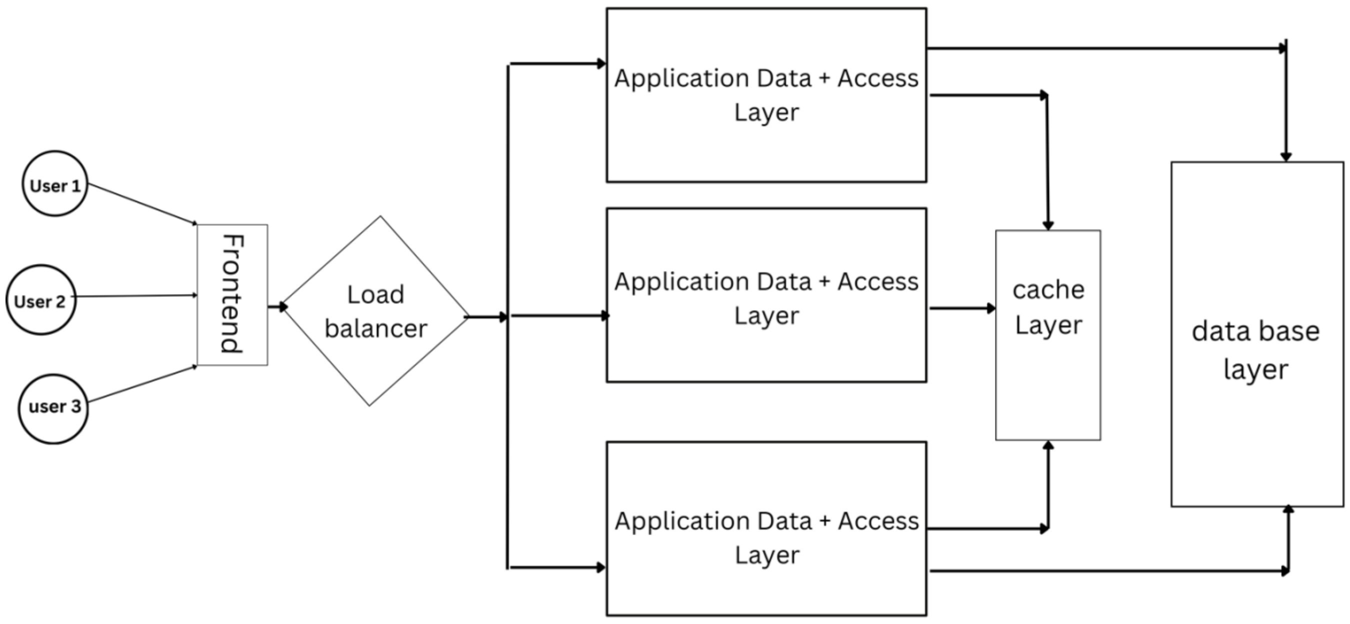
Backend Layered Architecture Documentation

Overview

This documentation outlines a layered backend architecture using:

- **Flask**: For the application/service layer with session-based authentication.
- **SQLAlchemy**: For the data access layer.
- **PostgreSQL**: As the database with persistent storage.
- **NGINX:** As load balancer. (Optional)
- **KeyDB**: As the caching layer for improved performance.

The backend instances and database run in separate Docker containers to ensure modularity, scalability, and ease of deployment. Persistent storage is configured for the database container.


User 1
User 2
user 3
Frontend
Load balancer
Application Data + Access Layer
Application Data + Access Layer
Application Data + Access Layer
cache Layer
data base layer

Architecture Design

1. **Application/Service Layer**:

    - Built with Flask, running as individual Docker containers.
    - Uses session-based authentication.
    - Implements business logic and handles requests from the frontend (handled separately).

2. **Data Access Layer**:

    - SQLAlchemy handles interactions with the database (PostgreSQL).
    - Follows an ORM approach to ensure abstraction from raw SQL queries.

3. **Caching Layer**:

    - KeyDB is used for caching.
    - Enables faster access to frequently used data (e.g., session data, query results).
    - Multi-threaded architecture allows handling high-concurrency workloads.
    - **Redis-compatible**, meaning you can use Redis clients, libraries, and commands to interact with KeyDB seamlessly.

4. **Database Layer**:

    - PostgreSQL container with persistent storage using a Docker volume.
    - Ensures durability and data integrity.

Key Components

1. **Docker Setup**:

    - Separate containers for:
        - Flask application (per instance).
        - KeyDB caching server.
        - PostgreSQL database server with persistent storage.
    - Docker Compose is used to orchestrate these services.

2. **Caching with KeyDB**:

    - Stores session data and query results to reduce load on the database.
    - Implements time-to-live (TTL) for cache entries to maintain freshness.
    - Active-Active replication is optional for distributed caching in high-availability setups.

3. **Persistent Database**:

    - PostgreSQL stores all application data with durability.
    - Regular backups can be configured using tools like `pg_dump` or custom scripts.

File Structure: For Flask

```
Root/
├── src/
|   ├── dbModels/       # SQLAlchemy definitions
|   ├── flasky/         # Flask setup, including CORS, LoginManager, etc
|   └── security        # Security code like hashing and encryption
|
├── scripts/            # Linux build scripts
├── .gitignore
├── README.md
├── LICENSE
├── main.py             # Main entry point for the app
└── requirements.txt    # Dependencies
```

Sample docker-compose.yml

```
version: '3.9'
```

```
services:
  app:
    build: ./scripts/docker
    ports:
      - "5000:5000"
    environment:
      - FLASK_ENV=production
    depends_on:
      - keydb
      - postgres
    volumes:
      - ./app:/usr/src/app

  keydb:
    image: eqalpha/keydb
    container_name: keydb
    ports:
      - "6379:6379"
    volumes:
      - keydb-data:/data

  postgres:
    image: postgres:15
    container_name: postgres
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: appdb
    ports:
      - "5432:5432"
    volumes:
      - pg-data:/var/lib/postgresql/data

volumes:
  pg-data:
  keydb-data:
```

Caching with KeyDB

Configuration

- Add the following `keydb.conf` file for custom KeyDB settings:

```
multi-threading yes
active-replica yes
maxmemory 256mb
maxmemory-policy allkeys-lru
```

Integration with Flask

In `cache.py`:

```
import redis

cache = redis.StrictRedis(host='keydb', port=6379, decode_responses=True)

# Example: Set and get cached data
def set_cache(key, value, ttl=3600):
    cache.setex(key, ttl, value)

def get_cache(key):
    return cache.get(key)
```

Application Initialization

In __init__.py:

```
from flask import Flask
from .cache import cache
from .models import db

def create_app():
    app = Flask(__name__)
    app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://user:password@postgres/appdb'
    app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
    app.secret_key = 'supersecretkey'

    # Initialize database
    db.init_app(app)

    # Ensure KeyDB connection
    cache.ping()

    return app
```

Security Features

- **Session-based Authentication**:
  - Flask sessions store user login data.
  - Cached session data in KeyDB for scalability.
- **Environment Variables**:
  - Sensitive data like database credentials are stored in `.env`.

Monitoring and Scaling

- **Scaling**:
  - Deploy multiple Flask instances for horizontal scaling.
  - KeyDB can handle concurrent reads/writes efficiently.
- **Monitoring**:
  - Use Prometheus or Grafana for KeyDB and PostgreSQL metrics.

Benefits

- **Improved Performance**: KeyDB's multi-threaded architecture ensures faster caching.
- **Scalability**: Separate containers allow horizontal scaling of application and caching layers.
- **Persistence**: PostgreSQL ensures reliable data storage.

Let me know if you need further refinements!