Portfolio Management Recommendation System

Developer Manual

Version 1.0

Table of Contents

---

1. System Overview

The Portfolio Management Recommendation System is a comprehensive financial application designed to help users create optimized investment portfolios based on their risk tolerance, investment horizon, and financial goals. The system combines modern portfolio theory with data analytics and machine learning techniques to recommend asset allocations that maximize returns for a given level of risk.

Key Features:

- **Personalized Portfolio Recommendations**: Tailors investment portfolios to individual risk tolerances and preferences.
- **Portfolio Analysis & Tracking**: Provides real-time monitoring and performance metrics for user portfolios.
- **Risk Management Tools**: Incorporates sophisticated risk modeling to help users understand potential downside risks.
- **Data-Driven Insights**: Leverages historical market data and analytics to support recommendations.
- **Interactive Tools**: Offers stock comparisons, calculators, and investment simulators.
- **User Management**: Includes comprehensive user account handling and preference management.
- **Visualization**: Presents data through intuitive charts and tables for better understanding.

Target Audience:

The system serves both novice and experienced investors, providing simplified insights for beginners while offering detailed analytics for professionals. It adapts its recommendations and level of detail based on the user's investment familiarity.

The system is built using a modern tech stack with React for the front-end, Node.js (Express) for the main server, Python for advanced algorithms, and MongoDB for data persistence. This architecture balances user experience, computational efficiency, and data management capabilities.

---

2. Architecture

2.1 High-Level Architecture

The Portfolio Management Recommendation System employs a two-tier client-server architecture consisting of:

1. **Front-End Application**: Single-page application (SPA) built with React.js, responsible for user interface and client-side logic.
2. **Back-End Services**:
    - **Node.js/Express API Server**: Handles HTTP requests, manages user authentication, and orchestrates data flow.
    - **Python Computation Services**: Handles computationally intensive tasks like portfolio optimization and data processing.
    - **MongoDB Database**: Stores user profiles, preferences, and generated portfolios.

This decoupled design allows independent development, testing, and deployment of the UI and server components, with communication handled via RESTful API calls.

2.2 Front-End Architecture

The front-end follows a component-based architecture with React:

- **Routing**: React Router handles navigation between views without full page reloads
- **State Management**: Uses React's native state and context APIs for state management
- **API Integration**: Axios for HTTP requests to the back-end
- **Visualization**: Recharts and Chart.js libraries for data visualization
- **UI Components**: Combination of custom components and third-party libraries

The front-end is organized into several key views:

- Home/Login
- User Profiles Dashboard
- Stock Board (Portfolio Display)
- User Preferences
- Stock Details
- Interactive Tools
- News and Insights
- Contact

2.3 Back-End Architecture

The back-end consists of several interconnected modules:

1. **Express Server (`server.js`)**:
    - Provides REST API endpoints
    - Handles user authentication with JWT
    - Processes requests and returns responses
    - Communicates with the database
    - Executes Python scripts for optimization
2. **Python Modules**:
    - **Portfolio Construction**: Implements optimization algorithms
    - **Data Processing**: Handles preprocessing of stock and fundamental data
    - **Machine Learning**: TFT model for predictions
3. **Data Storage**:
    - MongoDB for structured data
    - CSV files for cached data and configuration
    - File system for temporary data storage

2.4 Data Flow

1. **User Registration/Login**:
    - User submits credentials
    - Server authenticates and returns JWT token
    - Front-end stores token in localStorage
2. **Portfolio Generation**:
    - User submits risk preferences form
    - Server processes preferences
    - Python optimization algorithm generates portfolio
    - Result is stored in database and returned to front-end
3. **Portfolio Display and Monitoring**:
    - Front-end requests portfolio data
    - Server retrieves stored portfolio
    - Front-end fetches current stock prices from Finnhub API
    - Data is visualized in tables and charts
4. **Historical Analysis**:
    - Back-end maintains cached historical data
    - Data is processed to compute portfolio performance
    - Results are displayed in charts on front-end

This architecture ensures separation of concerns while maintaining efficient data flow between components.

---

3. Front-End Components

3.1 Core Components

App.js

Located at the root of the component hierarchy, `App.js` defines the routing structure for the entire application using React Router. It maps URL paths to specific components and establishes the overall navigation flow.

Key functions:

- Sets up `<Router>` with defined `<Routes>`
- Maps paths to components (e.g., `/stockboard/:email` to `<StockBoard />`)
- Handles dynamic parameters in routes

Home.js

The landing page and authentication portal for the application. This component handles user login and signup processes.

Key functions:

- `handleLoginClick()`: Opens authentication modal
- `handleLogin()`: Processes login form submission and authentication
- `handleSignup()`: Processes signup form submission
- `isValidEmail()`: Validates email format using regex

UserProfiles.jsx

A dashboard component displaying all user profiles with drag-and-drop capabilities for reordering. It allows administrators to manage users and their portfolios.

Key functions:

- `handleDragEnd()`: Processes the end of drag operations for reordering
- `handleDeleteUser()`: Initiates user deletion process
- `confirmDelete()`: Executes user deletion after confirmation
- `handleCreateUserClick()`: Opens modal for new user creation
- `handleCreatePortfolioClick()`: Navigates to portfolio creation for a user

3.2 User Authentication

Authentication Flow

The authentication system uses JWT tokens stored in localStorage to maintain user sessions.

Key components and functions:

- `Home.js`: Contains login and signup forms
- `handleLogin()`: Submits credentials to backend, stores token on success

- `handleSignup()`: Creates new user accounts
- Authentication check in route components to prevent unauthorized access

3.3 Portfolio Management

UserPreferences.js

Collects user risk tolerance and investment preferences through a questionnaire.

Key functions:

- `handleSelect()`: Updates response state when an answer is selected
- `validateForm()`: Ensures all questions have been answered
- `handleFormSubmit()`: Processes form submission and sends to API

StockBoard.js

Displays the user's stock portfolio with real-time market data in both table and chart views.

Key functions:

- `fetchStockData()`: Retrieves current stock data from Finnhub API
- `handleSort()`: Manages table column sorting
- `handleViewToggle()`: Switches between table and chart views
- `fetchPortfolio()`: Retrieves user's portfolio from API
- `updateHistoricalData()`: Refreshes historical data for visualization

3.4 Analysis Tools

InteractiveTools.js

Container component for various financial tools.

Key function:

- `handleTimeframeChange()`: Updates selected time period for analysis

StockComparison.js

Allows users to compare multiple stocks visually on a single chart.

Key functions:

- `fetchStockData()`: Retrieves historical data for selected stocks
- `handleRemoveStock()`: Removes a stock from the comparison list

StockDetails.js

Displays detailed information about a specific stock with price charts.

Key functions:

- `fetchStockGraphData()`: Retrieves and processes historical data for charting
- `handleTimeframeChange()`: Updates chart timeframe

3.5 Visualization Components

PortfolioChart.js

Creates line chart visualizations of portfolio value over time.

Key functions:

- `fetchHistoricalData()`: Retrieves historical price data for portfolio stocks
- `getFilteredData()`: Filters chart data based on selected time period
- `intervalToDays()`: Converts time interval string to number of days

3.6 Navigation System

The application uses React Router for navigation between different views. Navigation is handled through:

1. **URL-based routing**: Defined in App.js
2. **Programmatic navigation**: Using the `useNavigate()` hook
3. **Link-based navigation**: Using `<Link>` or `<NavLink>` components
4. **Sidebar menu**: Provides consistent navigation across the application

Key navigation functions (present in multiple components):

- `navigate('/path')`: Redirects to specified route
- `handleStockBoardClick()`, `handleNewsAndInsightsClick()`, etc.: Navigation event handlers

---

4. Back-End Services

4.1 Server Structure

The server is implemented as a Node.js Express application (`server.js`) with the following structure:

- **Middleware Setup**: Configurations for body parsing, CORS, etc.
- **Authentication Middleware**: JWT verification for protected routes
- **Routes/Endpoints**: API definitions for different functionalities
- **Helper Functions**: Utilities for common tasks like CSV processing
- **External Process Execution**: Methods to call Python scripts for optimization

4.2 API Endpoints

User Authentication

- **POST /login**: Authenticates user credentials and issues JWT
- **POST /signup**: Creates new user accounts
- **GET /get-email**: Extracts email from JWT token

User Management

- **GET /all-users**: Retrieves all registered users
- **DELETE /delete-user/:email**: Removes a user and associated data

Preferences

- **POST /save-preferences**: Stores user investment preferences
- **GET /user-preferences**: Retrieves saved preferences

Portfolio Management

- **GET /get-portfolio**: Retrieves portfolio for a specific user

Data Retrieval

- **GET /get-historical-data**: Fetches historical stock data
- **GET /update-all-historical-data**: Refreshes historical data for all portfolio stocks

4.3 Authentication

The server implements JWT-based authentication:

1. **Token Generation**: On successful login/signup, a JWT token is created with user email
2. **Token Verification**: `authenticateToken` middleware checks token validity
3. **Route Protection**: Protected routes use the middleware to ensure authentication

Key functions:

- `authenticateToken()`: Middleware to verify JWT tokens
- JWT signing/verification with `jwt.sign()` and `jwt.verify()`

4.4 Data Processing

CSV Handling

The server includes utilities for reading and writing CSV files, which are used for storing user credentials and preferences.

Key functions:

- `readCSV()`: Parses CSV files into arrays
- `writeCSV()`: Writes arrays to CSV format with proper handling of special characters

Portfolio Generation

When user preferences are saved, the server:

1. Calls `generatePortfolio()` to calculate parameters
2. Executes Python optimization script via `getPortfolio()`
3. Fetches initial stock prices with `fetchInitialPrices()`

4. Saves portfolio data with `savePortfolio()`

4.5 External Data Integration

The server integrates with external financial APIs:

1. **Finnhub API**: For real-time stock quotes and company information
2. **Yahoo Finance**: For historical stock data retrieval

Key integration functions:

- `fetchInitialPrices()`: Gets current stock prices from Finnhub
- `fetchHistoricalData()`: Retrieves historical stock data with caching mechanism

---

5. Portfolio Optimization Algorithms

5.1 Modern Portfolio Theory Implementation

The portfolio optimization is implemented in `portfolio_construction.py` and follows Modern Portfolio Theory (MPT) principles. The algorithm seeks to maximize expected return for a given level of risk by finding optimal asset weights.

Core function: `optimize_portfolio_rolling_parallel()`

This function orchestrates the optimization process:

1. Takes parameters (portfolio_size, lambda_val, latest_predictions, cov_matrix, investment_horizon)
2. Uses a greedy approach to build optimal portfolios
3. Returns selected assets and their optimal weights

5.2 Asset Selection Algorithm

The system uses a greedy algorithm for asset selection, implemented in `optimize_portfolio_rolling_parallel()`:

1. Start with an empty portfolio
2. Evaluate adding each remaining asset in parallel
3. Add the asset that maximizes the objective function
4. Repeat until target portfolio size is reached
5. Perform final optimization with tighter allocation constraints

The asset evaluation is handled by `optimize_single_asset()`, which:

1. Creates a temporary portfolio including the new asset
2. Extracts expected returns for the portfolio
3. Extracts the relevant subset of the covariance matrix
4. Defines an objective function to maximize return minus risk penalty
5. Uses `scipy.optimize.minimize()` to solve the optimization problem

6. Returns the result with objective value and optimal weights

5.3 Risk Modeling

Risk is modeled using the covariance matrix of asset returns. The implementation includes:

1. **Covariance Matrix Computation**: Implemented in `get_price_data.py` with the `compute_cov()` function
2. **Risk Aversion Parameter**: Lambda value derived from user preferences
3. **Portfolio Variance Calculation**: Using matrix multiplication: `w^T * cov_matrix * w`
4. **Risk-Return Tradeoff**: Balancing expected return against variance in the objective function

5.4 Expected Return Calculation

Expected returns are calculated in `get_price_data.py` using both historical data and the CAPM model:

Key function: `compute_expected_returns_multiple_rates()`

This function:

1. Computes beta coefficients for different time intervals
2. Uses multiple treasury rates as risk-free rates
3. Applies CAPM formula: `E(R) = Rf + β * (Rm - Rf)`
4. Creates expected return values for various time horizons

5.5 Covariance Matrix Computation

The covariance matrix calculation is implemented in `get_price_data.py`:

Key function: `compute_cov(prices_df, interval)`

This function:

1. Resamples price data according to specified interval
2. Computes returns for the interval
3. Calculates the covariance matrix of returns
4. Handles missing values and ensures valid matrix properties

5.6 Weight Optimization

The final weight optimization is performed using quadratic programming through scipy:

Key steps in the optimization:

1. Define objective function: `-(μ * w - λ/2 * w^T * Σ * w)`
2. Set constraints: Sum of weights = 1, bounds on individual weights
3. Solve using SLSQP method in `scipy.optimize.minimize()`
4. Apply post-processing to ensure practical allocation (e.g., minimum 2%, maximum 15%)

---

6. Data Processing Pipeline

6.1 Stock Price Data Retrieval

Implemented in `get_price_data.py`, this module handles fetching and processing stock price data.

Key functions:

- `fetch_stock_data(ticker)`: Retrieves historical price data from Yahoo Finance
- `calculate_horizon_returns(df)`: Calculates forward-looking returns for multiple horizons
- `get_price_data(tickers, all_close_prices)`: Processes data for multiple tickers

The pipeline steps include:

1. Fetching raw data from Yahoo Finance
2. Calculating returns over various time horizons (2-24 months)
3. Computing beta coefficients against S&P 500
4. Calculating expected returns using CAPM
5. Saving processed data to CSV files

6.2 Fundamental Data Processing

The system processes fundamental financial data in several steps:

merge_fundamental_data.py

Consolidates fundamental data across multiple source folders:

1. Loads ticker mapping from configuration
2. Finds relevant files for each ticker
3. Merges data with date alignment
4. Filters to include only data from 2009 onward
5. Saves consolidated data to individual ticker files

6.3 Data Merging

merge_ticker_data.py

Combines price and fundamental data:

1. Loads price and fundamental data files
2. Creates a complete timeline with all dates from both sources
3. For each date, fills missing data with values from closest previous date
4. Saves the merged data to CSV files

Key functions:

- `load_and_prepare_data()`: Loads and prepares datasets
- `find_closest_previous_date()`: Locates closest previous date with data
- `merge_ticker_data()`: Performs the actual merging operation

6.4 Data Standardization

process_merged_data.py

Standardizes data intervals and handles null values:

1. Creates evenly spaced intervals in the dataset
2. Processes null values by distributing non-null values across consecutive nulls

Key functions:

- `standardize_intervals()`: Creates evenly spaced data points
- `process_dataframe()`: Handles non-null value distribution
- `calculate_even_indices()`: Generates evenly spaced indices

6.5 Historical Data Caching

The system implements caching of historical data to minimize API calls:

1. **Server-Side Caching**: Implemented in `server.js` with the `fetchHistoricalData()` function
2. **File-Based Cache**: Stores data in JSON files in the `historical-data` directory
3. **Cache Invalidation**: Based on data age (refreshes if older than one day)

Key benefits:

- Reduces API calls (and potential rate limiting)
- Improves response times
- Ensures data consistency across the application

---

7. Machine Learning Models

7.1 Temporal Fusion Transformer

The system incorporates a Temporal Fusion Transformer (TFT) model for time series forecasting of stock returns.

Model Architecture

The TFT model is implemented using PyTorch and PyTorch Forecasting library:

- Multi-horizon time series forecasting
- Attention mechanisms for handling temporal dependencies
- Variable selection networks for feature importance

7.2 Model Fine-tuning

Implemented in `finetune_network.py`, this module provides functionality for updating pre-trained TFT models with new financial data.

Key function: `update_model(new_data_path, old_model_path, save_path)`

This function:

1. Loads and preprocesses new financial data
2. Creates appropriate training/validation datasets
3. Loads the existing model
4. Updates the model with reduced training epochs
5. Saves the updated model

The fine-tuning process involves:

- Creating TimeSeriesDataSet objects
- Setting up a PyTorch Lightning trainer
- Configuring appropriate hyperparameters for incremental learning
- Executing the training process with early stopping
- Saving model checkpoints

7.3 Prediction Generation

The TFT model generates predictions that feed into the portfolio optimization process:

1. Model predicts expected returns for different time horizons
2. These predictions are used as input to the optimization algorithm
3. The system can balance historical returns with model predictions

Key considerations:

- Training/validation split to prevent overfitting
- Feature normalization for stable model performance
- Handling of missing values in the time series

---

8. Security Implementation

8.1 JWT Authentication

The system implements JSON Web Token (JWT) authentication:

1. **Token Generation**:
    - Created upon successful login/signup
    - Contains user email as payload
    - Signed with a secret key
2. **Token Verification**:
    - `authenticateToken` middleware extracts and verifies tokens
    - Checks signature validity and expiration
    - Attaches user information to request object
3. **Token Storage**:
    - Stored in browser's localStorage
    - Included in Authorization header for API requests

Key functions in `server.js`:

- JWT signing: `jwt.sign({ email }, SECRET_KEY)`
- JWT verification: `jwt.verify(token, SECRET_KEY, (err, user) => {...})`

8.2 Data Protection

The system implements several data protection mechanisms:

1. **Password Storage**:
    - Passwords are stored in CSV format
    - No encryption is implemented in the current version (improvement area)
2. **Data Access Control**:
    - API endpoints check authentication
    - Routes verify that users can only access their own data
    - Parameter validation to prevent unauthorized access
3. **Secure Coding Practices**:
    - Input validation for all user inputs
    - Protection against common web vulnerabilities

8.3 API Security

API security measures include:

1. **CORS Protection**:
    - Cross-Origin Resource Sharing configured to allow only the frontend origin
    - Prevents cross-site request forgery
2. **Input Validation**:
    - All request parameters are validated before processing
    - Protects against injection attacks
3. **Error Handling**:
    - Generic error messages to avoid information leakage
    - Detailed logging on the server side

---

9. Deployment Guidelines

9.1 Environment Setup

The system requires the following environment setup:

1. **Node.js Environment**:
    - Node.js (v14+)
    - npm or yarn package manager
2. **Python Environment**:
    - Python 3.8+
    - Required packages (scipy, numpy, pandas, pytorch, etc.)
    - Virtual environment recommended

9.2 Configuration

Key configuration settings:

1. **Server Configuration**:
    - PORT: Server port (default: 3001)
    - SECRET_KEY: JWT secret key
    - Database connection parameters
2. **API Keys**:
    - Finnhub API keys
    - Other external API credentials
3. **File Paths**:
    - Data storage locations
    - Credential storage paths

These are configured through environment variables or configuration files.

9.3 Deployment Process

The deployment process involves:

1. **Backend Deployment**:
    - Install Node.js dependencies: `npm install`
    - Ensure Python environment is set up
    - Start the server: `node server.js`
2. **Frontend Deployment**:
    - Build the React application: `npm run build`
    - Serve the built files via a web server
3. **Database Setup**:
    - Ensure MongoDB is running
    - Initialize required collections
4. **Environmental Configuration**:
    - Set up necessary environment variables
    - Configure API keys and secrets

---

10. Development Workflow

10.1 Environment Setup

For development, set up the environment as follows:

1. **Clone the Repository**:
```
2. git clone <repository-url>
3. cd portfolio-recommendation-system
```
4. **Install Dependencies**:
```
5. # Install Node.js dependencies
6. cd backend
7. npm install
8.
9. # Set up Python environment
```

```
cd backend
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
pip install -r requirements.txt

# Install frontend dependencies
cd frontend
npm install
```

18. **Configure Environment Variables**:
    - Create `.env` files in backend with required configuration
    - Set up API keys for external services
19. **Start Development Servers**:

```
# Start backend server
cd backend
node server.js

# Start frontend development server
cd frontend
npm start
```

10.2 Code Organization

The codebase follows this organization:

1. **Frontend Structure**:
    - `src/components/`: React components
    - `src/assets/`: Images and static assets
    - `src/App.js`: Main application component and routing
    - CSS files are co-located with their components
2. **Backend Structure**:
    - `server.js`: Main Express server
    - Python scripts for data processing and optimization
    - CSV and JSON files for data storage
3. **Coding Standards**:
    - React functional components with hooks
    - Express middleware pattern
    - Python modular design

10.3 Testing Guidelines

The system supports the following testing approaches:

1. **Frontend Testing**:
    - Component testing with React Testing Library
    - E2E testing with Cypress (not implemented but supported)
2. **Backend Testing**:
    - API testing with Postman or similar tools
    - Unit testing for Python algorithms
3. **Manual Testing**:
    - User flow testing
    - Cross-browser compatibility testing

---

11. Troubleshooting & Debugging

11.1 Common Issues

Authentication Issues

- **Token Expiration**: JWT tokens may expire, requiring re-login
- **Unauthorized Access**: Missing or invalid JWT token in requests
- **Cross-Origin Issues**: CORS configuration problems

Solutions:

- Check localStorage for valid token
- Ensure Authorization header is properly set
- Verify CORS settings in server.js

Data Fetch Issues

- **API Rate Limiting**: External APIs may impose request limits
- **Network Connectivity**: Failed API calls due to network issues
- **Incorrect API Keys**: Invalid or expired API credentials

Solutions:

- Implement exponential backoff for retries
- Add error handling for network failures
- Verify and update API keys

Portfolio Optimization Issues

- **Algorithm Convergence**: Optimization may fail to converge
- **Invalid Constraints**: Conflicting constraints may cause failures
- **Data Quality**: Poor quality data can lead to unreliable results

Solutions:

- Adjust algorithm parameters
- Check constraint compatibility
- Improve data preprocessing

11.2 Debugging Tools

Frontend Debugging

- Browser DevTools (Console, Network tab)
- React Developer Tools for component inspection
- Use of console.log statements in component code

Backend Debugging

- Node.js debugging with --inspect flag

- API testing with Postman or similar tools
- Python debugging with print statements or debuggers

11.3 Logging

The system implements basic logging:

1. **Server Logs**:
    - Console logs for significant events
    - Error logging for exceptions
2. **Algorithm Execution Logs**:
    - Python print statements for algorithm steps
    - Output capture from executed scripts

Improve logging by:

- Implementing structured logging
- Adding log levels (INFO, WARNING, ERROR)
- Configuring log file rotation

---

12. Extending the System

12.1 Adding New Components

To add new front-end components:

1. **Create Component File**:
    - Add a new .js or .jsx file in the components directory
    - Implement the component using React functional components and hooks
2. **Add Routing** (if needed):
    - Update `App.js` to include the new route
    - Use React Router's `<Route>` component
3. **Connect to Data Sources**:
    - Add API calls as needed using axios
    - Implement state management for component data

12.2 Integrating New Data Sources

To add new data sources:

1. **API Integration**:
    - Add API client configuration in the backend
    - Implement authentication if required
    - Create data retrieval functions
2. **Data Processing**:
    - Add processing logic for new data format
    - Normalize data to match existing structures
    - Update storage mechanisms if needed
3. **Front-end Integration**:

    - Create new API endpoints for the data
    - Update components to display the new data

12.3 Implementing New Algorithms

To implement new portfolio optimization algorithms:

1. **Algorithm Implementation**:
    - Create a new Python script or module
    - Implement the algorithm following the existing structure
    - Add appropriate documentation
2. **Integration with Existing System**:
    - Update `portfolio_construction.py` to include the new algorithm
    - Add selection logic to choose between algorithms
    - Ensure consistent input/output formats
3. **Testing and Validation**:
    - Test the algorithm with sample data
    - Compare results with existing algorithms
    - Validate against expected outputs

---

13. API Reference

13.1 User Management

POST /login

Authenticates a user and generates a JWT token.

Request Body:

```
{
  "email": "user@example.com",
  "password": "password123"
}
```

Response:

```
{
  "message": "Login successful",
  "token": "jwt-token-string"
}
```

POST /signup

Creates a new user account.

Request Body:

```
{
  "email": "newuser@example.com",
  "password": "password123"
```

```
}
```

Response:

```
{
  "message": "Account created successfully!",
  "token": "jwt-token-string"
}
```

DELETE /delete-user/:email

Deletes a user and all associated data.

Parameters:

- `email`: User email to delete

Response:

```
{
  "message": "User and all associated data deleted successfully"
}
```

13.2 Portfolio Management

POST /save-preferences

Saves user investment preferences and generates portfolio.

Request Body:

```
{
  "email": "user@example.com",
  "familiar": "2",
  "portfolioDrop": "2",
  "riskInvestments": "2",
  "volatility": "2",
  "drawdown1": "2",
  "drawdown2": "2",
  "drawdown3": "2",
  "investmentHorizon": "6",
  "initialInvestment": "10000",
  "portfolioSize": "20"
}
```

Response:

```
Preferences save
```