Django Backend Architecture Documentation

Exchange Rate Flutterwave Application

Table of Contents

Overview

This Django backend serves as a real-time exchange rate API that: - Fetches exchange rates from Flutterwave API - Caches rates in Redis for fast access - Stores rates in SQLite database for persistence - Broadcasts rate updates via WebSockets in real-time - Provides REST API endpoints for rate queries

The architecture is designed for **instant response times** with no loading spinners, using a multi-layer caching strategy and real-time updates.

Project Structure

```
server/
├── backend/              # Main Django project configuration
│   ├── settings.py      # Django settings and configuration
│   ├── asgi.py          # ASGI application (WebSocket + HTTP)
│   ├── urls.py          # Root URL routing
│   └── wsgi.py          # WSGI application (traditional HTTP)
│
├── rates/               # Main application for exchange rates
│   ├── models.py        # Database models (ExchangeRate)
│   ├── views.py         # REST API endpoints
│   ├── services.py      # Business logic (Flutterwave API calls)
│   ├── cache.py         # Redis caching utilities
│   ├── consumers.py     # WebSocket consumers (real-time updates)
│   ├── routing.py       # WebSocket URL routing
│   ├── urls.py          # REST API URL patterns
│   └── management/
│       └── commands/
│           └── poll_rates.py  # Background rate polling command
│
├── manage.py            # Django management script
├── requirements.txt     # Python dependencies
└── db.sqlite3          # SQLite database file
```

Core Components Explained

Backend Configuration

`backend/settings.py`

Purpose: Central configuration file for the entire Django project.

Key Responsibilities: - **INSTALLED_APPS**: Registers all Django apps including `rates`, `channels`, `rest_framework`, `corsheaders` - **ASGI_APPLICATION**: Points to `backend.asgi.application` to enable WebSocket support - **CHANNEL_LAYERS**: Configures Redis as the channel layer for WebSocket communication - **CACHES**: Configures Redis as the caching backend (different Redis DB from channels) - **DATABASES**: Configures SQLite as the database - **CORS_ALLOW_ALL_ORIGINS**: Allows cross-origin requests (for Flutter app) - **FLUTTERWAVE_SECRET_KEY**: Stores the API key for Flutterwave

Why it matters: This file ties everything together. Without proper configuration here, WebSockets won't work, caching won't work, and the app won't connect to external services.

`backend/asgi.py`

Purpose: ASGI (Asynchronous Server Gateway Interface) application entry point.

Key Responsibilities: - **ProtocolTypeRouter**: Routes HTTP and WebSocket traffic to different handlers - **HTTP requests**: Handled by traditional Django WSGI application - **WebSocket connections**: Routed to `rates.routing.websocket_urlpatterns` - **AuthMiddlewareStack**: Adds authentication support to WebSocket connections

How it works:

```
application = ProtocolTypeRouter({
    "http": django_asgi_app,           # Regular HTTP requests
    "websocket": AuthMiddlewareStack(   # WebSocket connections
        URLRouter(rates.routing.websocket_urlpatterns)
    ),
})
```

Why it matters: This is what enables real-time WebSocket communication. Without ASGI, you can only use traditional HTTP requests (which require polling).

`backend/urls.py`

Purpose: Root URL routing configuration.

Key Responsibilities: - Maps `/admin/` to Django admin interface - Maps `/api/` to the `rates` app URLs

URL Structure: - `/admin/` → Django admin - `/api/rates/` → Single rate endpoint - `/api/rates/all/` → All rates endpoint - `/ws/rates/` → WebSocket endpoint (defined in `rates/routing.py`)

Rates App Components

`rates/models.py`

Purpose: Defines the database schema for storing exchange rates.

Key Model: `ExchangeRate`

Fields: - `source_currency`: 3-letter currency code (e.g., "USD", "CAD") - `destination_currency`: 3-letter currency code (e.g., "NGN", "GHS") - `rate`: The exchange rate (decimal with 8 decimal places) - `source_amount`: Amount in source currency - `destination_amount`: Amount in destination currency - `last_updated`: Timestamp of last update (auto-updated) - `created_at`: Timestamp of creation

Database Features: - **Unique constraint**: One rate per currency pair (source + destination) - **Indexes**: Fast lookups on currency pairs and update times - **Table name**: `exchange_rates`

Why it matters: This is where all exchange rates are persistently stored. The database acts as a backup when Redis cache expires or is cleared.

`rates/views.py`

Purpose: REST API endpoints that handle HTTP requests for exchange rates.

Key Classes:

1. `RatesView` (GET `/api/rates/`) - **Purpose**: Returns a single exchange rate for a currency pair - **Query Parameters**: `source_currency`, `destination_currency` - **Response**: Flutterwave-compatible JSON format

How it works (Priority Order): 1. **Check Redis Cache**: Fastest - returns immediately if found 2. **Check Database**: If not in cache, check SQLite database 3. **Check if Stale**: If database rate is older than 10 minutes, fetch fresh from Flutterwave 4. **Fallback to Flutterwave**: If not in database, fetch directly from Flutterwave API 5. **Save & Cache**: Store new rates in both database and Redis

2. `AllRatesView` (GET `/api/rates/all/`) - **Purpose**: Returns all exchange rates for a base currency - **Query Parameters**: `base_currency` (optional, defaults to "USD") - **Response**: Dictionary of all currency pairs for the base currency

Why it matters: These endpoints are what the Flutter app calls to get exchange rates. The multi-layer caching ensures instant responses.

`rates/services.py`

Purpose: Business logic for interacting with Flutterwave API.

Key Functions:

1. `fetch_flutterwave_rate(source_currency, destination_currency)` - **Purpose**: Makes HTTP request to Flutterwave API - **Endpoint**: `https://api.flutterwave.com/v3/transfers/rates` - **Features**: - Retry logic (3 attempts with exponential backoff) - 30-second timeout - Handles rate limiting (429 errors) - Returns raw Flutterwave JSON response

2. `save_rate_to_db(source_currency, destination_currency, fw_response)` - **Purpose**: Saves Flutterwave response to database - **Uses**: `update_or_create()` to avoid duplicates - **Extracts**: Rate, source amount, destination amount from Flutterwave response

3. `to_backend_shape(resp)` - **Purpose**: Ensures response matches Flutterwave format - **Currently**: Pass-through (returns response as-is)

Why it matters: This is the only place in the codebase that directly talks to Flutterwave. All rate fetching goes through these functions, making it easy to modify API interaction logic.

`rates/cache.py`

Purpose: Redis caching utilities for fast rate lookups.

Key Functions:

1. `get_rate(source_currency, destination_currency)` - **Purpose**: Retrieves rate from Redis cache - **Returns**: Cached rate data or `None` if not found - **Cache Key Format**: `fxrate:USD:NGN`

2. `set_rate(source_currency, destination_currency, payload, ttl_seconds)` - **Purpose**: Stores rate in Redis cache - **TTL**: Time-to-live in seconds (default: 120 seconds = 2 minutes) - **Why TTL**: Ensures cache doesn't hold stale data indefinitely

Why it matters: Redis is in-memory, making it 100x faster than database queries. This is the first layer of caching that provides instant responses.

`rates/urls.py`

Purpose: URL routing for the rates app REST API.

URL Patterns: - `rates/` → `RatesView` (single rate) - `rates/all/` → `AllRatesView` (all rates)

Full URLs: - `http://localhost:8000/api/rates/?source_currency=USD&destination_currency=NGN` - `http://localhost:8000/api/rates/all/?base_currency=CAD`

WebSocket Architecture

`rates/routing.py`

Purpose: Defines WebSocket URL patterns (similar to `urls.py` for HTTP).

Key Responsibilities: - Maps WebSocket URL `/ws/rates/` to `RatesConsumer` - Uses Django's `re_path` for regex-based routing

WebSocket URL: `ws://localhost:8000/ws/rates/`

Why it matters: This is the routing layer for WebSocket connections. Without this, WebSocket clients can't connect.

`rates/consumers.py`

Purpose: Handles WebSocket connections and real-time rate updates.

Key Class: `RatesConsumer`

Methods:

1. `connect()` - **When**: Client connects to WebSocket - **Actions**: - Accepts the connection - Joins the `rates_updates` group (for broadcasting) - Sends all current rates immediately

2. `disconnect(close_code)` - **When**: Client disconnects - **Actions**: Removes client from `rates_updates` group

3. `receive(text_data)` - **When**: Client sends a message - **Handles**: - `get_all_rates`: Sends all rates - `get_rate`: Sends specific rate

4. `rate_update(event)` - **When**: Background poller updates a single rate - **Action**: Broadcasts the update to all connected clients

5. `all_rates_update(event)` - **When**: Background poller finishes updating all rates - **Action**: Notifies all clients that rates have been refreshed

6. `get_all_rates_from_db()` / `get_rate_from_db()` - **Purpose**: Async database queries to fetch rates - **Uses**: `@database_sync_to_async` decorator to run sync Django ORM queries in async context

How Broadcasting Works: 1. Background poller (`poll_rates.py`) fetches new rate from Flutterwave 2. Poller calls `channel_layer.group_send('rates_updates', {...})` 3. All consumers in the `rates_updates` group receive the message 4. Each consumer's `rate_update()` method is called 5. Consumer sends the update to its connected client via WebSocket

Why it matters: This enables real-time updates without the Flutter app needing to poll. When rates change, all connected clients are instantly notified.

Background Tasks

`rates/management/commands/poll_rates.py`

Purpose: Background command that periodically fetches all exchange rates from Flutterwave.

Key Responsibilities: - Fetches rates for all currency pairs every 10 minutes (configurable) - Saves rates to database - Caches rates in Redis - Broadcasts updates via WebSocket

How to Run:

```
python manage.py poll_rates                    # Runs continuously
python manage.py poll_rates --once             # Runs once and exits
python manage.py poll_rates --interval 300     # Runs every 5 minutes
```

Currency Pairs: - **Source**: USD, CAD, GBP, EUR (4 currencies) - **Destination**: XOF, XAF, EGP, ETB, GHS, KES, MAD, NGN, ZAR, UGX, ZMW (11 currencies) - **Total Pairs**: 4 × 11 = 44 pairs (excluding same-currency pairs)

Workflow: 1. Loop through all source currencies 2. For each source, loop through all destination currencies 3. Fetch rate from Flutterwave API 4. Save to database 5. Cache in Redis (TTL = 2 × interval) 6. Broadcast update via WebSocket 7. Wait 0.5 seconds between requests (to avoid rate limiting) 8. After all pairs, broadcast "all rates updated" message 9. Sleep for interval (default: 10 minutes) 10. Repeat

Why it matters: This keeps the database and cache fresh. Without this running, rates would become stale. The WebSocket broadcasting ensures clients get updates in real-time.

Data Flow

1. Initial Setup Flow

```
Background Poller (poll_rates.py)
    ↓
Fetches from Flutterwave API
    ↓
Saves to Database (SQLite)
    ↓
Caches in Redis
    ↓
Broadcasts via WebSocket
    ↓
Flutter App receives update
```

2. API Request Flow (GET /api/rates/)

```
Flutter App Request
    ↓
RatesView.get()
    ↓
Check Redis Cache → Found? Return immediately ✓
    ↓ (Not found)
Check Database → Found? Check if stale
    ↓ (Stale or not found)
Fetch from Flutterwave API
    ↓
Save to Database
    ↓
Cache in Redis
    ↓
Return to Flutter App
```

3. Real-Time Update Flow

```
Background Poller detects new rate
    ↓
Saves to Database
    ↓
Caches in Redis
    ↓
Broadcasts via Channel Layer (Redis)
    ↓
All WebSocket Consumers receive message
    ↓
Each Consumer sends to connected client
    ↓
Flutter App updates UI instantly
```

Key Technologies

Django Channels

- **Purpose**: Adds WebSocket support to Django
- **Why**: Enables real-time bidirectional communication
- **Components**: Channel Layers, Consumers, Routing

Redis

- **Purpose**: Two uses:
  1. **Channel Layer**: Routes WebSocket messages between processes
  2. **Cache**: Stores exchange rates for fast access
- **Why**: In-memory storage is extremely fast (microseconds vs milliseconds)

Django REST Framework

- **Purpose**: Builds REST API endpoints
- **Why**: Provides standardized API responses and error handling

SQLite

- **Purpose**: Persistent storage for exchange rates
- **Why**: Simple, file-based database perfect for development and small deployments

Daphne

- **Purpose**: ASGI server that runs the Django application
- **Why**: Required to handle both HTTP and WebSocket connections
- **Alternative**: Could use uvicorn, but Daphne is Django's official ASGI server

---

Summary

This Django backend is architected for **speed and real-time updates**:

1. **Multi-layer caching**: Redis (fast) → Database (persistent) → Flutterwave (source of truth)
2. **Background polling**: Keeps data fresh without blocking user requests
3. **WebSocket broadcasting**: Pushes updates to clients instantly
4. **Stale rate detection**: Automatically refreshes rates older than 10 minutes
5. **Graceful fallbacks**: If cache fails, check database; if database fails, fetch from Flutterwave

The result: **Instant responses with no loading spinners**, exactly like TapTap's UX.

Quick Reference

Start the Server

```
cd server
source .venv/bin/activate
daphne -b 0.0.0.0 -p 8000 backend.asgi:application
```

Start the Poller

```
cd server
source .venv/bin/activate
python manage.py poll_rates
```

Test WebSocket

```
# Connect to: ws://localhost:8000/ws/rates/
# Send: {"type": "get_all_rates"}
```

Test REST API

```
curl "http://localhost:8000/api/rates/?source_currency=USD&destination_currency=NGN"
```

Document Version: 1.0
Last Updated: 2024
Author: Django Backend Documentation