Technical Design and Architecture Document (TDAD): Simple Currency Exchange Service (CES) - Django/Celery

1. System Architecture: Django MVT + Asynchronous Workers

The CES is structured as a **Django Project** utilizing the Model-View-Template (MVT) pattern, heavily integrated with the **Celery** distributed task queue for reliable background processing.

Component Breakdown (The "View" is separated into two concerns)

1. **Ingestion System (Asynchronous):**
    - **Celery Beat:** The cron-like scheduler that triggers the fetch_rates_task hourly.
    - **Celery Worker:** Executes the task. It calls the external client, performs data transformation, writes to PostgreSQL, and updates Redis.
2. **Conversion API (Synchronous):**
    - **Django Views/Serializers:** Receives the client request, validates input using Django Rest Framework (DRF) Serializers.
    - **Core Service/Manager:** The business logic layer that handles rate lookup (from cache), margin application, and delegates to the AuditManager before returning the response.

2. Data Modeling (PostgreSQL / Django ORM)

The core persistence strategy leverages the Django ORM and PostgreSQL's reliability.

2.1. ExchangeRate Model (Immutable Source of Truth)

This model is central to the audit trail and is only ever inserted, never updated.

Field Name	Type	Django Field Type	Constraints	Description
id	UUID	UUIDField	PK, Auto-Generated	Internal unique ID for the rate record.
base_currency	String (3)	CharField	Indexed	Currency code (e.g., 'USD').
target_currency	String (3)	CharField	Indexed	Currency code (e.g., 'NGN').
rate_value	Decimal	DecimalField(max_digits=15,	Required	The exchange rate (high precision).

		decimal_places=8)		
provider_name	String	CharField		Source of the rate (e.g., 'Fixer').
fetched_at	DateTime	DateTimeField	Auto-Add-Now	Timestamp when the rate was successfully fetched.

2.2. ConversionAudit Model (Immutable Audit Log)

Links every conversion back to the immutable ExchangeRate record via a Foreign Key.

Field Name	**Type**	**Django Field Type**	**Constraints**	**Description**
id	UUID	UUIDField	PK, Auto-Generated	Unique audit log ID.
rate_used	FK	ForeignKey(ExchangeRate)	Required	**Crucial link** to the exact immutable rate record used.
input_amount	Decimal	DecimalField	Required	Amount provided by the calling client.
output_amount	Decimal	DecimalField	Required	Calculated amount after conversion and margin.
converted_at	DateTime	DateTimeField	Auto-Now-Add	Timestamp of the conversion request.

3. Key Technical Implementations

3.1 Rate Ingestion with Celery Beat (Reliability)

The task fetch_rates_task will be configured in Celery Beat's settings (CELERY_BEAT_SCHEDULE).

- **Task Execution:** The task uses the requests library to call the external provider.
- **Immutability:** On success, it creates a **new** ExchangeRate record.
- **Cache Update:** Immediately after saving to the DB, the task uses Django's cache API to update the Redis backend with the new rate value.
- **Retry Mechanism:** If the external API call fails, the Celery task will be configured with an automatic retry policy (e.g., autoretry_for=(RequestException,), retry_backoff=True) before failing completely, ensuring high ingestion resilience.

3.2 Caching Strategy (Redis Backend)

We will utilize **Django's native Caching Framework** configured to use Redis.

- **Cache Key:** A predictable key based on the currency pair: f"{base_currency}:{target_currency}".
- **Cache Logic:** The RateService will attempt to cache.get(key). If a miss occurs, it will perform a fallback query to the PostgreSQL database for the *most recent* rate, serve that, and log a warning (Cache Miss) before returning.

3.3 Transaction Management and Margin

- **Conversion Logic:** The Conversion View calls a dedicated Django Manager/Service class that performs the conversion logic (amount * rate * (1 - margin)).
- **Atomic Audit Log:** The creation of the ConversionAudit entry is executed using the Django ORM within a standard atomic block to ensure the audit log is consistent with the response.

4. System Design Diagram (Architectural Flow)

The system is visualized below, showing the separation of synchronous (API) and asynchronous (Celery) processes.