Detailed ER Diagram Description

Overview

In the Stock Market Portfolio application, the design uses a single collection, Stock, for both storing stock data and managing user watchlists. This approach simplifies the database schema by avoiding the need for a separate collection for the watchlist. Here's a detailed explanation:

ER Diagram:

1. Collection Overview

Collection Name: Stock

Purpose:

- **Data Retrieval**: This collection stores all stock-related information which is retrieved and displayed to users.
- **Watchlist Management**: It also manages user watchlists by storing stocks that users want to keep track of. There is no separate Watchlist collection; instead, the watchlist functionality is integrated into the Stock collection.

2. Data Fields

- **company**: Represents the name of the company associated with the stock (e.g., "Apple Inc.").
- **description**: Provides a brief description of the company or the stock (e.g., "Technology company specializing in consumer electronics").
- **initial_price**: Records the initial price of the stock when it was first listed (e.g., $50).
- **price_2002**: Contains the price of the stock in the year 2002 (e.g., $25).
- **price_2007**: Contains the price of the stock in the year 2007 (e.g., $40).
- **symbol**: A unique identifier or ticker symbol for the stock (e.g., "AAPL" for Apple Inc.).

3. Functionality

Retrieval of Stock Data:

- **GET Requests**: The application retrieves stock data from the Stock collection using GET requests to display stock information on the frontend.

Adding to Watchlist:

- **POST Requests**: Users can add stocks to their watchlist by making POST requests to the backend. The same Stock collection is used to handle these requests.
    - **Adding New Stocks**: When a new stock is added to the watchlist, it is inserted as a new document into the Stock collection.
    - **Updating Existing Stocks**: If the stock already exists, it can be updated with new information or marked as part of the user's watchlist.

4. Design Rationale

Unified Collection:

- **Simplicity**: Using a single collection for both stock data and watchlist management reduces complexity in the schema and simplifies data access and manipulation.
- **Efficiency**: This design eliminates the need for joins or multiple queries to manage and retrieve watchlist items, improving performance and maintainability.

5. Example Document

Here's an example of how a stock document might look in the Stock collection:

json


```
{
  "_id": "ObjectId('...')",
  "company": "Apple Inc.",
  "description": "Technology company specializing in consumer electronics.",
  "initial_price": 50,
  "price_2002": 25,
  "price_2007": 40,
  "symbol": "AAPL"
}
```


6. Benefits

- **Ease of Use**: Users can view and manage their watchlist using the same interface as for viewing all stocks.

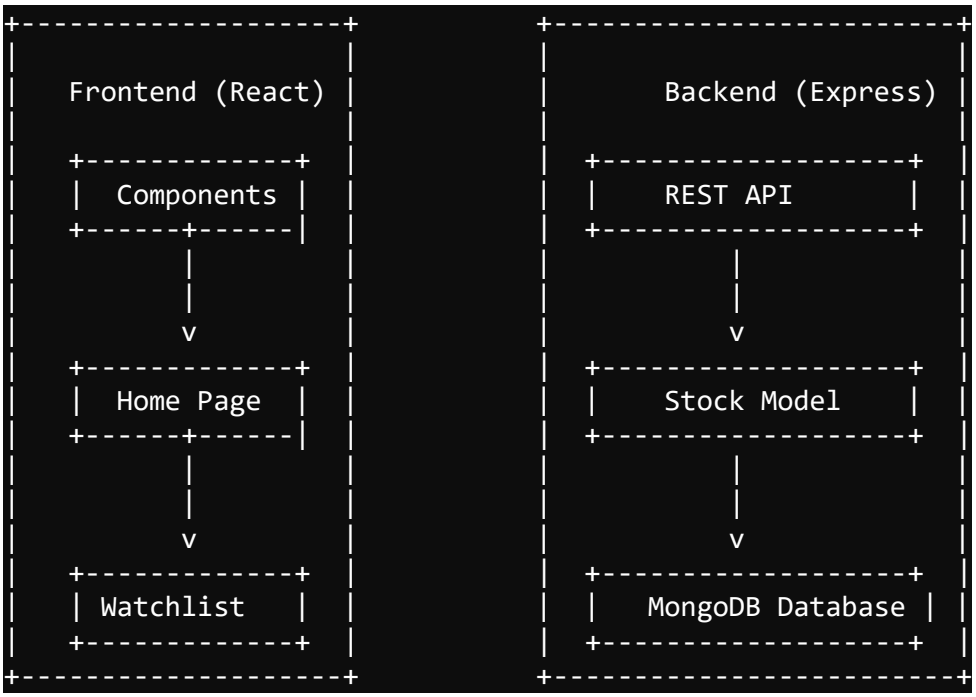
- **Streamlined Operations**: Backend operations are streamlined by handling all stock-related requests within a single collection.

Conclusion

The design choice of using a single Stock collection for both stock data retrieval and watchlist management is efficient and straightforward, making the application easier to develop and maintain.

Architecture Diagram Description

The architecture diagram provides a visual representation of the Stock Market Portfolio application's components and their interactions. Here's a detailed breakdown of each element in the diagram:



- **Frontend (React)**: Manages the user interface and communicates with the backend via API requests.
- **Backend (Express)**: Processes API requests, interacts with the database, and sends responses.
- **Database (MongoDB)**: Stores and manages the application's data.

This architecture ensures a clear separation of concerns, where the frontend handles user interactions, the backend manages business logic and data processing, and the database stores persistent data.

1. Frontend (React)

Components:

- **Components**: This represents the various React components that make up the user interface. Components are modular and reusable pieces of the UI, such as buttons, lists, and forms.
    - **Home Page**: A component that likely serves as the main landing page or dashboard of the application.

- **Watchlist**: A component that displays the stocks a user has added to their watchlist.

Flow:

- The frontend communicates with the backend to fetch data and update the user interface based on user actions.

2. Backend (Express)

REST API:

- **REST API**: Represents the set of endpoints exposed by the backend to handle requests from the frontend. These include:
  - **GET /api/stocks**: Retrieves a list of stocks from the database.
  - **POST /api/watchlist**: Adds a new stock to the watchlist in the database.

Stock Model:

- **Stock Model**: Defines the schema for stock documents in the MongoDB database. It specifies the structure of stock data, including fields like company, description, initial_price, and symbol.

Flow:

- The backend processes requests from the frontend, interacts with the database to perform CRUD operations, and sends responses back to the frontend.

3. Database (MongoDB)

MongoDB Database:

- **MongoDB Database**: Stores the application's data. In this case, it contains:
  - **Stock Collection**: Holds documents with information about each stock, such as company details, prices, and symbols.

Flow:

- The backend interacts with MongoDB to store and retrieve stock data based on API requests. MongoDB handles the data persistence and querying.

Visual Representation of Data Flow

1. **Frontend Interaction**:
   - The **React frontend** requests data from the **Express backend** via API endpoints.
   - The **Stocks component** fetches stock data from the /api/stocks endpoint and displays it to users.
   - When a user adds a stock to their watchlist, the **Watchlist component** makes a **POST** request to /api/watchlist to save the stock data.

2. **Backend Processing**:
    - The **Express backend** receives requests from the frontend and interacts with the **MongoDB database** to fetch or store data.
    - The backend processes **GET** requests to retrieve stock information and **POST** requests to add stocks to the watchlist.
3. **Database Operations**:
    - The **MongoDB database** stores the stock data and watchlist entries.
    - The backend uses Mongoose to define the **stock model** and perform database operations, ensuring data is saved and retrieved correctly.

Stock Market Portfolio Application Documentation

Introduction

The Stock Market Portfolio application is a MERN stack-based web tool designed for effective management and tracking of stock investments. It allows users to view stock details, add stocks to a watchlist, and monitor price changes. The application uses a MongoDB database to store stock information, which is critical for providing users with up-to-date and accurate data about their investments.

Database Schema

MongoDB Schema:

The MongoDB schema is designed using Mongoose to structure the stock data. The schema includes a single collection, Stock, which serves both as the primary data store for stock details and for managing user watchlists. Here are the details of the schema:

- **id:** ObjectId (Primary Key)
  A unique identifier for each stock entry in the database.
- **company:** String
  The name of the company associated with the stock.
- **description:** String
  A brief description of the company or stock.
- **initial_price:** Number
  The price of the stock at the time of its initial listing.
- **price_2002:** Number
  The stock price in the year 2002.
- **price_2007:** Number
  The stock price in the year 2007.
- **symbol:** String
  The stock's ticker symbol.

ER Diagram

The ER diagram reflects that the same Stock collection is used for both retrieving stock data and adding new entries (acting as a watchlist). There is no separate Watchlist collection; instead, the Stock collection serves this purpose.

- The same Stock collection is used for both retrieving stock data and adding new stock entries, acting as the watchlist. There is no separate Watchlist collection.

Backend Code

Server Setup:

javascript

```
const express = require("express");
const mongoose = require("mongoose");
const cors = require("cors");
const bodyParser = require("body-parser");
const app = express();
const PORT = process.env.PORT || 4000; // Changed the port number
// Middleware
app.use(cors());
app.use(bodyParser.json());
// MongoDB Connection URI
const mongoURI = "mongodb://localhost:27017/Stock_Market"; // Updated the database name
mongoose.connect(mongoURI, {
  useNewUrlParser: true,
  useUnifiedTopology: true,
})
.then(() => console.log("MongoDB connected"))
.catch(err => console.error("MongoDB connection error:", err));
// Schema and Model
const stockSchema = new mongoose.Schema({
  company: String,
  description: String,
  initial_price: Number,
  price_2002: Number,
  price_2007: Number,
  symbol: String,
});
```

```
const Stock = mongoose.model("Stock", stockSchema);
// Routes
app.get("/api/stocks", async (req, res) => {
  try {
    const stocks = await Stock.find();
    res.json(stocks);
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: "Internal Server Error" });
  }
});
app.post("/api/watchlist", async (req, res) => {
  try {
    const {
      company,
      description,
      initial_price,
      price_2002,
      price_2007,
      symbol,
    } = req.body;
    const stock = new Stock({
      company,
      description,
      initial_price,
      price_2002,
      price_2007,
      symbol,
    });
    await stock.save();
```

```
res.json({ message: "Stock added to watchlist successfully" });
    } catch (error) {
        console.error(error);
        res.status(500).json({ error: "Internal Server Error" });
    }
});

// Start Server
app.listen(PORT, () => {
    console.log(`Server is running on port ${PORT}`);
});
```

MongoDB Queries

Retrieve Data (GET):

- **Endpoint:** GET /api/stocks
- **Description:** Fetches a list of all stocks from the Stock collection.
- **Postman Request:**
    - Method: GET
    - URL: http://localhost:4000/api/stocks
    - Headers: None needed
    - Body: Not applicable for GET requests

Create Data (POST):

- **Endpoint:** POST /api/watchlist
- **Description:** Adds a new stock to the Stock collection, which acts as the watchlist.
- **Request Body:**

```
json
Copy code
{
  "company": "Apple",
  "description": "Technology company",
  "initial_price": 150.25,
```

"price_2002": 120.30,

"price_2007": 175.50,

"symbol": "AAPL"

}

- **Postman Request:**
    - Method: POST
    - URL: http://localhost:4000/api/watchlist
    - Headers: Content-Type: application/json
    - Body: Raw JSON data as shown above

Data Storage and Retrieval

Data Stored:

The database stores detailed information about various stocks, which is crucial for users to manage and track their stock investments. Each stock entry in the database includes the following data:

- **_id:** A unique identifier (ObjectId) assigned to each stock entry. This field is automatically generated by MongoDB.
- **company:** The name of the company associated with the stock.
- **description:** A brief description of the company's operations or the stock itself.
- **initial_price:** The price of the stock at the time of its initial listing.
- **price_2002:** The stock price recorded in the year 2002.
- **price_2007:** The stock price recorded in the year 2007.
- **symbol:** The stock's ticker symbol used for trading.

Data Entries:

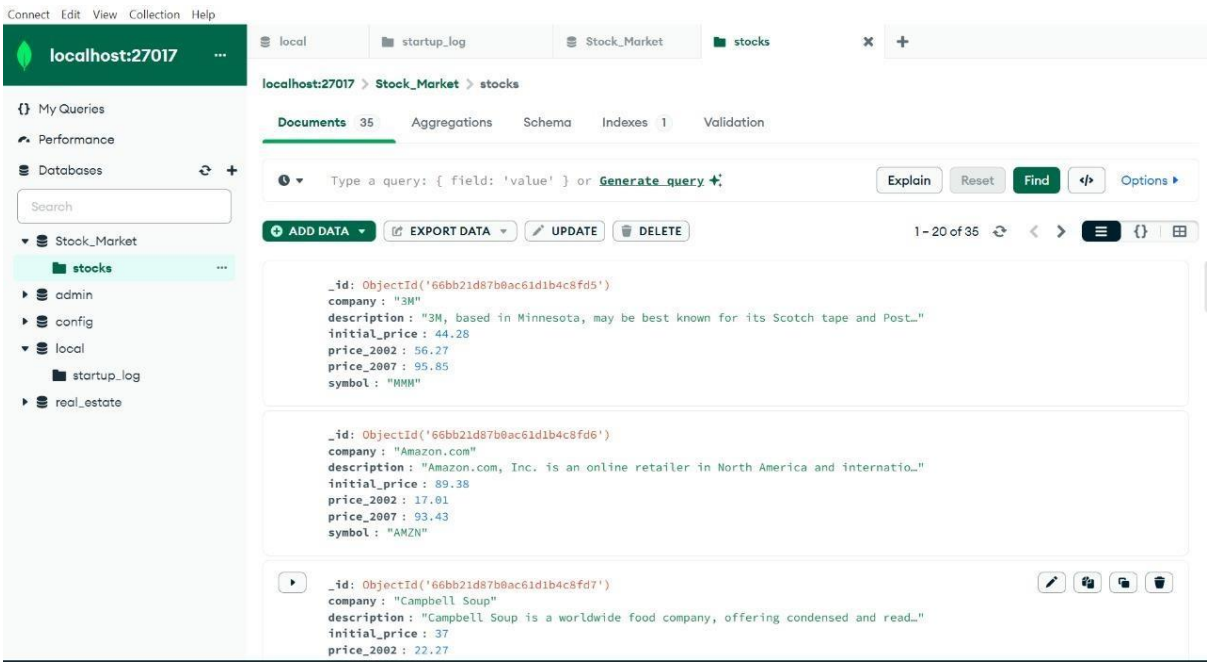
1. **Apple Inc.**
    - **company:** Apple Inc.
    - **description**: Technology company known for its iPhones, iPads, and Mac computers.
    - **initial_price:** 150.25
    - **price_2002:** 120.30
    - **price_2007:** 175.50
    - **symbol:** AAPL
2. **Microsoft Corporation**
    - **company**: Microsoft Corporation
    - **description:** Leading technology company specializing in software, hardware, and cloud services.
    - **initial_price:** 290.40
    - **price_2002:** 52.70
    - **price_2007:** 32.90

- **symbol:** MSFT

3. **Tesla, Inc.**
   - **company**: Tesla, Inc.
   - **description:** Electric vehicle and clean energy company.
   - **initial_price:** 720.50
   - **price_2002:** 14.40
   - **price_2007:** 33.70
   - **symbol:** TSLA
4. **Amazon.com, Inc.**
   - **company:** Amazon.com, Inc.
   - **description:** E-commerce and cloud computing giant.
   - **initial_price:** 3500.15
   - **price_2002:** 25.20
   - **price_2007:** 90.50
   - **symbol:** AMZN
5. **Google LLC (Alphabet Inc.)**
   - **company:** Google LLC
   - **description:** Multinational technology company known for its search engine and advertising services.
   - **initial_price:** 2700.75
   - **price_2002:** 100.10
   - **price_2007:** 400.50
   - **symbol:** GOOGL

MongoDB Compass:

Stocks Collection: The stocks data has stored in the stocks collection.

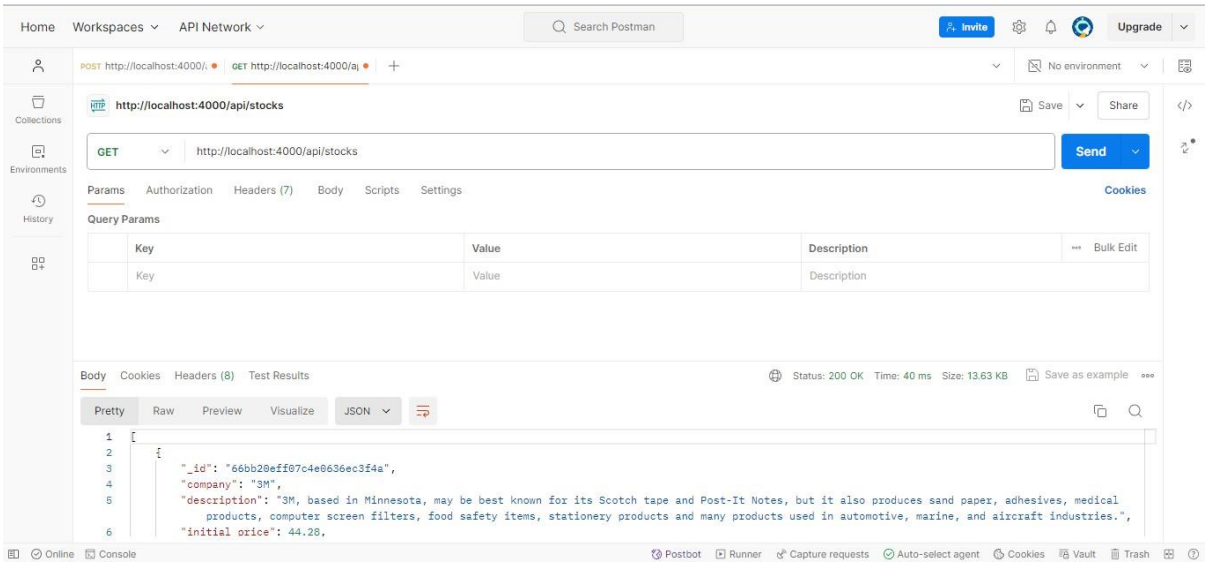


Data Retrieval

The GET /api/stocks endpoint retrieves all stock entries from the database and displays them on the Stocks page. Users can then add any stock to their watchlist by clicking the "Add to My Watchlist" button. This action triggers a POST request to the /api/watchlist endpoint, which adds the selected stock to the database.

1.Retrieve Data (GET):

URL: http://localhost:4000/api/stocks



2.Create Data (POST)

URL: http://localhost:4000/api/watchlist

Inserting the data using Postman:

{

"company": "Apple",

"description": "Technology company",

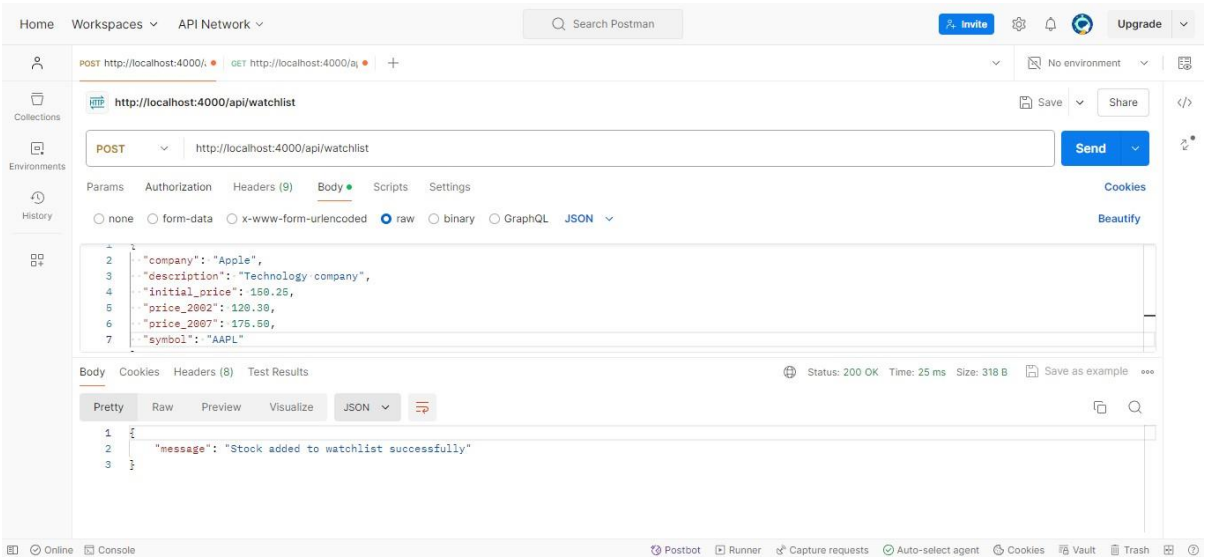
"initial_price": 150.25,

"price_2002": 120.30,

"price_2007": 175.50,

"symbol": "AAPL"

}



Final Web Page:

The web page integrates with the backend to fetch stock data and update the watchlist. The Stocks page allows users to view available stocks and add them to their watchlist. The Watchlist page displays the stocks that have been added by the user.


Stocks
Watchlist
Stock Market MERN App
My Watchlist
3M (MMM) -
$44.28
Amazon.com (AMZN) -
$89.38
Exxon Mobil (XOM) -
$39
The Gap (GPS) -
$46

Stock Market Portfolio Application

Frontend Description

The frontend of the application is built using React and provides a user interface for interacting with stock data and managing a watchlist. The user can view a list of stocks, add stocks to their watchlist, and navigate between different views.

1. **App.js**

```
// src/App.js

import React, { useState, useEffect } from "react";
import {
  BrowserRouter as Router,
  Routes,
  Route,
  NavLink,
} from "react-router-dom";
import "./App.css";

const Stocks = ({ addToWatchlist }) => {
  const [stocks, setStocks] = useState([]);

  useEffect(() => {
    // Fetch stock data from the backend
    fetch("http://localhost:4000/api/stocks")
      .then((res) => res.json())
      .then((data) => setStocks(data))
      .catch((error) => console.error("Error fetching stocks:", error));
  }, []);
  console.log(setStocks, "Stocksdata");

  const getRandomColor = () => {
    const colors = ["#FF0000", "#00FF00"]; // Red and Green
    return colors[Math.floor(Math.random() * colors.length)];
  };

  return (
    <div className="App">
      <h1>Stock Market MERN App</h1>
      <h2>Stocks</h2>
      <ul>
        {stocks.map((stock) => (
          <li key={stock.symbol}>
            {stock.company} ({stock.symbol}) -
            <span style={{ color: getRandomColor() }}>
```

```
                    {" "}
                    ${stock.initial_price}
                  </span>
                  <button onClick={() => addToWatchlist(stock)}>
                    Add to My Watchlist
                  </button>
                </li>
              ))}
            </ul>
          </div>
        );
      };

      const Watchlist = ({ watchlist }) => {
        const getRandomColor = () => {
          const colors = ["#FF0000", "#00FF00"]; // Red and Green
          return colors[Math.floor(Math.random() * colors.length)];
        };

        return (
          <div className="App">
            <h1>Stock Market MERN App</h1>
            <h2>My Watchlist</h2>
            <ul>
              {watchlist.map((stock) => (
                <li key={stock.symbol}>
                  {stock.company} ({stock.symbol}) -
                  <span style={{ color: getRandomColor() }}>
                    {" "}
                    ${stock.initial_price}
                  </span>
                </li>
              ))}
            </ul>
          </div>
        );
      };

      function App() {
        const [watchlist, setWatchlist] = useState([]);

        const addToWatchlist = (stock) => {
          // Add stock to watchlist
          fetch("http://localhost:4000/api/watchlist", {
            method: "POST",
            headers: {
```

```
            "Content-Type": "application/json",
          },
          body: JSON.stringify(stock),
        })
          .then((res) => res.json())
          .then((data) => {
            // Show an alert with the message received from the server
            alert(data.message);
            setWatchlist([...watchlist, stock]);
          })
          .catch((error) =>
            console.error("Error adding to watchlist:", error)
          );
      };

      return (
        <Router>
          <nav>
            <NavLink to="/stocks">Stocks</NavLink>
            <NavLink to="/watchlist">Watchlist</NavLink>
          </nav>
          <Routes>
            <Route
              path="/stocks"
              element={<Stocks addToWatchlist={addToWatchlist} />}
            />
            <Route
              path="/watchlist"
              element={<Watchlist watchlist={watchlist} />}
            />
          </Routes>
        </Router>
      );
    }

    export default App;
```

- **Purpose**: Acts as the main component that sets up routing and handles state management for the watchlist.
- **Components**:
    - **Stocks**: Displays a list of all stocks fetched from the backend.
    - **Watchlist**: Shows stocks that the user has added to their watchlist.

Process and Output:

1. **Routing**:
    - **<Router>**: Uses React Router to handle navigation.
    - **<Routes>** and **<Route>**: Define routes for /stocks and /watchlist paths.
2. **State Management**:
    - **useState**: Manages the watchlist state (watchlist).
    - **addToWatchlist Function**: Adds a stock to the watchlist by sending a POST request to the backend.
3. **Components**:
    - **Stocks Component**:
        - Fetches stock data from http://localhost:5000/api/stocks using fetch and useEffect.
        - Displays stocks with their company names, symbols, and initial prices.
        - Uses getRandomColor to dynamically color the price based on a random selection between red and green.
        - Provides an "Add to My Watchlist" button to add stocks to the watchlist.
    - **Watchlist Component**:
        - Displays stocks added to the watchlist.
        - Also uses getRandomColor to color the prices.

Expected Output:

- **Stocks Page**:
    - Displays a list of stocks with company names, symbols, and colored prices.
    - Each stock has a button to add it to the watchlist.
- **Watchlist Page**:
    - Shows the stocks that have been added to the user's watchlist with colored prices.

2. App.css

```
/* src/App.css */

body {
  font-family: 'Arial', sans-serif;
  background-color: #d9d7ca;
  margin: 0;
  padding: 0;
```

```
}

.App {
  text-align: center;
  padding: 20px;
}

h1 {
  color: #1f454d;
}

h2 {
  color: #3c8d93;
  margin-top: 30px;
}

ul {
  list-style-type: none;
  padding: 0;
}

li {
  background-color: #3c8d93;
  color: #d9d7ca;
  padding: 10px;
  margin: 10px 0;
  border-radius: 5px;
  display: flex;
  justify-content: space-between;
  align-items: center;
}

button {
  background-color: #1f454d;
  color: #d9d7ca;
  border: none;
  padding: 8px;
  border-radius: 5px;
  cursor: pointer;
  transition: background-color 0.3s ease;
}

button:hover {
  background-color: #3c8d93;
}
```

```
/* Navigation bar styles */
nav {
  background-color: #1f454d;
  padding: 15px 0;
}

nav a {
  color: #d9d7ca;
  text-decoration: none;
  margin: 0 20px;
  font-size: 18px;
  transition: color 0.3s ease;
}

nav a:hover {
  color: #3c8d93;
}
```

- **Purpose**: Provides styling for the application to ensure a cohesive and visually appealing user interface.

Styling Details:

- **General Styles**:
    - Sets font, background color, and margin for the body.
    - Styles .App to center text and add padding.
- **Text and Lists**:
    - Styles headers (h1 and h2) with colors and margins.
    - Styles the list (ul and li) with background colors, padding, and alignment.
- **Buttons and Navigation**:
    - Styles buttons with color, padding, and hover effects.
    - Styles navigation bar and links with colors and hover effects.

Backend Description

The backend of the application is built using Node.js with Express and MongoDB. It manages the stock data and handles requests to fetch stock information and update the watchlist.

1. **server.js**

```
const express = require("express");
const mongoose = require("mongoose");
const cors = require("cors");
```

```
const bodyParser = require("body-parser");

const app = express();
const PORT = process.env.PORT || 4000;

app.use(cors());
app.use(bodyParser.json());

mongoose.connect("mongodb://localhost:27017/Stock_Market", {
  useNewUrlParser: true,
  useUnifiedTopology: true,
});

const stockSchema = new mongoose.Schema({
  company: String,
  description: String,
  initial_price: Number,
  price_2002: Number,
  price_2007: Number,
  symbol: String,
});

const Stock = mongoose.model("Stock", stockSchema);

app.get("/api/stocks", async (req, res) => {
  try {
    const stocks = await Stock.find();
    res.json(stocks);
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: "Internal Server Error" });
  }
});

app.post("/api/watchlist", async (req, res) => {
  try {
    const {
      company,
      description,
      initial_price,
      price_2002,
      price_2007,
      symbol,
    } = req.body;
    const stock = new Stock({
      company,
```

```
            description,
            initial_price,
            price_2002,
            price_2007,
            symbol,
        });
        await stock.save();
        res.json({ message: "Stock added to watchlist successfully" });
    } catch (error) {
        console.error(error);
        res.status(500).json({ error: "Internal Server Error" });
    }
});

app.listen(PORT, () => {
    console.log(`Server is running on port ${PORT}`);
});
```

- **Purpose**: Sets up the Express server, connects to MongoDB, and defines API endpoints for stock data and watchlist management.

Process and Output:

1. **Server Setup**:
    - Initializes Express and configures middleware (cors and bodyParser).
    - Connects to MongoDB using Mongoose with connection settings.
2. **Schema Definition**:
    - **stockSchema**: Defines the schema for stock data with fields such as company, description, initial_price, price_2002, price_2007, and symbol.
3. **API Endpoints**:
    - **GET /api/stocks**:
        - Fetches all stock records from MongoDB.
        - Returns stock data as a JSON response.
        - **Output**: A JSON array of stock objects, each with fields defined in the schema.
    - **POST /api/watchlist**:
        - Accepts a stock object in the request body.
        - Saves the stock to MongoDB.
        - Returns a success message upon successful addition.

- **Output**: A JSON response with a message confirming the stock was added to the watchlist.

Expected Output:

- **Stock Data**: A list of stock objects in JSON format, available on the /api/stocks endpoint.
- **Watchlist Addition**: A confirmation message in JSON format when a stock is successfully added to the watchlist via the /api/watchlist endpoint.

Interaction Between Frontend and Backend

1. **Fetching Stocks**:
    - The Stocks component in the frontend makes a GET request to the /api/stocks endpoint to retrieve stock data.
    - The backend responds with a JSON array of stock objects.
2. **Adding to Watchlist**:
    - When a user clicks the "Add to My Watchlist" button, the addToWatchlist function sends a POST request to the /api/watchlist endpoint with the stock data.
    - The backend saves this data to MongoDB and responds with a success message.
3. **Displaying Data**:
    - The frontend displays the retrieved stock data and watchlist items, with prices dynamically colored and interactive elements for user actions.

Detailed Description of Backend Code

Backend Code:

```
const express = require("express");
const mongoose = require("mongoose");
const cors = require("cors");
const bodyParser = require("body-parser");
const app = express();
const PORT = process.env.PORT || 4000;
app.use(cors());
app.use(bodyParser.json());
mongoose.connect("mongodb://localhost:27017/Stock_Market", {
   useNewUrlParser: true,
   useUnifiedTopology: true,
});
const stockSchema = new mongoose.Schema({
   company: String,
   description: String,
   initial_price: Number,
   price_2002: Number,
   price_2007: Number,
   symbol: String,
});
const Stock = mongoose.model("Stock", stockSchema);
app.get("/api/stocks", async (req, res) => {
   try {
      const stocks = await Stock.find();
      res.json(stocks);
   } catch (error) {
      console.error(error);
```

```
        res.status(500).json({ error: "Internal Server Error" });
    }});
app.post("/api/watchlist", async (req, res) => {
    try {
        const {
            company,
            description,
            initial_price,
            price_2002,
            price_2007,
            symbol,
        } = req.body;
        const stock = new Stock({
            company,
            description,
            initial_price,
            price_2002,
            price_2007,
            symbol,
        });
        await stock.save();
        res.json({ message: "Stock added to watchlist successfully" });
    } catch (error) {
        console.error(error);
        res.status(500).json({ error: "Internal Server Error" });
    }
});
app.listen(PORT, () => {
    console.log(`Server is running on port ${PORT}`);
});
```

1. Dependencies and Setup

- **express**:
    - A web application framework for Node.js that simplifies the creation of web servers and APIs. It provides a robust set of features for web and mobile applications.
- **mongoose**:
    - An Object Data Modeling (ODM) library for MongoDB and Node.js. It provides a schema-based solution to model application data, manage database interactions, and validate data.
- **cors**:
    - Middleware that enables Cross-Origin Resource Sharing. It allows your backend server to respond to requests from different origins, which is useful for front-end applications making requests to the server from a different domain.
- **body-parser**:
    - Middleware to parse incoming request bodies before your handlers. It parses JSON payloads and makes them available as JavaScript objects in your route handlers.

2. Middleware Configuration

- **app.use(cors())**:
    - Configures the server to allow requests from different origins. This is essential for front-end applications that need to interact with the backend server from a different domain.
- **app.use(bodyParser.json())**:
    - Configures the server to parse JSON request bodies. This allows the server to handle JSON data sent in POST requests and convert it into JavaScript objects accessible via req.body.

3. Database Connection

- **mongoose.connect("mongodb://localhost:27017/Stock_Market", {...})**:
    - Establishes a connection to a MongoDB database hosted locally at mongodb://localhost:27017/Stock_Market.
    - **useNewUrlParser: true**: Uses the new URL parser engine to handle MongoDB connection strings.
    - **useUnifiedTopology: true**: Uses the new Server Discover and Monitoring engine, providing a more robust and stable connection experience.

4. Stock Schema and Model

- **const stockSchema = new mongoose.Schema({...})**:
    - Defines the schema for the Stock collection in MongoDB. The schema specifies the structure of the documents, including the data types for each field:
        - **company**: A String representing the name of the company.
        - **description**: A String providing a description of the stock.
        - **initial_price**: A Number for the stock's initial price.
        - **price_2002**: A Number representing the stock price in 2002.
        - **price_2007**: A Number representing the stock price in 2007.
        - **symbol**: A String for the stock's ticker symbol.
- **const Stock = mongoose.model("Stock", stockSchema)**:
    - Creates a model named Stock using the schema defined. This model allows interaction with the Stock collection in MongoDB, enabling CRUD operations such as creating, reading, updating, and deleting stock documents.

5. API Endpoints

- **GET /api/stocks**:
    - **Purpose**: Fetches all stock documents from the database.
    - **Implementation**:
        - Uses Stock.find() to retrieve all documents from the Stock collection.
        - Sends the retrieved data as a JSON response to the client.
        - Handles errors by logging them and sending a 500 Internal Server Error status code with an error message if an error occurs.
- **POST /api/watchlist**:
    - **Purpose**: Adds a new stock document to the Stock collection.
    - **Implementation**:
        - Extracts stock details from the request body.
        - Creates a new Stock instance using the provided data.
        - Saves the new stock document to the database using stock.save().
        - Responds with a success message if the operation is successful.
        - Handles errors by logging them and sending a 500 Internal Server Error status code with an error message if an error occurs.

6. Server Setup

- **app.listen(PORT, () => {...})**:
    - Starts the server and listens on the specified port, defaulting to 4000.
    - Logs a message indicating that the server is running and listening on the specified port.

Overall Architecture

The backend code is structured to handle API requests and interact with a MongoDB database. It defines a schema for the Stock collection, provides endpoints for fetching and adding stocks, and uses middleware for request handling. The integration with MongoDB through Mongoose ensures efficient data management and interaction, while middleware components handle cross-origin requests and JSON parsing.