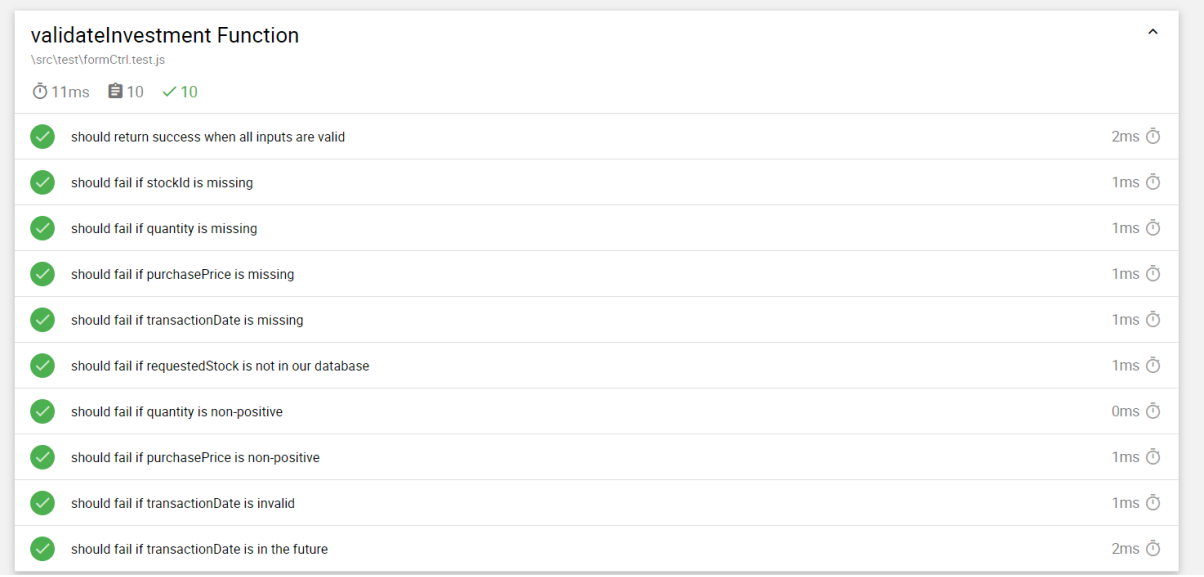
Unit Testing

Unit Test on Add Investment Controller Form:




```
export function validateInvestment(input) {
    const { stockId, quantity, purchasePrice, transactionDate } = input;

    const stockData = JSON.parse(fs.readFileSync('./public/allstocks.json', 'utf8'));
    // Check if all fields are provided
    if (
        stockId === undefined ||
        quantity === undefined ||
        purchasePrice === undefined ||
        transactionDate === undefined
    ) {
        return { success: false, error: "All fields (stockId, quantity, purchasePrice, transactionDate) are required.
    }

    const stockExists = stockData.companies.some(company => company.symbol === stockId);
    if (!stockExists) {
```

Summary of Unit Tests for validateInvestment

The unit tests for the `validateInvestment` function validate various input scenarios to ensure the function operates correctly and adheres to expected business rules. The test cases cover the following scenarios:

1. Valid Inputs:

- Ensures the function returns success when all required fields are present, correctly formatted, and meet the validation criteria.

2. Missing Fields:

- Tests cases where one or more required fields (stockId, quantity, purchasePrice, transactionDate) are omitted. These tests verify that the function correctly identifies the missing data and returns an appropriate error message.

3. Stock Existence:

- Validates that the function checks for the existence of the stock in the provided database. If the **stock ID** is invalid or not found, it ensures the appropriate error is returned.

4. Quantity Validation:

- Ensures the **quantity** field is a positive number. If the value is zero, negative, or invalid, the function should return an error.

5. Purchase Price Validation:

- Verifies that the **purchasePrice** is a positive number. Negative or zero values should trigger a failure response.

6. Transaction Date Validation:

- Confirms the function handles transaction **date** validations, including:
  - Rejecting invalid date formats.
  - Ensuring the date is not in the future.

Each test asserts the expected output against the actual result from the 'validateInvestment' function, ensuring correctness and reliability under diverse input conditions.

Unit Test on Top Gainers and Losers Function:

```
TopGainersAndLosers
  ✓ should correctly calculate gain/loss for valid transactions and stock details
  ✓ should return an empty array when transactions are empty
```

```
export function TopGainersAndLosers({ stockDetails, transactions }) {
  // console.log('Transactions and Stock Details:', transactions, stockDetails);

  const data2 = transactions ? transactions.map((transaction) => {
    const stockDetail = stockDetails.find((s) => s.stockId === transaction.stockId);

    // Check if stockDetail and the re                any d price information are available
    if (!stockDetail || !stockDetail.result || !stockDetail.result.price?.regularMarketPrice) return null;

    // Extract the regularMarketPrice
    const regularMarketPrice = stockDetail.result.price.regularMarketPrice;

    // Calculate gain/loss percentage
    const gainLoss = ((regularMarketPrice - transaction.purchasePrice) / transaction.purchasePrice) * 100;

    return {
      stockId: transaction.stockId,
      name: stockDetail.result.price.longName || stockDetail.result.price.shortName || transaction.stockId,
      gain: gainLoss,
    };
  }): [];

  // Filter out null or undefined entries and sort by gain/loss
  const data = data2.filter(Boolean).sort((a, b) => b.gain - a.gain);

  // console.log(data, data2);
```

Summary of Unit Test for Top Gainers and Losers Function:

1. **Valid Transactions and Stock Details**:
   - Ensures accurate calculation of percentage gain/loss based on the difference between purchase price and current market price. Verifies correct mapping of stock names.
2. **Empty Transactions**:
   - Verifies the function returns an empty array when no transactions are provided, handling edge cases gracefully.