Real Estate Chatbot Dashboard - Project Explanation

Project Overview

This is a full-stack web application that implements a Real Estate Chatbot Dashboard. Users can input natural language queries about real estate data (like "Show trends for Wakad" or "Compare rent in Bandra vs Wakad"), and the system processes these queries to return summarized insights, interactive charts, and filtered data tables.

The application was built to demonstrate skills in full-stack development, data processing with Pandas, REST API design, and modern web development practices.

Tech Stack

Backend

- **Django 5.2.9**: Web framework for rapid development
- **Django REST Framework**: For building REST APIs
- **Pandas**: Data manipulation and analysis
- **OpenPyXL**: Excel file reading
- **Django CORS Headers**: Handling cross-origin requests

Frontend

- **React 18**: Component-based UI library
- **Bootstrap 5**: CSS framework for responsive design
- **Chart.js & React-Chartjs-2**: Data visualization
- **Axios**: HTTP client for API calls

Development Tools

- **Python 3.10**: Backend runtime
- **Node.js**: Frontend runtime
- **Virtual Environment**: Isolated Python dependencies
- **npm**: Package management

Project Structure

```
real-estate-chatbot/
├── backend/                    # Django backend
│   ├── backend/                # Django project settings
│   │   ├── settings.py         # Project configuration
│   │   ├── urls.py             # Main URL routing
│   │   ├── wsgi.py             # WSGI configuration
│   │   └── asgi.py             # ASGI configuration
│   └── chatbot/                # Django app
│       ├── views.py            # API views
```

```
│       ├── urls.py           # App URL routing
│       ├── models.py         # Data models (empty)
│       ├── apps.py           # App configuration
│       └── migrations/       # Database migrations
├── frontend/                  # React frontend
│   ├── public/               # Static assets
│   ├── src/
│   │   ├── App.js            # Main React component
│   │   ├── App.css           # Styles
│   │   ├── index.js          # React entry point
│   │   ├── services/
│   │   │   └── api.js        # API service functions
│   │   └── setupTests.js     # Test setup
│   └── package.json          # Dependencies
├── Sample_data.xlsx          # Real estate data
├── manage.py                 # Django management script
├── venv/                     # Python virtual environment
├── README.md                 # Project documentation
└── TODO.md                   # Development tasks
```

Backend Architecture

Django Project Setup

The backend is structured as a standard Django project with one main app called `chatbot`.

Key Configuration in `backend/settings.py`:

- Added `rest_framework`, `corsheaders`, and `chatbot` to `INSTALLED_APPS`
- Configured CORS middleware for frontend communication
- Set `CORS_ALLOWED_ORIGINS` to allow requests from `http://localhost:3000`

Data Processing Logic

The core functionality resides in `chatbot/views.py` with the `QueryView` class:

```python
class QueryView(APIView):
    def post(self, request):
        query = request.data.get('query', '')
        # Process query and return response
```

Query Processing Steps:

1. **Load Data**: Read Excel file using Pandas
2. **Parse Query**: Extract location, year, and query type using keyword matching
3. **Filter Data**: Apply filters based on parsed parameters
4. **Generate Insights**: Calculate summaries, trends, or comparisons
5. **Prepare Response**: Format data for charts and tables

Query Types Supported:

- **Trends**: "Show trends for Wakad" - Line chart of price over years
- **Comparisons**: "Compare rent in Bandra vs Wakad" - Bar chart comparing locations
- **Averages**: "Show average price for Andheri over last 3 years" - Summary statistics

API Endpoint

Endpoint: `POST /query/`
Request Body:

```
{
  "query": "Show trends for Wakad"
}
```

Response Format:

```
{
  "summary": "Prices in Wakad have increased by 20% over the last 3 years...",
  "chart_data": {
    "labels": ["2020", "2021", "2022", "2023"],
    "datasets": [{
      "label": "Price",
      "data": [50000, 55000, 60000, 65000]
    }]
  },
  "table_data": [
    {"Location": "Wakad", "Year": 2020, "Price": 50000, "Demand": 70, "Supply":
60},
    // ... more rows
  ]
}
```

Frontend Architecture

React Application Structure

The frontend is a single-page application built with React.

Main Components:

- **App.js**: Root component containing the entire dashboard
- **api.js**: Service layer for API communication

UI Components

Search Interface:

- Input field for natural language queries
- Submit button
- Bootstrap styling for responsive design

Results Display:

- **Summary Alert**: Bootstrap alert showing text summary
- **Chart Component**: Chart.js integration for line/bar charts
- **Data Table**: Bootstrap table displaying filtered data

State Management

The app uses React's `useState` hook to manage:

- `query`: Current search input
- `summary`: API response summary
- `chartData`: Chart configuration object
- `tableData`: Array of data rows
- `loading`: Loading state during API calls

API Integration

Axios Configuration in `services/api.js`:

```
const API_BASE_URL = 'http://localhost:8000';

export const queryData = async (query) => {
  const response = await axios.post(`${API_BASE_URL}/query/`, { query });
  return response.data;
};
```

How the Application Works

User Flow

1. **User Input**: User types a query like "Show trends for Wakad"
2. **API Request**: Frontend sends POST request to `/query/` endpoint
3. **Backend Processing**:
    - Load Excel data into Pandas DataFrame
    - Parse query to extract keywords (location: "Wakad", type: "trends")
    - Filter data for Wakad location
    - Calculate trend analysis (price changes over years)
    - Generate chart data structure
    - Return filtered table data
4. **Frontend Rendering**:
    - Display summary text
    - Render Chart.js chart with trend data

- Show data table with relevant rows

Data Processing Example

For query "Show trends for Wakad":

Data Filtering:

```
# Filter for Wakad
wakad_data = df[df['Location'].str.lower() == 'wakad']

# Group by year and calculate averages
yearly_data = wakad_data.groupby('Year').agg({
    'Price': 'mean',
    'Demand': 'mean',
    'Supply': 'mean'
}).reset_index()
```

Chart Data Generation:

```
chart_data = {
    'labels': yearly_data['Year'].tolist(),
    'datasets': [{
        'label': 'Average Price',
        'data': yearly_data['Price'].tolist(),
        'borderColor': 'rgb(75, 192, 192)',
        'tension': 0.1
    }]
}
```

Key Features Implemented

1. Natural Language Processing

- Simple keyword-based parsing
- Support for location names, years, and query types
- Flexible query handling

2. Data Visualization

- Line charts for trends
- Bar charts for comparisons
- Responsive Chart.js integration

3. Data Tables

- Bootstrap-styled tables

- Dynamic column rendering
- Filtered data display

4. Responsive Design

- Bootstrap grid system
- Mobile-friendly interface
- Clean, professional UI

5. Error Handling

- API error catching
- User-friendly error messages
- Loading states

Installation and Setup

Backend Setup

1. **Create Virtual Environment:**

```
python -m venv venv
venv\Scripts\activate  # Windows
```

2. **Install Dependencies:**

```
pip install django djangorestframework pandas openpyxl django-cors-headers
```

3. **Run Migrations:**

```
python manage.py migrate
```

4. **Start Backend Server:**

```
python manage.py runserver
```

Frontend Setup

1. **Navigate to Frontend Directory:**

```
cd frontend
```

2. **Install Dependencies:**

```
npm install
```

3. **Start Development Server:**

```
npm start
```

Running the Application

- Backend: http://127.0.0.1:8000
- Frontend: http://localhost:3000

Code Walkthrough

Backend Views (`chatbot/views.py`)

The `QueryView` class handles the main logic:

```
class QueryView(APIView):
    def post(self, request):
        query = request.data.get('query', '').lower()

        # Load data
        df = pd.read_excel('Sample_data.xlsx')

        # Parse query
        locations = self.extract_locations(query)
        years = self.extract_years(query)
        query_type = self.determine_query_type(query)

        # Filter and process data
        filtered_data = self.filter_data(df, locations, years)
        summary = self.generate_summary(filtered_data, query_type)
        chart_data = self.generate_chart_data(filtered_data, query_type)
        table_data = filtered_data.to_dict('records')

        return Response({
            'summary': summary,
            'chart_data': chart_data,
            'table_data': table_data
        })
```

Frontend Component (`src/App.js`)

The main React component:

```
function App() {
  const [query, setQuery] = useState('');
  const [summary, setSummary] = useState('');
  const [chartData, setChartData] = useState(null);
  const [tableData, setTableData] = useState([]);

  const handleSubmit = async (e) => {
    e.preventDefault();
    try {
      const response = await queryData(query);
      setSummary(response.summary);
      setChartData(response.chart_data);
      setTableData(response.table_data);
    } catch (error) {
      console.error('Error fetching data:', error);
    }
  };

  return (
    <div className="container mt-5">
      {/* Search form and results display */}
    </div>
  );
}
```

Challenges and Solutions

1. CORS Issues

Challenge: Frontend couldn't communicate with backend due to CORS policy.
Solution: Installed and configured `django-cors-headers` middleware.

2. Query Parsing

Challenge: Implementing natural language understanding for various query types.
Solution: Used simple keyword matching and regex patterns for location and year extraction.

3. Data Visualization

Challenge: Integrating charts with dynamic data.
Solution: Used Chart.js with React wrapper, structured data appropriately for chart consumption.

4. Excel Data Handling

Challenge: Reading and processing Excel files in Django.
Solution: Used Pandas for data manipulation and OpenPyXL for Excel reading.

Future Improvements

1. Advanced NLP

- Integrate NLP libraries like spaCy or NLTK for better query understanding
- Support more complex queries with multiple conditions

2. Database Integration

- Replace Excel with proper database (PostgreSQL/MySQL)
- Add data models for real estate properties

3. Enhanced Visualizations

- Add more chart types (pie charts, scatter plots)
- Interactive charts with drill-down capabilities

4. User Authentication

- Add user accounts and saved queries
- Personalized dashboards

5. Real-time Data

- WebSocket integration for live data updates
- API integrations with real estate data providers

6. Testing

- Unit tests for backend logic
- Integration tests for API endpoints
- Frontend component tests

Learning Outcomes

This project demonstrates:

- Full-stack development with Django and React
- Data processing and analysis with Pandas
- REST API design and implementation
- Modern frontend development practices
- Problem-solving and debugging skills
- Project structure and organization
- Documentation and code quality

The application successfully handles real estate data queries, providing valuable insights through text summaries, visualizations, and data tables, making it a comprehensive demonstration of full-stack capabilities.