InfoDock - Documentation

June 11, 2025

Contents

1. Project Overview

1.1. Project Summary

This application allows users to browse and visualize currency exchange rate data from the NBP API (National Bank of Poland). The application provides a welcoming and intuitive web interface. Users can fetch exchange rate and gold price data, view it as interactive charts, and store fetched data in a MongoDB database for future use.

1.2. Project Components

- **Web-Client (React + Vite)** – UI layer with data fetching and visualization.
- **Server (Python, Flask)** – REST API for saving and retrieving exchange rates.
- **Database (MongoDB)** – Stores saved currency rate records.
- **NBP API** – External data provider.

1.3. Environment

The application is designed to run in Docker containers, providing an easy setup and deployment process. It is fully functional in both local and cloud environments. To run the application, users need to have Docker and Docker Compose installed.

2. Docker Configuration

Docker is used to containerize the entire application, including the backend, frontend, and database services. This allows for a consistent development environment, easy deployment, and scalability. The configuration includes defining services, ports, and volumes for each component.

2.1. Backend Service (Flask Server)

The backend service is containerized using Docker and runs the Flask application. The container listens on port 5000 and depends on the MongoDB container for data storage. It exposes the necessary API endpoints for interacting with the tasks and user data.

2.2. Frontend Service (React Application)

The frontend is built using React and is also containerized using Docker. The React app is served on port 7666 by default. The frontend interacts with the backend API to display and manage tasks.

2.3. Database Service (MongoDB)

MongoDB (Flask version) is used as the database to store currency data. It is containerized as a separate service and is configured to persist data using Docker volumes. The container runs and listens on port 27017.

2.4. docker-compose.yml

To simplify the management and orchestration of the services, Docker Compose is used to define and run multi-container applications. The `docker-compose.yml` file specifies the configuration for each container and their relationships.

- **Services:** The setup includes three main services:
  - **mongodb** – A MongoDB container used as the primary database. Initialized with default admin credentials and a specific database (`infodock`).
  - **server** – The backend API service, built from the `./Server` directory. It depends on MongoDB and uses environment variables from a `.env` file.
  - **client** – The frontend React application, built from the `./Web-Client` directory. It depends on the backend server.
- **Networks:** A custom Docker network named `backendnet` is used to enable isolated and secure communication between the services.
- **Ports:**
  - `mongodb` exposes port `27017` for database access.
  - `server` is exposed on port `5000` for backend API access.
  - `client` is exposed on port `7666`, mapped from port `80` inside the container, making the frontend accessible via browser.

- **Volumes:** The backend server is mounted as a volume to enable real-time development without rebuilding the image on every change. MongoDB does not yet have a named volume configured for data persistence (this could be added in the future).
- **Environment:** MongoDB is configured with an initial root username, password, and default database using environment variables.

3. Architecture Overview

3.1. Data Flow

1. Web-Client sends request and receive data from the Server.
2. Server sends valid requests to the NBP API to fetch data.
3. NBP API sends responses to the Server.
4. Server sends and receives data from Mongo Database.
5. Server forwards data send by Mongo Database to Web-Client.

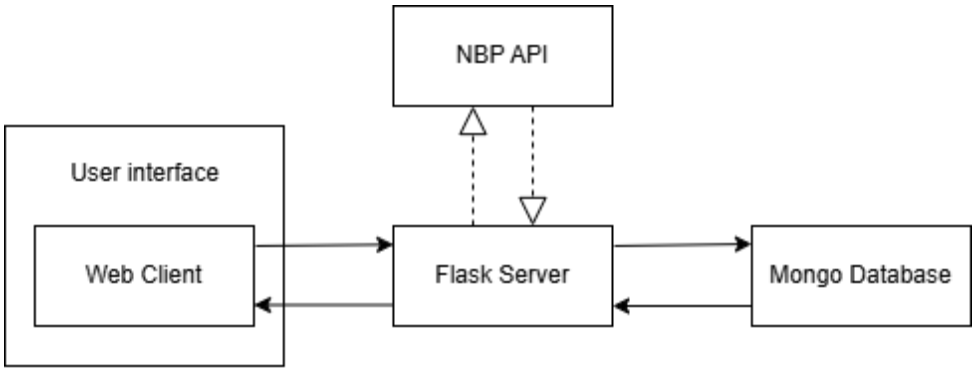

Figure 1: App Data Flow Diagram

3.2. Data Flow in Use Cases

3.2.1 Getting Currency Data from NBP API

1. User interacts with UI to select data he needs.
2. Web-Client sends request to the Server to fetch data.
3. Server validates user data and sends request to the NBP API to fetch data.
4. Server sends response to Frontend Web-Client.
5. Data is visualized on chart and React components.

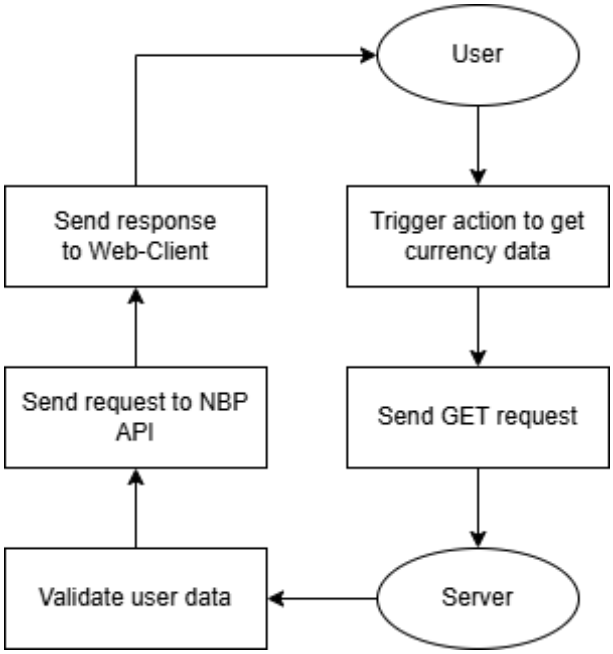

Figure 2: Getting Currency Data from NBP API diagram

3.2.2 Saving Currency Data to Database

1. User triggers an action to save fetched currency data (via an UI button).
2. Web-Client sends a `POST` request with JSON payload to the Server.
3. Server parses and validates the payload (e.g., currency code and rates).
4. Valid data is stored in the `currency_history` collection in MongoDB.
5. Server responds with a confirmation message to the Web-Client.

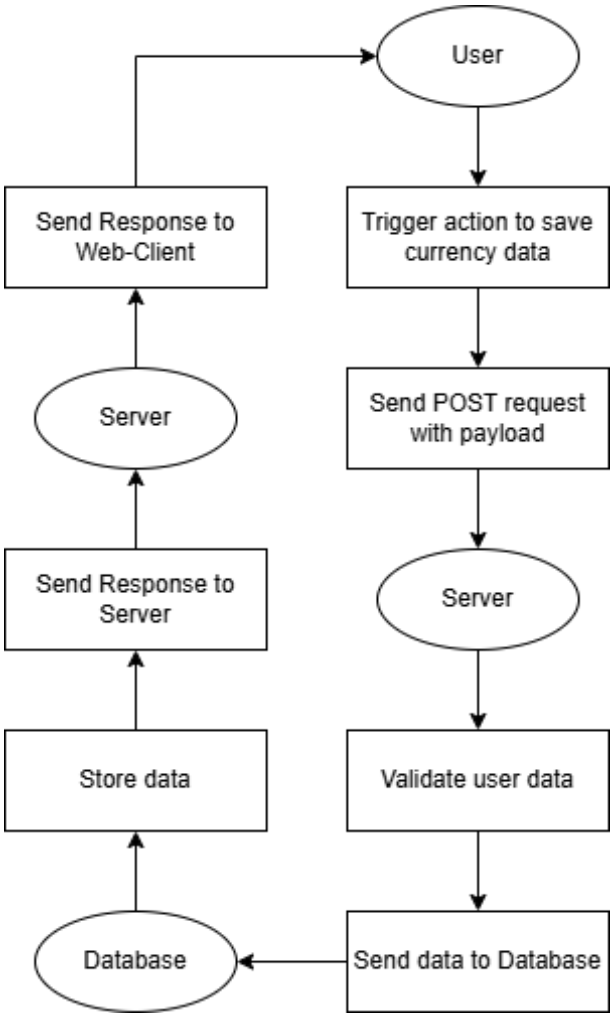

Figure 3: Saving Currency Data to Database Diagram

3.2.3 Retrieving All Saved Currency History

1. User requests to view previously saved currency history.
2. Web-Client sends a `GET` request to the Server.
3. Server queries the `currency_history` collection in MongoDB.
4. Retrieved entries are formatted and sent back in the response.
5. Web-Client displays the historical data using charts or lists.

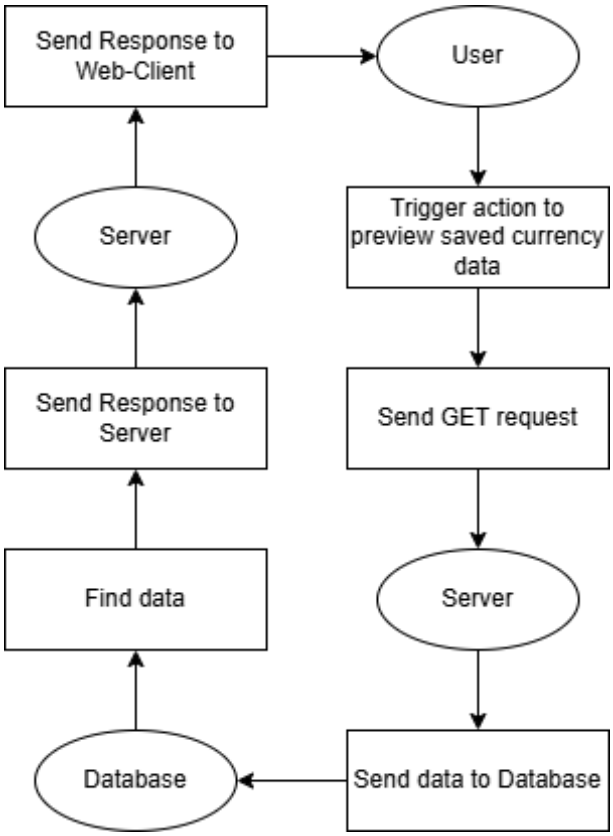

Figure 4: Retrieving All Saved Currency History Diagram

3.2.4 Deleting All Currency History from Database

1. User triggers an action to clear saved currency history (via an UI button).
2. Web-Client sends a `DELETE` request to the Server.
3. Server performs a bulk delete operation on the `currency_history` collection in MongoDB.
4. Server responds with the number of deleted entries and a success message.
5. UI displays confirmation of successful deletion to the user.

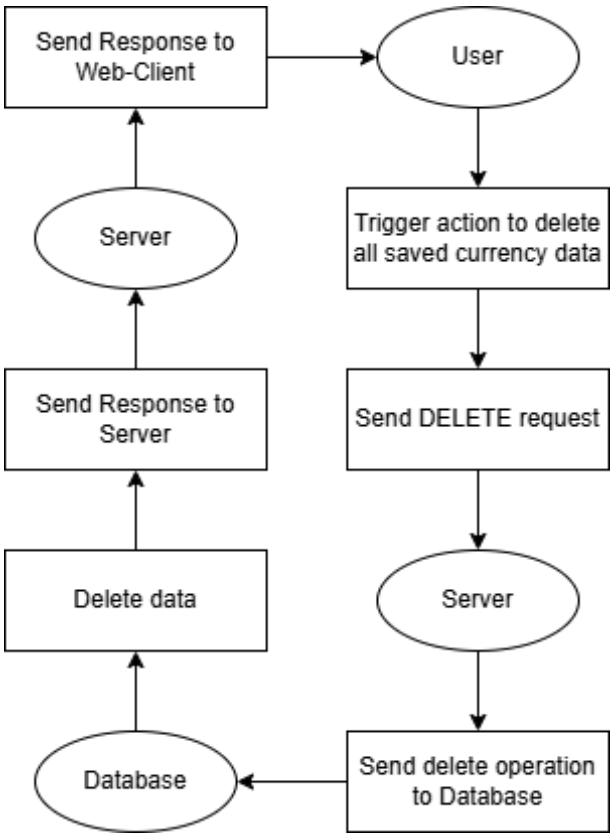

Figure 5: Deleting All Currency History Diagram

4. Server

4.1. Introduction

This backend application provides a RESTful API that interacts with the Polish National Bank (NBP) to fetch exchange rates and gold prices. It also allows the saving and retrieval of currency rate history using MongoDB. The system serves as a core data provider for the frontend interface that displays financial data to users.

4.2. Technology Stack

Technology	**Purpose**
Flask	Backend API framework
Flask-PyMongo	MongoDB integration with Flask
MongoDB	NoSQL database for storing historical currency data
python-dotenv	Environment variable management
Postman	Manual testing of API endpoints

4.3. Folder Structure

```
Server/
 app/
    __init__.py      # Backend files initializer
    config.py        # Backend config file
    main.py
    db/
       mongo.py     # MongoDB setup file
    routes/
       currency.py  # All endpoints Blueprint file
    services/
        currency_service.py
 Dockerfile
 requirements.txt
 run.py               # App starting file
```

4.4. MongoDB

MongoDB is configured using the connection string:

`mongodb://admin:admin123@mongodb:27017/infodock?authSource=admin`

The name of the database is `infodock`.

4.4.1 Database Schema

Figure 6: Database Diagram

4.4.2 Database Collections

The database uses a collection named:

- **currency_history** - Stores currency exchange rates historically in the format:

```
{
  "code": "USD",
  "rates": [
    {"date": "2025-06-01", "rate": 4.0},
    ...
  ]
}
```

4.5. Endpoints

- **GET /api/currency/**
  Fetch current exchange rate. Optional query param: `code` (default: USD)
- **GET /api/currency/range/**
  Fetch exchange rates over a date range. Query params: `code`, `startDate`, `endDate`
- **GET /api/currency/codes/**
  Retrieve list of supported currency codes.
- **GET /api/currency/gold/**
  Get current gold price.
- **GET /api/currency/gold/range/**
  Get gold prices over a date range. Query params: `startDate`, `endDate`

- **POST /api/currency/currency-history/save/**
  Save historical exchange rates to MongoDB. Expects JSON with:

```
{
  "code": "USD",
  "rates": [
    {"date": "2025-06-01", "rate": 4.0},
    ...
  ]
}
```

- **GET /api/currency/currency-history/**
  Retrieve all saved exchange rate entries from MongoDB.
- **DELETE /api/currency/currency-history/delete-all/**
  Remove all stored exchange rate data from MongoDB.

4.6. Testing

API endpoints were manually tested using Postman. All routes were verified for correctness, response formatting, and proper error handling for missing or malformed inputs.

5. Web-Client

5.1. Introduction

This React-based UI is designed to display exchange rate and gold price data from the NBP API (National Bank of Poland).

5.2. User Features

Users can:

- Go to Market Dashboard or Market History Dashboard,
- Search currency rates or gold prices,
- Search various currencies (offered by NBP API),
- Select various date ranges,
- Visualize data on a responsive line chart using `Recharts`,
- Save fetched data to a backend database,
- Browse and delete previously saved records.

5.3. Technology Stack

Technology	Purpose
React (with Vite)	UI and page management
JavaScript (ES6+)	Application logic
SASS (SCSS)	Styling with custom theming and variables
Recharts	Rendering charts for data visualization
Axios	HTTP requests
React Router	creating Webpage as SPA

5.4. Folder Structure

```
src/
 common/                # Common project files
    variables.scss
 components/            # Reusable UI components
    CustomCheckbox/
       CustomCheckbox.jsx
       CustomCheckbox.module.scss
    LoadingIndicator/
       LoadingIndicator.jsx
       LoadingIndicator.module.scss
    MarketCurrencyChart/
       MarketCurrencyChart.jsx
       MarketCurrencyChart.module.scss
    MarketForm/
```

```
      MarketForm.jsx
      MarketForm.module.scss
   NavBar/
      NavBar.jsx
      NavBar.module.scss
   SaveCurrencyDataButton/
       SaveCurrencyDataButton.jsx
       SaveCurrencyDataButton.module.scss
pages/                  # Route-based views
   HomePage/
      HomePage.jsx
      HomePage.module.scss
   MarketDashboard/
      MarketDashboard.jsx
      MarketDashboard.module.scss
   MarketHistoryDashboard/
      MarketHistoryDashboard.jsx
      MarketHistoryDashboard.module.scss
   NotFoundPage/
       NotFoundPage.jsx
       NotFoundPage.module.scss
services/               # API abstraction
   axiosInstance.js
App.jsx                 # Routing configuration
App.scss
index.scss              # Main app styles
main.jsx                # Entry point
```

5.5. Routing Setup

Routing is created by using React Router (7.6.2). All URLs are public, no secure routes used (no user authentication needed). Used routes URLs:

- '/' - HomePage
- '/marketDashboard' - MarketDashboard
- '/marketHistoryDashboard' - MarketHistoryDashboard
- '*' - NotFoundPage (all other URLs)

5.6. SCSS Styling

Styling is managed via SCSS using global variables and ".module" feature.

Fonts

```
$font-headings: (
  font-family: ('IBM Plex Mono', monospace),
  font-weight: 400
);

$font-general: (
  font-family: ('Inter', sans-serif),
  font-weight: 300
);
```

Color Palette

```
$dark-blue: #0D1B2A;
$dark-blue-hover: color.adjust($dark-blue, $lightness: 15%);
$gray: #666666;
$light-gray: #CFCFCF;
$black: #000000;
$light-black: #2f2f2f;
$white: #ffffff;
$dark-white: #f2f2f2;

$green: #4CAF50;
$red: #C81E18;
$red-hover: color.adjust($red, $lightness: 10%);

$market-background-color: #CFCFCF;
$market-currency-color: #00B4D8;
$market-currency-gold-color: #FFD700;
$market-currency-dark-gold-color: #cbb11f;

$Text-Black-Headers: #1a1a1a;
$Text-Black-Texts: #333333;
$Text-Black-SubTexts: #666666;
$Text-White-Headers: #ffffff;
$Text-White-Texts: #f5f5f5;
$Text-White-SubTexts: #cccccc;
```

5.7. Axios Instance

Application uses axios library and creates "axiosInstance" with baseURL set for default Server address.

5.8. State Management

- No Redux, Zustand, or React Query
- State managed locally using `useState` and `useEffect`

```
const [currencyData, setCurrencyData] = useState({});

useEffect(() => {
    getCurrencyCodes();
}, []);
```

5.9. Error handling

User is informed about existing error. All errors are showed as plain, red text below the element they apply to. Errors are set and cleared using useState hook e.g.:

```
const [loadingErrors, setLoadingErrors] = useState({
    loadingCurrencyCodesError: null,
    loadingCurrencyDataError: null,
});
```

5.10. Testing

Application was tested only manually.