Exercise 3: Django Multitenant Application

This is my proposal for implementing a multi-tenant Django application. I'll focus on a simple solution and explain some of the challenges in maintaining it.

Abstract

The main goal of a multi-tenant application is data security and isolation: a tenant must only be able to access their own data, while the system remains scalable and easy to manage.

In this proposal, the selected approach is a **Shared Database, Separate Schema** architecture [1]. This provides a good balance of security, performance, and cost. It's implemented using the `django-tenants` library [2], which leverages PostgreSQL's native schema support.

Multi-tenancy approaches

There are three primary ways to design a multi-tenant database, each with significant trade-offs:

- **Separate Databases**: This is the most secure approach, giving each tenant their own database. It offers maximum isolation but comes with very high operational costs and complexity. Managing migrations, backups, and connections for hundreds or thousands of databases is a major challenge.
- **Shared Database, Shared Schema**: This is the simplest model, where all tenants share the same tables, and a tenant_id column on each row separates the data. While easy to start with, it offers the weakest isolation. A single programming error in a query could easily lead to a serious data leak. Also, as the number of tenats and operations grows, queries and tables need to extremely optimized.
- **Shared Database, Separate Schemas**: This hybrid model uses a single PostgreSQL database, but each tenant gets their own private *schema*, a logical grouping of tables. This leverages a powerful feature of PostgreSQL to provide strong, database-level data isolation without the complexity of managing separate databases. This is our recommended path.

Tools and infrastructure

The following is the proposed technology stack, a modern and proven one.

Hosting

Google Cloud Run [3], as it offers a wide range of options for scalability and monitoring, making it easy to run containers. Additionally, its network configuration can be customized to handle tenant-specific subdomains like `tenant_a.prowler.com`, or custom domains like `prowler.tenant_a.com`.

Database

PostgreSQL, managed via **Google Cloud SQL** [4], offers easy backup management, high availability, and scalability.

Structure

- **Public schema**: Contains shared information, such as tenant account details and billing subscriptions.
- **Tenant schemas**: Handle each tenant's private information, such as their users, scan results, or reports.

Framework:

Django and **Django REST framework**, which form the core of the Prowler API.

Webserver

Uvicorn [5], for its support of asynchronous API calls [6] and database access [7].

Multi-tenant management:

`django-tenants`, the key of this approach. It uses PostgreSQL's `search_path` [8] to automatically manage tenant schemas, making data isolation transparent to the application code.

Challenges

Although this approach is robust, there are several complexities we must acknowledge and prepare for, all of which I've encountered before.

- **Migrations**: Since the data model is replicated in each tenant's schema, running migrations across all tenants can be very slow and risky; a single failure could halt the entire process. Therefore, the migration process must be reversible or resumable, thoroughly tested in various environments, and ideally rolled out using canary deployments to a small group of tenants first, versioning the API.
- **Backup and recovery**: Recovering a backup for a single tenant can be difficult without affecting other tenants. A robust, automated recovery strategy is essential, as the alternatives (like recovering the entire database, performing a manual recovery our using the Separata Database approach) are often impractical.
- **Cross-tenant analytics**: Performing analytics and running aggregations across multiple schemas is inefficient. Potential solutions include writing

event metadata to a shared analytics schema in real-time or running periodic ETL processes to create a unified dataset for analysis.

- **Tenant lifecycle**: The onboarding and offboarding of tenants (which includes tasks like creating and dropping their schemas) must be automated and atomic to prevent data inconsistencies and errors.
- **Noisy neighbor** [9]: A tenant could overuse the application, creating performance problems that affect other tenants, especially in the database. Solutions like API rate limiting can be implemented, but this comes with the trade-off of a more limited user experience. *Note*: I didn't know the name of this one, although I've suffered it.

References

1. Bytebase - Multi-Tenant Database Architecture Patterns Explained: https://www.bytebase.com/blog/multi-tenant-database-architecture-patterns-explained/
2. django-tenants: https://github.com/django-tenants/django-tenants
3. Google Cloud Run: https://cloud.google.com/run
4. Google Cloud SQL: https://cloud.google.com/sql
5. Uvicorn: https://www.uvicorn.org/
6. adrf - Async Django REST framework: https://github.com/em1208/adrf
7. Django - Asynchronous support - Queries & the ORM: https://docs.djangoproject.com/en/5.2/topics/async/#queries-the-orm
8. PostgreSQL - Schemas - The Schema Search Path: https://www.postgresql.org/docs/current/ddl-schemas.html#DDL-SCHEMAS-PATH
9. Microsfot Learn - Noisy Neighbor antipattern: https://learn.microsoft.com/en-us/azure/architecture/antipatterns/noisy-neighbor/noisy-neighbor
10. University of Pittsburgh Library System - Citation Styles: APA, MLA, Chicago, Turabian, IEEE: https://pitt.libguides.com/citationhelp/ieee

Submission requisite

As a PDF file was requested, I generated a `README.pdf` version of this Markdown file using Pandoc:

```
pandoc README.md -o README.pdf
```

Note

As the submission is a PDF file, I chose to use IEEE citation style [10].