the PostgreSQL credentials as a single `DATABASE_URL` environment variable derived from the database resource, not as individual NAME/USER/HOST settings.[3]
- `dj_database_url.config` parses that URL and builds the correct `DATABASES["default"]` dictionary, enabling the same settings file to work locally (with `DEFAULT_DB_URL`) and in production (with `DATABASE_URL`).[4][3]
- Explicitly forcing the `ENGINE` to `django_tenants.postgresql_backend` guarantees tenant routing even if the URL does not specify an engine.[2]

2.4 Static Files for Render

```
STATIC_URL = "static/"

# Required so collectstatic knows where to put files
STATIC_ROOT = os.path.join(BASE_DIR, "staticfiles")

# Keep your existing additional static directories if any
STATICFILES_DIRS = [
    os.path.join(BASE_DIR, "widget"),
]
```

Why STATIC_ROOT is necessary

- Render's recommended flow runs `python manage.py collectstatic` during the build; without `STATIC_ROOT`, Django raises errors or skips collection, and your CSS/JS will not be served.[5][3]
- A dedicated `staticfiles` directory is standard practice and works well with WhiteNoise and Render's filesystem.[6][3]

3. Production Requirements (requirements.txt)

Use a **focused** `requirements.txt` that contains everything needed for the deployed app but omits development-only tools like Jupyter and matplotlib.[7][2]

```
# Core framework
Django==5.2.7
asgiref==3.10.0
sqlparse==0.5.3
```

```
pytz==2025.2
tzdata==2025.2

# Multi-tenancy & API
django-tenants==3.9.0
djangorestframework==3.16.1
django-cors-headers==4.9.0
dj-rest-auth==7.0.1
drf-spectacular==0.28.0
inflection==0.5.1

# Database & configuration for Render
psycopg2-binary==2.9.11
dj-database-url==2.1.0

# Auth & security utilities
argon2-cffi==25.1.0
argon2-cffi-bindings==25.1.0

# Static files & production server
whitenoise==6.7.0
gunicorn==23.0.0

# Chatbot / NLP stack (minimal for the shared engine)
requests==2.32.5
sentence-transformers==5.1.1
transformers==4.57.1
torch==2.9.0
scikit-learn==1.7.2
scipy==1.16.2
numpy==2.3.4
pandas==2.3.3
tqdm==4.67.1
regex==2025.10.23

# Payments (Task 2 – enable when needed)
razorpay==2.0.0
```

- `psycopg2-binary`, `dj-database-url`, `gunicorn`, and `whitenoise` are commonly recommended for Django on Render with PostgreSQL and static files.[8][5][3]
- The NLP dependencies match your chatbot architecture and can be pruned if you later move the model to a separate service.[9][1]

4. Build Script (build.sh) for Render

Render needs a build command that installs dependencies, collects static files, and runs tenant-aware migrations.[3]

Create `build.sh` in the project root:

```
#!/usr/bin/env bash
set -o errexit  # exit on error

# 1. Install dependencies
pip install -r requirements.txt

# 2. Collect static files into STATIC_ROOT
python manage.py collectstatic --no-input

# 3. Run django-tenants migrations for shared apps
python manage.py migrate_schemas --shared
```

Make it executable:

```
chmod a+x build.sh
```

Configure Render's **Build Command** to:

```
./build.sh
```

Why migrate_schemas --shared instead of migrate

- In a django-tenants project, `migrate` is no longer sufficient because it only targets the default schema; it is unaware of per-tenant schemas and routing.[2][3]

- `migrate_schemas` understands `SHARED_APPS` and `TENANT_APPS`, creating the **public schema** and applying shared migrations to it.[2]
- Using the `--shared` flag in the build step ensures that all global/shared tables (auth, tenants, dashboard, etc.) exist in the public schema before the app first starts.[3]
- Tenant-specific migrations (without `--shared`) can be run later if you add tenants or introduce schema-specific changes.

5. Render Dashboard Configuration

5.1 Create the PostgreSQL Database

1. In Render, click **New → PostgreSQL**.[3]
2. Name: **chatter-db** (or similar).
3. Optionally set **Database Name** to `chatterplatform` to mirror your local configuration.[3]
4. Choose a region close to your primary users (for example, Frankfurt or Mumbai).[3]
5. Create the database and wait for it to provision.
6. Copy the **Internal Database URL**; you will use this as `DATABASE_URL` for the web service.[3]

5.2 Create the Django Web Service

1. Click **New → Web Service** and connect your GitHub repository that contains the Django project.[3]
2. Name: `multi-tenant-chatbot` (or any descriptive name).[3]
3. Runtime: **Python 3** (use the version you run locally, for example `3.11`).[3]
4. **Build Command**: `./build.sh`.
5. **Start Command**:

   ```
   gunicorn config.wsgi:application
   ```

   Replace `config` with your Django project package name if different.[3]
6. Choose an instance type (Free for testing, Starter or higher for production traffic).[3]

5.3 Environment Variables

In the Web Service → **Environment** tab, add:

Key	Example value	Purpose
`PYTHON_VERSION`	`3.11.5`	Matches your local Python version to avoid subtle package issues.[3]
`DATABASE_URL`	`postgres://…` (Internal DB URL from database)	Parsed by `dj_database_url` to configure `DATABASES["default"]`.[3]
`SECRET_KEY`	A long random string	Used for cryptographic signing; never commit this.[3]
`DEBUG`	`False`	Ensures production behavior (no debug pages).[3]
`DISABLE_COLLECTSTATIC`	`0` or leave unset	You **want** collectstatic to run, because `build.sh` calls it.[3]
Any third-party keys (e.g. `RAZORPAY_KEY_ID`, `RAZORPAY_KEY_SECRET`)	As needed	Keep all secrets out of the codebase.[2]

This configuration lets you manage all environment-specific values directly from Render's UI without editing code between environments.[3]

6. Post-Deployment Initialization: Public Tenant

6.1 Why the App Crashes on First Deploy

After the first build, the database tables exist, but **no public tenant row** exists in `tenants_client` or `tenants_domain`.[3]
`HeaderTenantMiddleware` attempts to resolve the incoming host/header to a tenant; without a matching entry, schema selection fails and Django returns 500 errors.[2]

6.2 Creating the Public Tenant via Shell

1. Open your Web Service in Render and wait until it shows as **Live**.

2. Go to the **Shell** tab.

3. Start a Django shell:

```
python manage.py shell
```

4. In the Python shell, run:

```
from tenants.models import Client, Domain

# 1. Create the public tenant
tenant = Client(
    schema_name="public",
    name="Public Tenant",
)
tenant.save()

# 2. Attach the Render domain
domain = Domain(
    domain="your-app-name.onrender.com",  # replace with actual Render URL
    tenant=tenant,
    is_primary=True,
)
domain.save()
```

5. Exit the shell with `exit()` or `Ctrl+D`.

Now, requests to `your-app-name.onrender.com` resolve to the `public` schema, and `django-tenants` can route tenant-aware traffic correctly.[2][3]

7. Developer Checklist (Quick Reference)

- **Before pushing to GitHub**
    - Refactor `settings.py` as shown: environment-based `SECRET_KEY`, `DEBUG`, and `DATABASES` using `dj_database_url`; add `STATIC_ROOT`; update `ALLOWED_HOSTS` to include Render.[2][3]
    - Create `build.sh` and make it executable; ensure `requirements.txt` includes Django, `django-tenants`, REST stack, `psycopg2-binary`, `dj-database-url`, `gunicorn`, and `whitenoise`, plus NLP and payment libraries as needed.[2][3]

- **On Render**
    - Create the PostgreSQL database and copy its **Internal Database URL** as `DATABASE_URL`.[3]
    - Create the Web Service pointing at your repo with `./build.sh` as Build Command and `gunicorn config.wsgi:application` as Start Command.[3]
    - Configure environment variables (`SECRET_KEY`, `DEBUG`, `DATABASE_URL`, etc.).[3]
- **After first deploy**
    - Use `python manage.py shell` from Render to create the **public tenant** and its **domain** entry as shown above so that tenant resolution works and the app becomes accessible.[2][3]

Following this guide, you will have a clean, reproducible pipeline from local development to a multi-tenant Django deployment on Render, with clear separation between public and tenant schemas and a solid foundation for scaling the chatbot platform.

⁂

1. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/22098629/215927dc-4d90-4c22-9e06-1991a7eaed0e/Tasks-Chatbot-Developer-Requirements-Architecture-Document.pdf
2. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/22098629/af2d634e-0d41-4e3e-aab3-f0af4ac68c27/settings-copy.py
3. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/22098629/3932b963-00e2-4a5d-9546-7a648eb26163/documentation.pdf
4. https://pypi.org/project/dj-database-url/
5. https://docs.render.com/deploy-django
6. https://render.com/docs/deploy-django
7. https://forum.djangoproject.com/t/best-practices-managing-requirements-txt/10353
8. https://testdriven.io/blog/django-render/
9. https://sbert.net
10. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/22098629/18e6bad4-e677-4eb7-b529-d4f68ffbc462/image.jpg

11. https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/22098629/b29fda01-4145-4adc-99ca-99c7850cbecf/image.jpg