NEARSHORE Documentation

1. Challenge Requirements

The challenge required building a Django-based web platform called MyCurrency that allows users to calculate currency exchange rates with these key features:

1. Store currencies and daily exchange rates in a database
2. Implement the Adapter Design Pattern for multiple currency data providers
3. Create a REST API for currency conversion and historical rates
4. Build an admin interface for currency conversion
5. Implement asynchronous processing for historical data loading
6. Make the system resilient with provider fallback mechanisms
7. Allow dynamic provider prioritization and activation/deactivation
8. Create a system that works with EUR, CHF, USD, and GBP currencies

2. What Was Achieved

We successfully implemented:

1. A Django project with models for currencies and exchange rates
2. The Adapter Design Pattern with two providers (CurrencyBeacon and Mock)
3. A RESTful API with endpoints for currency conversion and rate history
4. Custom admin pages for currency conversion and historical data loading
5. Asynchronous tasks using Celery for loading historical exchange rates
6. A resilient system with provider fallback and prioritization
7. A database that caches exchange rates to improve performance

3. Tech Stack Used

- **Backend**: Python 3.11, Django 5.0.2

- **API**: Django REST Framework 3.14.0
- **Database**: PostgreSQL (with the option to use SQLite for development)
- **Asynchronous Tasks**: Celery 5.2.7
- **Message Broker**: Redis
- **HTTP Client**: Requests
- **Environment Management**: python-dotenv

4. Project Functionality

1. **Currency Management**:
   - Store and retrieve currency information (code, name, symbol)
   - Support for EUR, USD, GBP, and CHF currencies
2. **Exchange Rate Providers**:
   - Adapter pattern implementation for different providers
   - CurrencyBeacon provider integration with real API calls
   - Mock provider for testing with random but realistic rates
   - Provider prioritization and fallback mechanisms
   - Option to activate/deactivate providers at runtime
3. **Exchange Rate Storage**:
   - Database storage of currency exchange rates
   - Caching mechanism to avoid repeated API calls
   - Historical exchange rate data storage and retrieval
4. **API Endpoints**:
   - Currency CRUD operations
   - Currency conversion with specified amount
   - Historical exchange rates for specified time periods
   - Provider selection capability

5. **Admin Interface**:
    - Custom currency converter tool
    - Historical data loading interface
    - Management of currencies and exchange rates
6. **Asynchronous Processing**:
    - Background tasks for loading historical exchange rate data
    - Celery worker implementation for handling heavy processing

5. Using the Project

API Endpoints

1. **Get all currencies**:

```
curl -X GET http://localhost:8000/api/currencies/
```

2. **Convert currency**:

```
curl -X POST http://localhost:8000/api/rates/convert/ \
  -H "Content-Type: application/json" \
  -d '{"source_currency": "USD", "amount": 100, "exchanged_currency": "EUR"}'
```

3. **Get historical rates**:

```
curl -X POST http://localhost:8000/api/rates/rates_list/ \
  -H "Content-Type: application/json" \
  -d '{"source_currency": "USD", "date_from": "2023-03-01", "date_to": "2023-03-10"}'
```

UI Access

1. **Admin Interface**:

- Access at http://localhost:8000/admin/
- Login with the superuser credentials
- Navigate to Currencies and Exchange Rates sections

2. **Currency Converter Tool**:
   - In the admin interface, go to http://localhost:8000/admin/currency-converter/
   - Select source currency, enter amount, select target currencies
   - Click "Convert" to see results
3. **Historical Data Loading**:
   - In the admin interface, go to http://localhost:8000/admin/load-historical-data/
   - Select days to go back, source currencies, and target currencies
   - Click "Load Data" to start the background task

6. Technical Documentation

Project Structure

```
mycurrency/
├── api/                # API app
│   ├── serializers.py    # DRF serializers
│   ├── urls.py          # API endpoints
│   └── views.py         # API view logic
├── core/               # Core app
│   ├── admin.py         # Admin customization
│   ├── models.py        # Database models
│   ├── services.py      # Business logic
│   ├── tasks.py         # Celery tasks
│   └── templates/       # Admin templates
├── providers/          # Providers app
│   ├── adapters/        # Provider adapters
```

```
│   │   ├── base.py       # Base adapter interface
│   │   ├── currency_beacon.py  # CurrencyBeacon implementation
│   │   └── mock.py       # Mock implementation
│   └── factory.py        # Provider factory
└── mycurrency/           # Project settings
    ├── celery.py         # Celery configuration
    ├── settings.py       # Django settings
    └── urls.py           # Main URL routing
```

Key Components

1. **Adapter Pattern Implementation**:
   - Base adapter interface (`ProviderAdapter`) with `get_exchange_rate` method
   - Concrete implementations for each provider
   - Provider factory to create provider instances and manage priorities
2. **Currency Service**:
   - Central exchange rate service with provider fallback
   - Database caching of exchange rates
   - Currency conversion calculations
3. **Asynchronous Tasks**:
   - Celery task implementation for historical data loading
   - Configurable parameters for source/target currencies and time period
4. **API**:
   - REST endpoints using Django REST Framework
   - Serializers for data validation and transformation
   - Viewsets and actions for different operations
5. **Admin Interface**:
   - Custom admin site implementation
   - Admin views for currency conversion and data loading

- Custom forms and templates

Database Schema

1. **Currency Model**:
   - code: 3-character currency code (e.g., "USD")
   - name: Currency name (e.g., "US Dollar")
   - symbol: Currency symbol (e.g., "$")
2. **CurrencyExchangeRate Model**:
   - source_currency: Foreign key to Currency model
   - exchanged_currency: Foreign key to Currency model
   - valuation_date: Date of the exchange rate
   - rate_value: Exchange rate value with 6 decimal places
   - provider: Provider name that provided the rate
   - created_at: Timestamp of when the rate was added

7. Important Notes

1. **API Keys**:
   - Keep API keys secure in environment variables
   - CurrencyBeacon has a free tier with limited requests
2. **Provider Limitations**:
   - CurrencyBeacon provides historical data
   - Some providers may only support specific base currencies
3. **Performance Considerations**:
   - Use database caching to minimize API calls
   - Implement rate limiting to avoid provider API limits
   - Use asynchronous tasks for bulk operations
4. **Fault Tolerance**:

- The system falls back to alternative providers if the primary fails
- The Mock provider always works as a last resort

5. **Future Improvements**:
    - Add more currency providers
    - Implement more sophisticated caching strategies
    - Add user authentication for API access
    - Create a frontend interface for end users
    - Add comprehensive test suite
    - Implement monitoring for API calls and errors
6. **Maintenance**:
    - Regularly check provider API changes
    - Monitor for exchange rate anomalies
    - Update currency symbols if they change
7. **Deployment**:
    - Ensure Redis is properly configured for production
    - Set up proper database connections with connection pooling
    - Configure proper logging for production
    - Set up SSL for secure API access

MyCurrency Project - Simple Explanation

HTML UI Links

1. **Main Admin Interface**: http://localhost:8000/admin/
2. **Currency Converter Tool**: http://localhost:8000/admin/currency-converter/
3. **Historical Data Loading**: http://localhost:8000/admin/load-historical-data/

4. **Currencies List**: http://localhost:8000/api/currencies/
5. **Exchange Rates List**: http://localhost:8000/api/rates/

Purpose of Each File We Changed

1. `core/models.py`

- **What we did**: Created database models for currencies and exchange rates
- **Purpose**: Store information about currencies (EUR, USD, etc.) and their exchange rates

2. `providers/adapters/base.py`

- **What we did**: Created an abstract base class for currency providers
- **Purpose**: Define a common interface that all currency providers must implement

3. `providers/adapters/currency_beacon.py`

- **What we did**: Implemented an adapter for the CurrencyBeacon API
- **Purpose**: Get real exchange rates from the CurrencyBeacon service

4. `providers/adapters/mock.py`

- **What we did**: Created a mock provider that generates random exchange rates
- **Purpose**: Have a fallback that always works even if external APIs are down

5. `providers/factory.py`

- **What we did**: Built a factory to create providers and manage their priorities
- **Purpose**: Allow the system to try different providers in order of priority

6. `core/services.py`

- **What we did**: Created a service to handle exchange rate logic
- **Purpose**: Core business logic for getting rates and converting currencies

7. `core/tasks.py`

- **What we did**: Implemented background tasks for loading historical data
- **Purpose**: Load exchange rates for many dates without blocking the user interface

8. `api/serializers.py`

- **What we did**: Created serializers to validate API input/output
- **Purpose**: Ensure data sent to/from the API is in the correct format

9. `api/views.py`

- **What we did**: Implemented API endpoints for currencies and exchange rates
- **Purpose**: Allow external applications to get exchange rates and convert currencies

10. `api/urls.py`

- **What we did**: Defined URL patterns for API endpoints
- **Purpose**: Map web URLs to our API views

11. `core/admin.py`

- **What we did**: Created custom admin views and site
- **Purpose**: Make it easy to use the currency converter and load historical data

12. `core/templates/admin/*.html`

- **What we did**: Created HTML templates for the admin interface
- **Purpose**: User-friendly forms for currency conversion and data loading

13. `mycurrency/settings.py`

- **What we did**: Configured Django settings and provider settings
- **Purpose**: Set up database connection, API keys, provider priorities, etc.

14. `mycurrency/celery.py`

- **What we did**: Set up Celery for background tasks
- **Purpose**: Handle long-running tasks without making the user wait

How Close Are We to Meeting Expectations?

We've successfully implemented all the core requirements:

1. ✅ **Currency Models**: Created database models as specified
2. ✅ **Adapter Pattern**: Implemented for two providers (CurrencyBeacon and Mock)
3. ✅ **REST API**: Created endpoints for conversion and historical rates
4. ✅ **Admin Interface**: Built custom admin pages for converter and historical data
5. ✅ **Async Tasks**: Set up Celery for loading historical data
6. ✅ **Provider Resilience**: Implemented fallback when providers fail
7. ✅ **Provider Priority**: Made providers configurable with priority settings

The implementation follows good practices:

1. ✅ **Code Structure**: Organized into logical components
2. ✅ **Resilience**: System continues working if external APIs fail
3. ✅ **Performance**: Caches exchange rates in the database
4. ✅ **Flexibility**: Easy to add new currency providers

Simple Steps to Use the Application

1. **Convert Currency**:
   - Go to http://localhost:8000/admin/currency-converter/
   - Select source currency (e.g., USD)
   - Enter amount (e.g., 100)
   - Select target currencies (e.g., EUR, GBP)
   - Click "Convert"

2. **Load Historical Data**:
    - Go to http://localhost:8000/admin/load-historical-data/
    - Choose how many days of history to load
    - Select which currencies to include
    - Click "Load Data"
3. **View Currencies**:
    - Go to http://localhost:8000/admin/
    - Click on "Currencies"
4. **View Exchange Rates**:
    - Go to http://localhost:8000/admin/
    - Click on "Currency Exchange Rates"
5. **Use the API**:
    - Use tools like curl, Postman, or your own application to call the API endpoints

The application is production-ready with a few additions that would be needed:

- More comprehensive testing
- User authentication for API access
- More detailed error handling and logging
- Production deployment configuration