



Eötvös Loránd University
Faculty of Informatics
Department of Media And Educational
Informatics

Web application for tracking cryptocurrency portfolio

Győző Horváth, PhD
Senior Lecturer

Bence Knáb
Computer Scientist, BSc

Budapest, 2019

Contents

1	Introduction	1
1.1	Cryptocurrencies	1
1.2	Portfolio	1
1.3	Task description	1
1.4	Motivation	1
2	User Documentation	3
2.1	Short description	3
2.2	System requirements	3
2.3	How to use the application	3
2.3.1	Website structure	3
2.3.2	Authentication	9
2.3.3	Creating a new portfolio	11
2.3.4	Creating a new transaction	11
2.3.5	Updating a transaction	13
2.3.6	Deleting a transaction	14
2.3.7	Filtering the portfolio	14
2.3.8	Changing the portfolio graph start date	14
3	Developer documentation	16
3.1	High-level design	16
3.2	Architecture design	16
3.2.1	Database	17
3.2.2	Backend server	18
3.2.3	Market data API	19
3.2.4	Frontend client application	19
3.3	Development environment	19
3.4	Common technologies	21
3.5	MongoDB database	22
3.5.1	Users	22
3.5.2	Transactions	23
3.5.3	Tags	23
3.6	Express REST API service	23
3.6.1	Models	25
3.6.2	CryptoCompare API	26
3.6.3	Controllers	27
3.6.4	Routes	28

3.7	React application	32
3.7.1	Hooks	33
3.7.2	Contexts	34
3.7.3	Components	35
3.8	Testing	38
3.8.1	Integration tests	39
3.8.2	End-to-End tests	40
3.8.3	Manual tests	41
3.9	Deployment	44
3.10	Possibilities for further development	45
4	References	46

1 Introduction

1.1 Cryptocurrencies

A cryptocurrency [1] is a digital asset that uses cryptography to secure financial transactions, from which holding of an asset can be verified in a decentralized manner as opposed to a central authority. In the cryptocurrency industry, cryptocurrencies are sometimes referred to as simply coins. Bitcoin [2] is considered to be the first cryptocurrency, released in 2009. Since the release of Bitcoin, several thousand other similar digital assets [3] were launched.

1.2 Portfolio

In finance, a portfolio [4] is a grouping of investments held by an investor and/or financial institutions. In the context of this thesis, the term "position" will be used for transactions which involve the same cryptocurrency. The portfolio will be a collection of these positions.

1.3 Task description

The topic of my thesis is a web application in which a user can keep track of their cryptocurrency portfolio. The goal is to provide an application that aggregates the input transactions into individual positions and portfolios, and calculates relevant financial information about them using real-time market data. The total portfolio value is also available on a historical chart until the date of the earliest transaction. The unrealized profit or loss is computed for every investment from the related cryptocurrency's current price, and an average profit or loss amount is estimated for the whole position. Each position can be supplemented with optional tags, which allows the user to analyze the relative performance of these specific groups.

1.4 Motivation

The motivation for this subject primarily comes from my fascination with the disruptive force of cryptocurrencies and partly from my experiences developing web applications during my university studies and internship. I've decided to merge these two areas in my thesis when I couldn't find simple, free alternatives for basic portfolio management; paid apps have too many advanced features which I wouldn't utilize and free apps have strict limitations on the number of transactions. I've had the opportunity to try the *React* [5] *JavaScript* [6] library during my internship and I've been using it for personal projects ever since, that's why this library was an

obvious choice for the frontend browser application. For the backend application, I've chosen the *Express* [7] *Node.js* [8] framework because I didn't have substantial experience with backend *JavaScript* technologies – due to mainly focusing on *Java* [9] before – but since I've started experimenting with *ECMAScript 6* [10] while developing modern frontend applications I always wanted to try *JavaScript* in a different environment. I aspired to develop a complete web application applying my prior *JavaScript* knowledge, while also trying out something new from this incredibly fast-evolving field.

2 User Documentation

2.1 Short description

A simple and intuitive cryptocurrency portfolio management application to measure the total portfolio value and assess the unrealized gains or losses for all open positions.

2.2 System requirements

Users are required to have a stable internet connection and a modern browser, no additional installation is needed. The full list of available browsers are listed on https://browserl.ist/?q=%3E0.2%25%2C+not+dead%2C+not+ie+%3C%3D+11%2C+not+op_mini+all, but it's recommended to use one of the more popular browsers:

- Firefox
- Chrome
- Safari
- Opera
- Edge

The application is also usable with any mobile device which has internet access and one of the listed browsers.

2.3 How to use the application

The web application is hosted on the <https://walfo.herokuapp.com> website.











2.3.1 Website structure

Coin lists page











The home page (Figure 1) lists the top 10 cryptocurrencies by market capitalization [11] and volume [12] in the last 24 hours. Besides the top lists, a searchable list shows the first 10 matching results' price and price change.

[LOGIN](#)
[REGISTER](#)

Search cryptocurrencies

Name	Price	Change (24h)
 Bitcoin	\$5,228.6200	0.0829%
 Ethereum	\$156.3900	1.1709%
 Litecoin	\$72.2500	0.0693%
 Dash	\$111.9800	0.5658%
 Monero	\$63.2600	0.7004%
 Nxt	\$0.0294	0.8977%
 Ethereum Classic	\$5.5000	4.1667%
 Dogecoin	\$0.0025	2.1277%
 ZCash	\$61.8900	3.0298%
 Bitshares	\$0.0533	2.4554%

Top cryptocurrencies by market cap

#	Name	Market Cap
1	 Bitcoin	\$92,379,451,619
2	 XRP	\$29,787,572,352
3	 Ethereum	\$16,550,684,272
4	 EOS	\$5,060,290,738
5	 Bitcoin Cash	\$4,645,989,388
6	 Litecoin	\$4,444,676,121
7	 Binance Coin	\$4,111,768,247
8	 Tether	\$2,846,831,291
9	 Stellar	\$1,896,455,537
10	 Cardano	\$1,779,374,851

Top cryptocurrencies by 24h volume











#	Name	Volume (24h)
1	 Bitcoin	\$2,162,915,763
2	 Ethereum	\$1,387,619,802
3	 EOS	\$856,866,343
4	 Litecoin	\$617,762,793
5	 XRP	\$358,145,438
6	 Bitcoin Cash	\$316,228,121
7	 TRON	\$178,245,515
8	 Binance Coin	\$144,239,767
9	 ZCash	\$131,183,244
10	 Dash	\$123,320,282

Figure 1: The coin lists page.

Detailed coin page

If a user would like to examine all of the attainable information about a coin, they can navigate to the detailed coin page by clicking on a cryptocurrency's name. Below the general info and market data blocks, two charts illustrate the price and volume change respectively during the last 7 days. An example of Bitcoin on 2019-04-27 is visible in Figure 2.



Figure 2: The detailed coin page.

Login page

The login page, shown in Figure 3, is accessible from the coin lists or the detailed coin page through the top navigation bar. It contains just the login form with an extra link to the register page. The coin lists page can be visited by clicking the "BROWSE COINS" navigation item in the header.

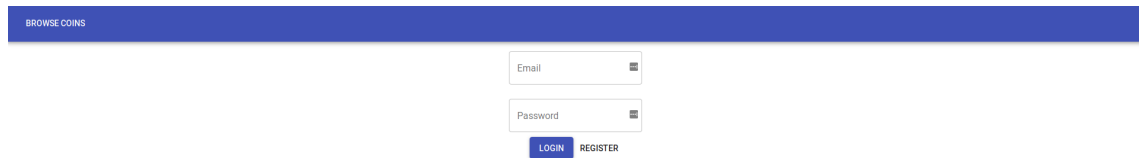
The login page features a dark blue header bar with the text "BROWSE COINS" in white. Below the header, there are two input fields: "Email" and "Password", each with a small icon on the right. At the bottom, there are two buttons: "LOGIN" in blue and "REGISTER" in white.

Figure 3: The login page.

Registration page

The registration page, just like the login page, can be accessed from the coin lists or the detailed coin page. It includes the registration form and the same header as on the login page. The registration page is displayed in Figure 4.

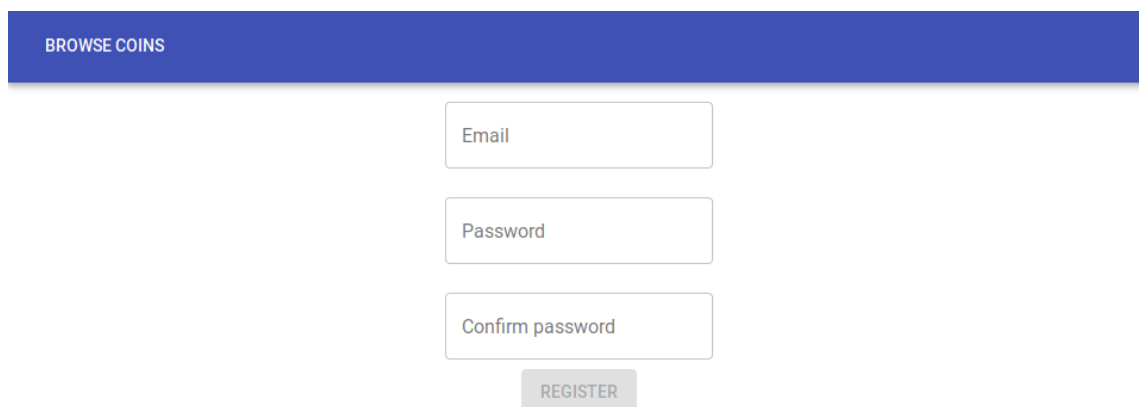
The registration page features a dark blue header bar with the text "BROWSE COINS" in white. Below the header, there are three input fields: "Email", "Password", and "Confirm password". At the bottom, there is a "REGISTER" button.

Figure 4: The registration page.

Dashboard page

The dashboard page, which is only available for authenticated users, can be seen in Figure 5. The header application bar contains the toolbar, where on the left side the currently selected portfolio's name ("First portfolio") is displayed together with an icon button. The user can use this button to open the side drawer (open position is exhibited in Figure 6) in which they can select one of their portfolios or start an entirely new one. On the right side, the "BROWSE COINS" button takes the user back to the coin lists page, "CHANGE PASSWORD" button opens the change password form in a dialog and the "LOGOUT" button can be used to log out from the current account. The layout of the selected portfolio consists of three main components. On the top of the page, two of these components are responsible for displaying information about the whole portfolio. The user can see the historical portfolio values in the chart, starting from the earliest transaction. To the left, general information about the current state of the portfolio is available, this is the portfolio report section. Under the portfolio report and the graph, the transactions

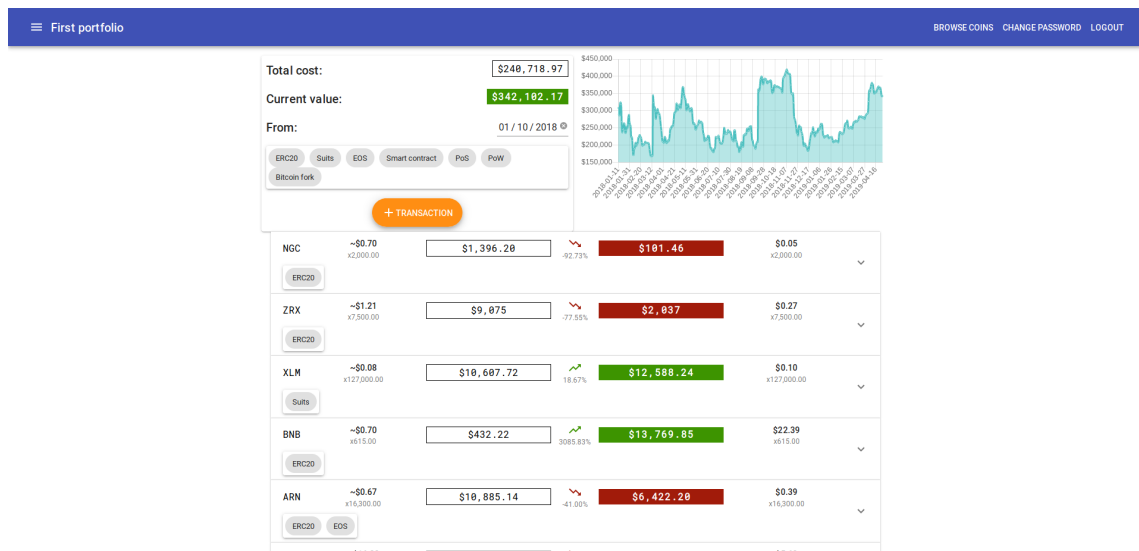


Figure 5: An example dashboard page.

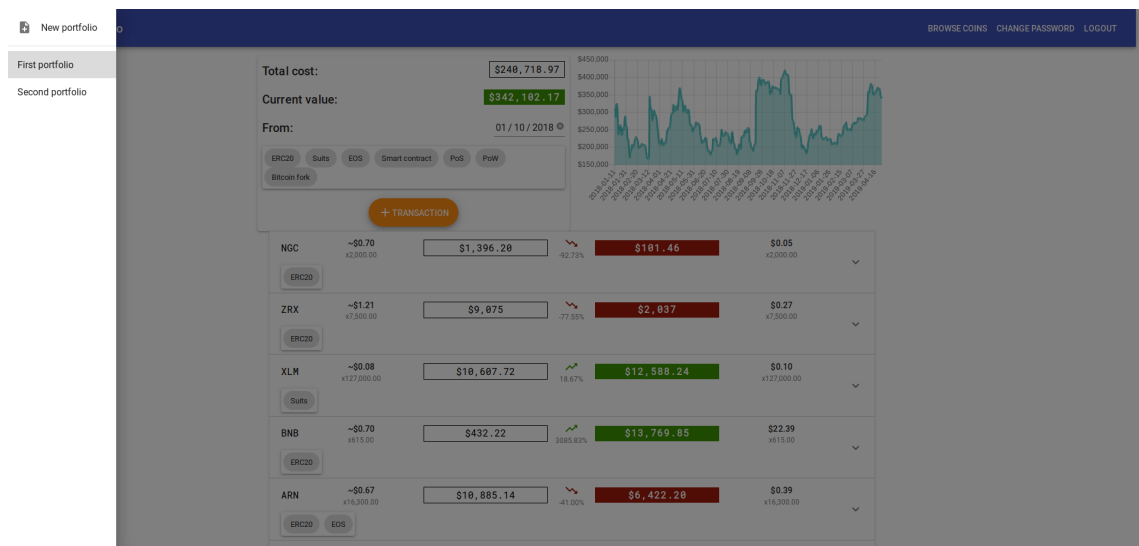


Figure 6: Dashboard page with open drawer.

are grouped into positions. Any of the position items can be expanded to examine every individual investment related to the same cryptocurrency, seen in Figure 7.

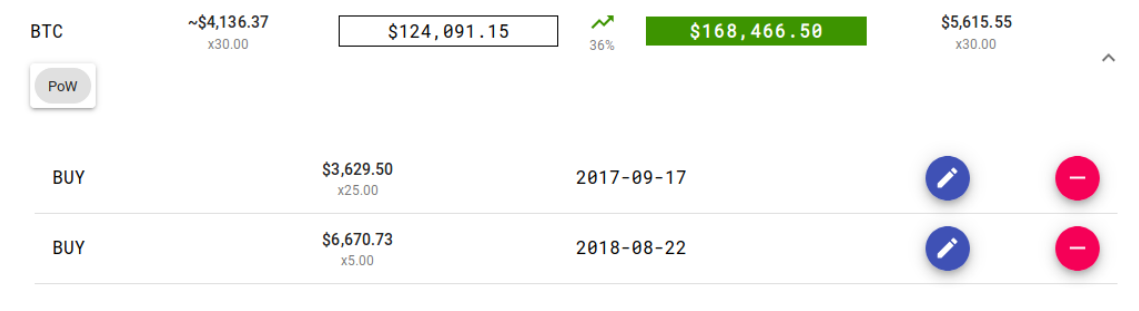


Figure 7: Expanded position.

Portfolio report

The interface displays the following information:

- Total cost:** \$252,181.31
- Current value:** \$369,679.69
- From:** 01 / 10 / 2017
- Filters:** ERC20, Pre-mined, Suits, EOS, Smart contract, PoS, PoW, Bitcoin fork
- Action:** + TRANSACTION

Figure 8: Portfolio report with general portfolio information and input fields.

The "Total cost" is the sum of all of the positions' cost until the current date. "Current value" is determined from acquiring market data for each of the cryptocurrencies held in the portfolio combining it with the input transactions to measure the present market value. The "From" field, by default, is set to the date of the earliest transaction but it can be changed to showcase the portfolio's performance within a shorter time span. All of the tags are merged into a box under the "From" input, and any of the tag chips are clickable to filter the portfolio for positions which include that particular tag. The last item, an action button with the text "+ TRANSACTION" represents the most common action on the screen, adding a new transaction to the portfolio. A portfolio report example is shown in Figure 8.

Historical graph

The chart (Figure 9) shows the total portfolio value between the specified date and the current date. The steps are automatically adjusted in a way that the graph will fit into the available space.

Positions

Every transaction for the portfolio is grouped into positions. An example position list is shown in Figure 10. A position item displays the aggregated data from the transactions and the change in value relative to the investment cost. The first component of a position item is the ticker symbol [13] of the cryptocurrency. The next component contains the average cost of an investment and the holdings for



Figure 9: A graph for historical portfolio value.

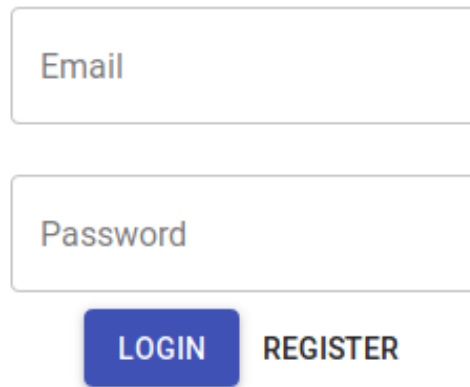
NGC	~\$0.70 x2000.00	\$1,396.20	~91%	\$120.50	\$0.06 x2000.00	▼
ERC20						
ZRX	~\$1.21 x7500.00	\$9,075.00	~75%	\$2,226.00	\$0.30 x7500.00	▼
ERC20						
WAVES	~\$7.06 x1623.56	\$11,462.33	~63%	\$4,286.20	\$2.64 x1623.56	▼
Pre-mined						
XLM	~\$0.08 x127000.00	\$10,607.72	~29%	\$13,665.20	\$0.11 x127000.00	▼
Suits						
BNB	~\$0.70 x615.00	\$432.22	~3104%	\$13,849.80	\$22.52 x615.00	▼
ERC20						

Figure 10: Position list.

the position, next to them in the bordered box, the product of these two numbers shows the total investment cost. Between the investment cost and current market value, a green uptrend or red downtrend arrow with the average profit or loss ratio under it indicates the performance of the position. The current market value is calculated from the product of the cryptocurrencies' market price and the number of holdings. This value has a red background if it has unrealized losses otherwise, it has a green background. If a position item is expanded, shown in Figure 7, the detailed transaction information, with transaction type (Buy/Sell), the price, the amount and the date is available for the user; here they can also interact with the two action buttons to edit or delete the associated transaction.

2.3.2 Authentication

To access the management application it's required to have a user account. After opening the website, a user has only two options available for them – as shown in Figure 11 – to login with an already existing account or navigate to the register page and create a new one.

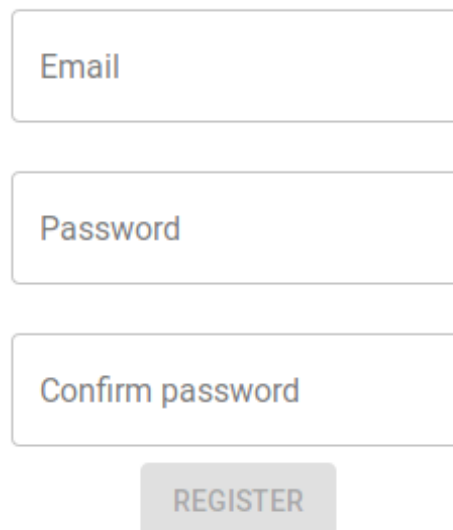


The login form consists of two input fields stacked vertically. The top field is labeled "Email" and the bottom field is labeled "Password". Below these fields are two buttons: a blue "LOGIN" button and a grey "REGISTER" button.

Figure 11: The login form.

Registration

For successful registration, the user has to enter a valid email and two matching passwords. The "REGISTER" button is only available if all of the input data meet



The registration form consists of three input fields stacked vertically. The top field is labeled "Email", the middle field is labeled "Password", and the bottom field is labeled "Confirm password". Below these fields is a single grey "REGISTER" button.

Figure 12: The register form.

the requirements. Having two accounts for one email is also prohibited: after clicking the "REGISTER" button the server will verify that the email hasn't yet been taken. Figure 13 shows the possible error states. If the registration is successful,

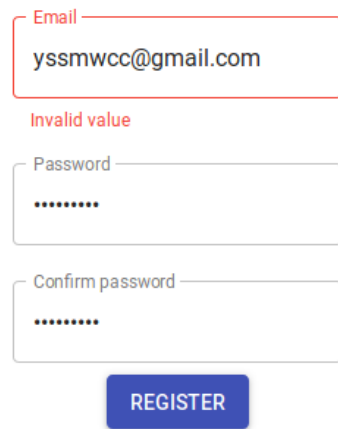
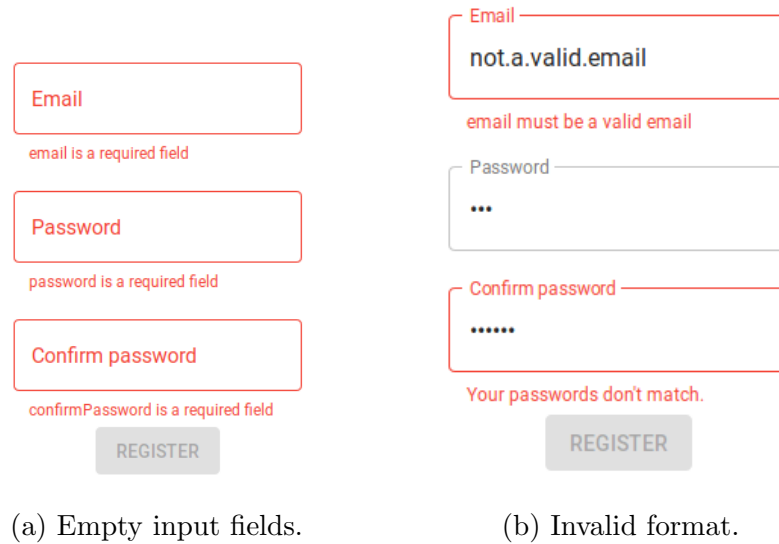


Figure 13: Error messages for various failed registration attempts.

the dashboard page will be shown, with the user already logged in with their new account.

Login

The user can log in with their previously registered account by providing their email and password. If the authentication fails, the application will alert them that either their email or password is wrong. Just like after the successful registration, after logging in the user is ready to use the portfolio management system.

Session storage

The client application stores a user-specific token in the browser after authentication, which can instantly log in the user when they visit the website again, and they'll be redirected to their dashboard page.

Email

i.havent@registered.yet

Wrong email

Password

.....

or password!

LOGIN REGISTER

Figure 14: Failed login.

Logout

The user can log out from the application to return to the login or register page. Logging out also deletes the stored session token from the browser and they won't be automatically authenticated on their next visit.

Changing password

On the dashboard page, the password change dialog can be opened with the "CHANGE PASSWORD" toolbar button. To successfully change their password, the user only needs to input two matching password, like in the registration form.

2.3.3 Creating a new portfolio

A new portfolio can be created after navigating to the "New portfolio" page, selected in the drawer. The user can easily initialize a new portfolio with adding a new valid transaction. The only restriction for the portfolio input data is that the supplied portfolio name should be unique. The "+ CREATE TRANSACTION" button will remain disabled until the form has an error. Examples for the various input and error states of this form can be seen in Figure 15. The detailed description of how to add transactions is in the 2.3.4 section.

2.3.4 Creating a new transaction

Clicking the "+ TRANSACTION" action button on the portfolio page will open a dialog with the new transaction form. The "Coin" field is a typeahead select input field, which means typing in this field will search for the most similar cryptocurrencies available for real-time market data, then the preferred cryptocurrency can be selected by clicking on it. The expanded state of this field is

Create a transaction for your new portfolio!

Portfolio

Coin

Type

Buy

Amount

Price

Date

04 / 24 / 2019

New tag

+ CREATE TRANSACTION

Create a transaction for your new portfolio!

Third portfolio

Bitcoin (BTC)

Type

Buy

1

5000

Date

04 / 24 / 2019

with tags

New tag

+ CREATE TRANSACTION

(a) Form in its initial state.

(b) A valid form state.

Create a transaction for your new portfolio!

Portfolio

portfolio is a required field

Coin

Please provide a valid coin!

Type

Buy

Amount

Price

amount is a required field

price is a required field

Date

04 / 24 / 2019

New tag

+ CREATE TRANSACTION

(c) Form with every mandatory field left empty.

Portfolio

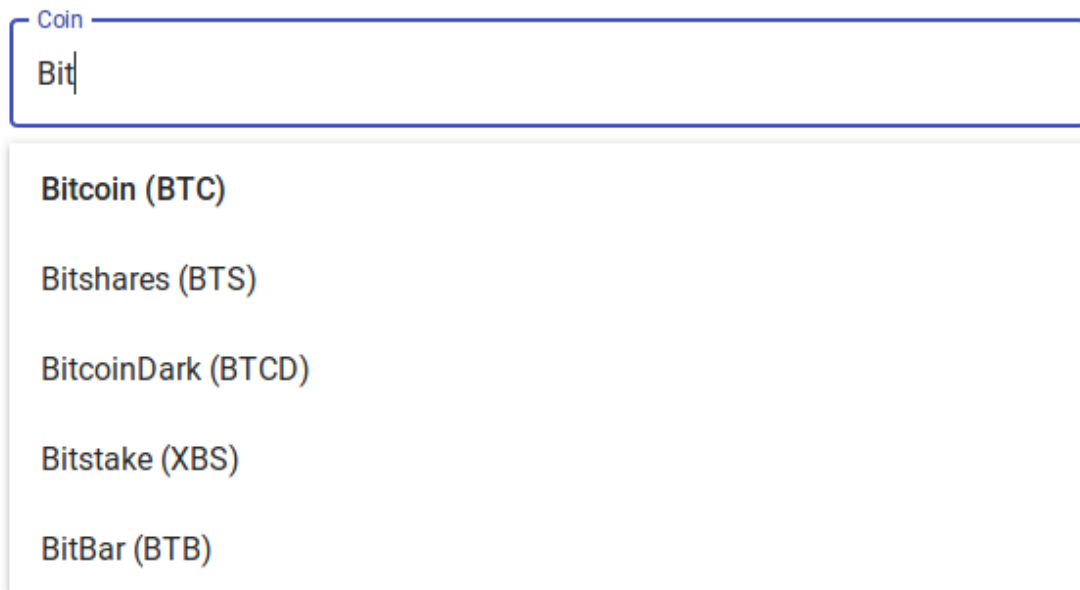
First portfolio

portfolio must not be one of the following values: First portfolio, Second portfolio

(d) Error message for an existing portfolio.

Figure 15: Possible state for the new portfolio form.

demonstrated in Figure 16. The "Amount" and "Price" fields are basic mandatory number inputs which can't have negative values. The "Type" is a simple select input field where the user can choose between "Buy" or "Sell" options. The user is allowed to sell more units than the amount they're currently holding. The date of the transaction is a mandatory date input which is set by default to the current date, and the input date can't be later than this date. The tags input is an optional field, which will be prefilled with the tags related to the selected cryptocurrency if its position has any. A new tag can be added to the list by pressing the TAB button, then the text input will reset and another tag can be created.



Coin

Bit

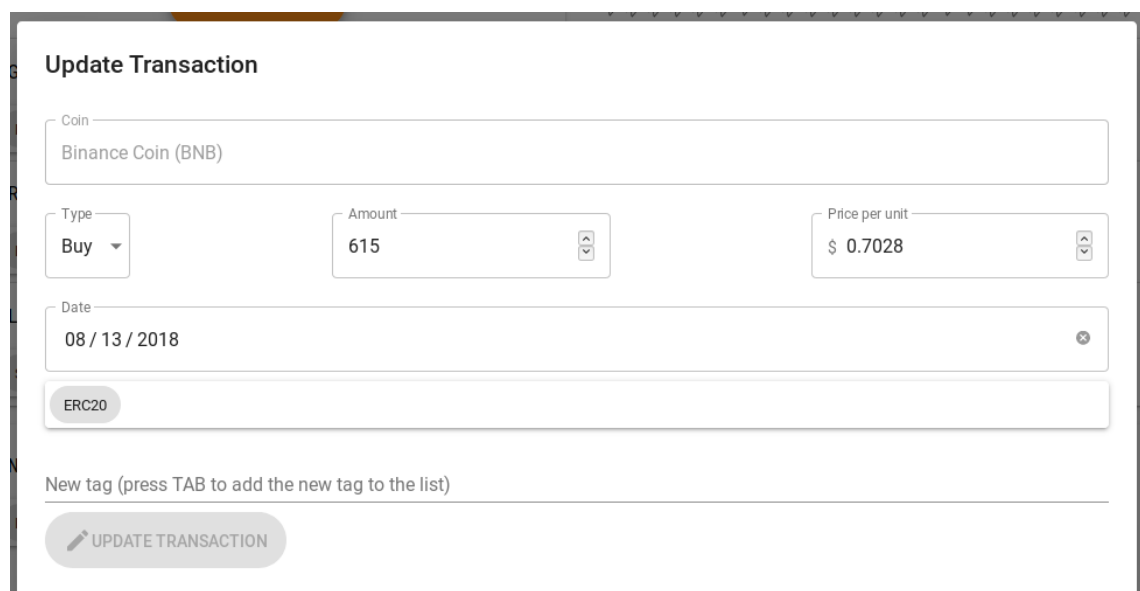
- Bitcoin (BTC)
- Bitshares (BTS)
- BitcoinDark (BTCD)
- Bitstake (XBS)
- BitBar (BTB)

This image shows a dropdown menu for selecting a coin. The search bar contains the text "Bit". The dropdown list displays five options: Bitcoin (BTC), Bitshares (BTS), BitcoinDark (BTCD), Bitstake (XBS), and BitBar (BTB).

Figure 16: Expanded coin select.

2.3.5 Updating a transaction

A transaction can be updated by expanding the related position item and clicking the edit action button, which will open a dialog with the same transaction form when creating a new transaction. Every field validation is similar to the transaction creation form, except the symbol can't be changed; to change the symbol the user needs to delete the wrong transaction and recreate the desired one.



Update Transaction

Coin
Binance Coin (BNB)

Type
Buy

Amount
615

Price per unit
\$ 0.7028

Date
08 / 13 / 2018

ERC20

New tag (press TAB to add the new tag to the list)

UPDATE TRANSACTION

This image shows the "Update Transaction" dialog form. It contains several input fields: "Coin" (Binance Coin (BNB)), "Type" (Buy), "Amount" (615), "Price per unit" (\$ 0.7028), and "Date" (08 / 13 / 2018). There is also a section for "ERC20" and a "New tag" field. At the bottom, there is a button labeled "UPDATE TRANSACTION".

Figure 17: Update transaction dialog.

2.3.6 Deleting a transaction

The delete action button is next to the edit button inside a transaction item; clicking the "YES" button in the confirmation dialog will permanently delete the transaction.

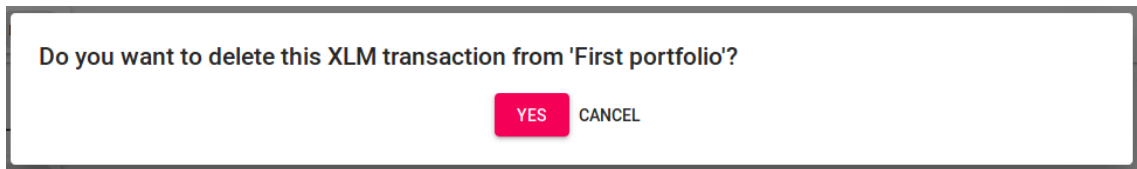


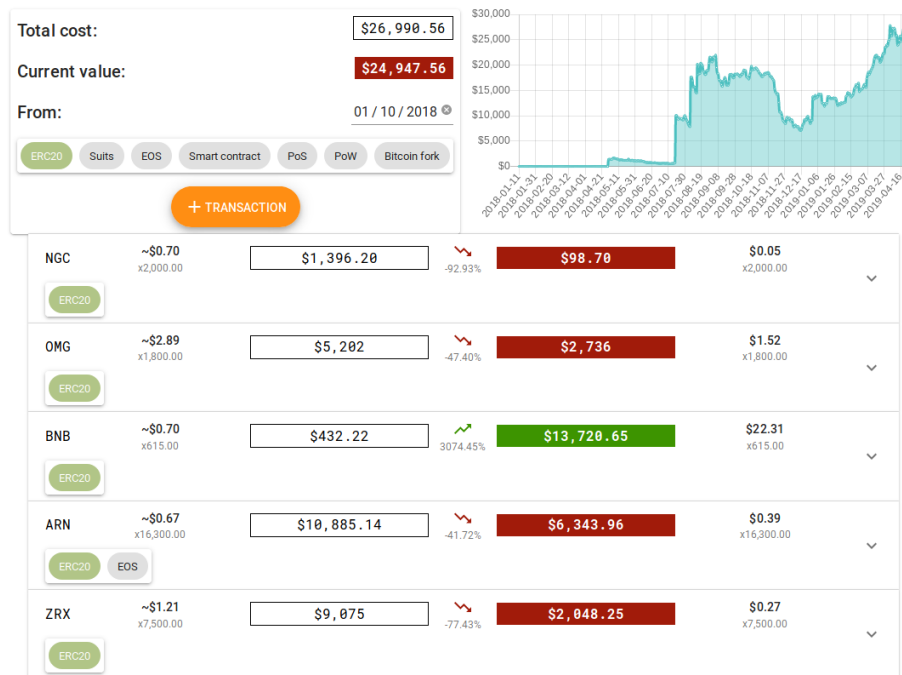
Figure 18: Delete transaction dialog.

2.3.7 Filtering the portfolio

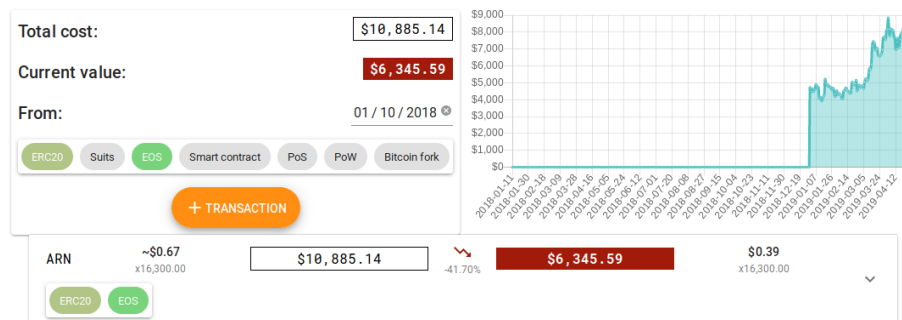
Selecting tags by clicking on them in the tags selector component will remove positions from the aggregation which doesn't have all of the selected tags. The total cost and current value of the portfolio will be adjusted to represent the change in the available positions. The historical graph will also only include matching positions. The different query states for an example portfolio is presented in Figure 19.

2.3.8 Changing the portfolio graph start date

Changing the "From" date field in the portfolio report component will only change the starting date of the historical graph and won't change the number of displayed transactions. Currently, changing the ending date from the current date isn't possible, the central aim of the application is to display information about the current state of the open positions.



(a) With one tag selected.



(b) With multiple tags selected.

Figure 19: Same portfolio with different selected tags.

3 Developer documentation

3.1 High-level design

Two types of users are differentiated in the application, most of the features are available only after authentication, but basic cryptocurrency market data is also available for unregistered users. A guest can interact with the application in the following ways:

- View the cryptocurrency top lists
- Search for any coin to get price data
- Access detailed information about a cryptocurrency
- Register
- Log in

Authenticated users can do everything that guests can do except "Register" and "Log in" and additionally, they can:

- Create any number of portfolios
- Create, modify or delete transactions related to their portfolio
- Have access to aggregated position and portfolio information
- Filter their open positions by tags
- Examine their portfolio's market values for each day, starting from their first transaction
- Change their account's password
- Log out

Potential user interactions are presented in Figure 20.

3.2 Architecture design

The complete application consists of four subsystems, the database, the backend service, the cryptocurrency market data API and the frontend client application. The architecture diagram in Figure 21 shows which parts communicate with each other.

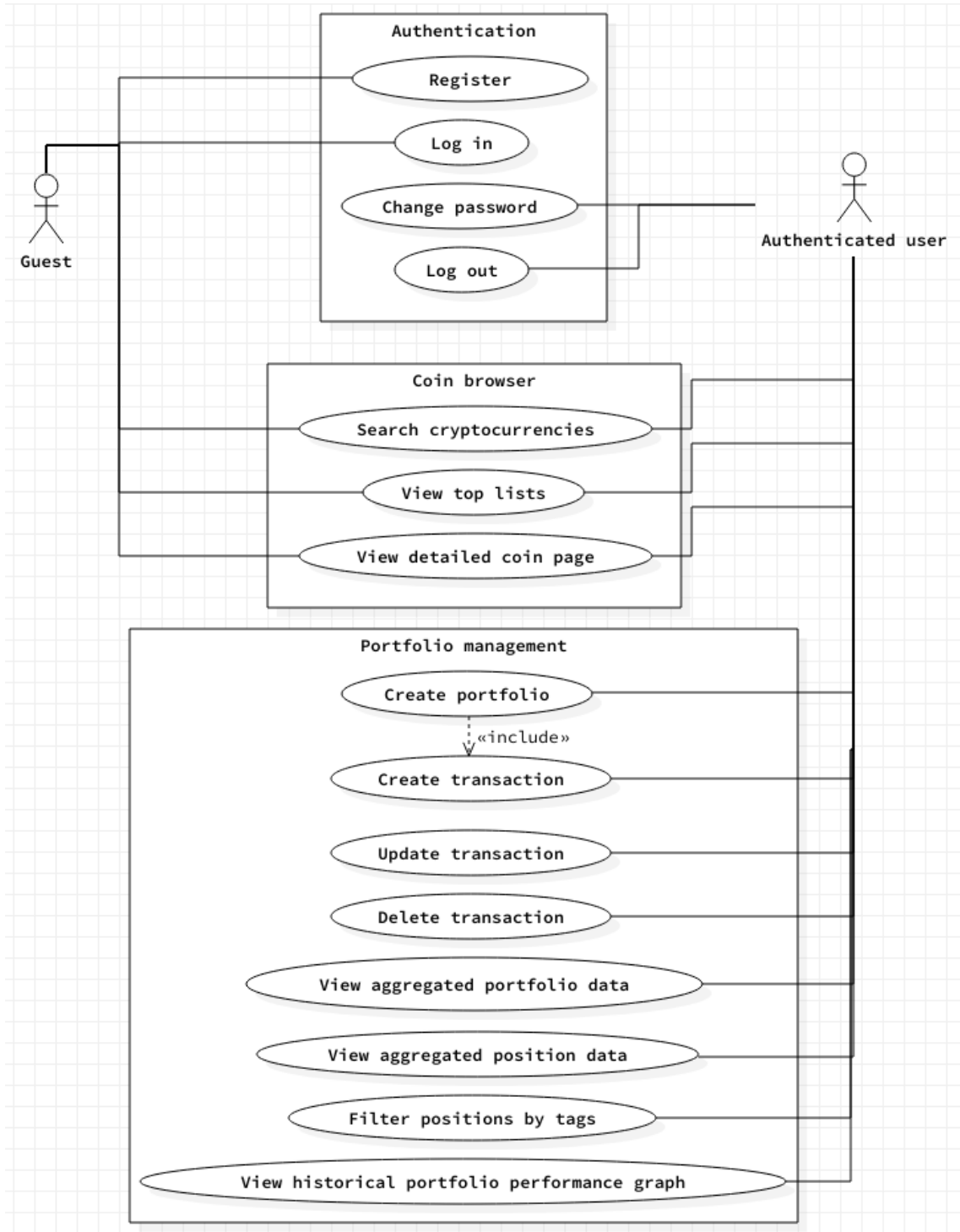


Figure 20: The use case UML diagram.

3.2.1 Database

A *MongoDB* [14] database is responsible for persisting and retrieving user and transaction information. *MongoDB* is a document-oriented *NoSQL* [15] database program, which stores data in *JSON-like* documents instead of rows and columns in a series of tables modeled in relational databases [16]. *NoSQL* is built to support

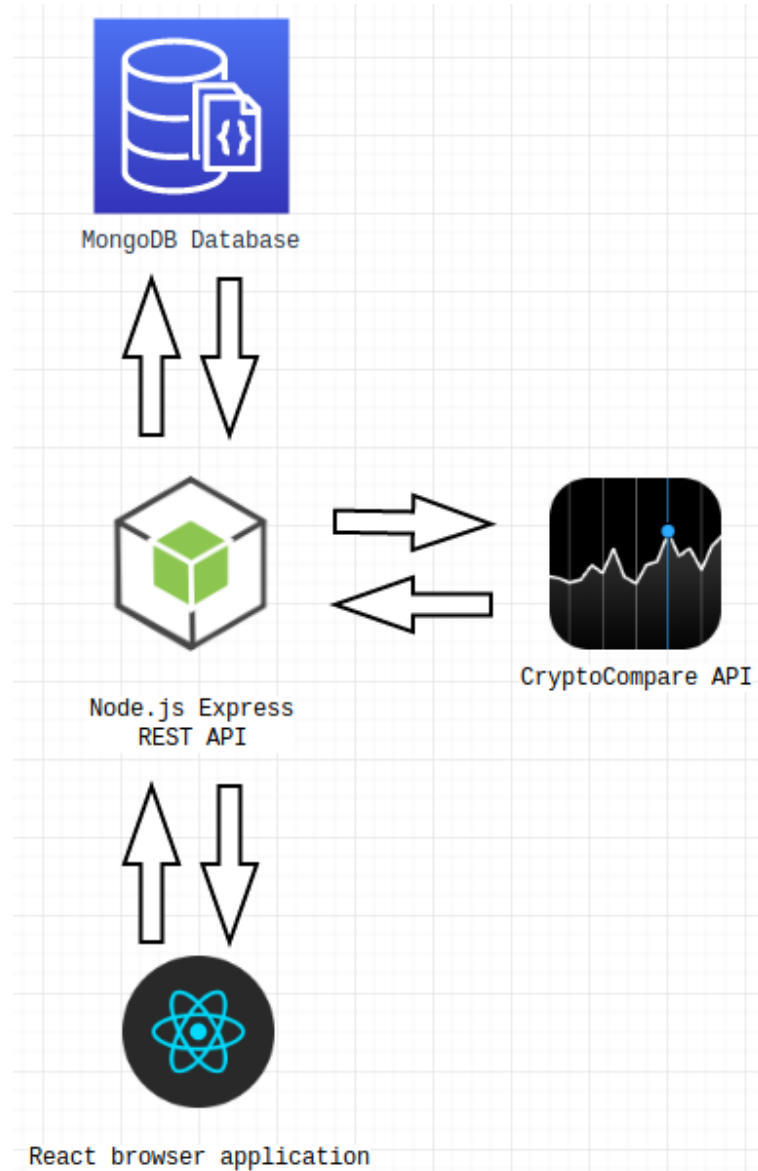


Figure 21: The architecture diagram.

flexible data models which help in scaling the application easier and faster. The downside is that it may not guarantee ACID (Atomicity, Consistency, Isolation, Durability) [17] properties for database transactions but still offers reasonable fault tolerance. The document data model is also useful in mapping a set of fields to programming language data types, which reduces the time needed for database schema design and development.

3.2.2 Backend server

The backend server implements the Representational state transfer (REST) [18] style of software architecture and is developed in the *Node.js Express* framework. Similar to the advantages of *NoSQL*, the Node asynchronous event-driven *JavaScript* runtime environment is designed to increase scalability. In contrast to more common

concurrency thread-based environments, *Node.js* frees the developers of potential blocking behavior, utilizing asynchronous events ensure the process never blocks. *Express* is a minimalist web framework for *Node.js*, which provides a thin layer of fundamental web application features but it's easily extensible with additional packages. The RESTful web service provides uniform interoperability between systems by allowing the requesting systems to access and manipulate their web resources with a predefined set of stateless operations. The *Express* server accepts request payloads formatted in *JSON* [19] and will answer with similar response payloads. The RESTful nature of the system makes the web service reliable, and because web resources can be updated without affecting the whole service, the server becomes easily scalable.

3.2.3 Market data API

The *CryptoCompare API* gives access to historical and real-time cryptocurrency data. It has information about almost 6000 cryptocurrencies on more than 200 exchanges and has a limit on 100,000 API calls/month for non-commercial projects. The API data service is built on REST principles and is accessible with API key authentication. This REST style of software architecture makes the integration with the backend server simple.

3.2.4 Frontend client application

The frontend application was written in the *React JavaScript* library. *React* introduced the concept of component-based architecture [20] which emphasizes encapsulating individual pieces of a large user interface into composable components [21] with their own state. *React* makes it simple to build interactive UIs and it uses an efficient algorithm to detect changes to the component tree and render only the right components instead of rerendering the entire page.

3.3 Development environment

The source code is hosted on the *GitHub* [22] software development platform, and is available at the following link: <https://github.com/bkiac/walfo>. The following dependencies are required before the development can start:

- **Node.js** and **Yarn** package manager
- Access to a **MongoDB** database
- **CryptoCompare API** key
- A **JWT secret** string for *Passport.js* token signature generation

If all the requirements are met, the *JavaScript* dependencies can be downloaded using the `yarn install` command, this will first download the *Lerna* library, which will bootstrap the frontend and backend packages.

The following variables must be set in the two `.env` files to prevent compilation errors:

- `backend/.env`:
 - `HOST=<desired server address>`
 - `PORT=<desired server port>`
 - `DATABASE=<connection string for a MongoDB database>`
 - `API_URL=https://min-api.cryptocompare.com/data/`
 - `API_KEY=<CryptoCompare API key>`
 - `JWT_SECRET=<JWT secret to sign tokens>`
- `frontend/.env`:
 - `REACT_APP_API_URL=<the backend REST API URL>`

Development servers are available for both frontend and backend applications, which can be started with running the `yarn watch` command in their respective directories. It's important to familiarize ourselves with the directory structure for effective development. This is how the source files of this application is organized:

- `package.json`: Lists the *Lerna* dependency and scripts for the root project
- `lerna.json`: Configuration file for *Lerna* package manager
- `travis.yml`: Configuration file for *Travis* continuous integration service
- `node_modules/`: Contains the shared dependencies of the backend and frontend packages
- `backend/`: Contains the backend package
 - `package.json`: Lists the backend dependencies and scripts
 - `.env`: Lists environment variables, e.g.: the API key, database connection string or the *JWT* secret
 - `jest.config.js`: Configuration file for the *Jest* testing framework
 - `node_modules/`: Contains dependencies for the backend server
 - `test/`: Contains configuration and helper functions used during tests
 - `src/`: Contains the source code of the server

- **frontend/**: Contains the frontend package
 - **package.json**: Lists the frontend dependencies and scripts
 - **.env**: Lists environment variables, e.g.: the API URL of the backend REST API
 - **.browserslistrc**: Lists the target browsers, this list is used by the build tools
 - **node_modules/**: Contains dependencies for the frontend application
 - **public/**: Contains the root **index.html** file of the application
 - **src/**: Contains the source code of the frontend application
 - * **components/{containers,pages,providers,views}/**
 <Component>/
 - **index.jsx**: *JavaScript* source code of the **Component**
 - **style.module.scss**: Style sheet module for the **Component** written in *SCSS* [23]

3.4 Common technologies

Both the backend and frontend systems are implemented using the *JavaScript* language, and the core functionality of the language is extended with packages downloaded from a global registry. This is a list of packages that were used during the development of the backend server and the frontend application:

- **axios**: Promise based HTTP client for the browser and *Node.js*; simplifies sending requests and receiving responses
- **dotenv**: Module to load environment variables from a **.env** file into **process.env**
- **ESLint**: Linting utility for *JavaScript* and *JSX*; analyzes the source code to flag errors, bugs and potential optimizations
- **Lerna**: Tool for managing *JavaScript* projects with multiple packages; makes it easy to separate the codebase into packages while keeping it organized into a single repository
- **Lodash**: *JavaScript* utility library
- **Prettier**: An opinionated code formatter; enforces consistent coding style
- **Yarn**: Dependency manager for *JavaScript* packages

3.5 MongoDB database

The document model supports a flexible data structure, but it's still important to design the core properties of a data record and the relationship between the collections. The database structure is shown in Figure 22.

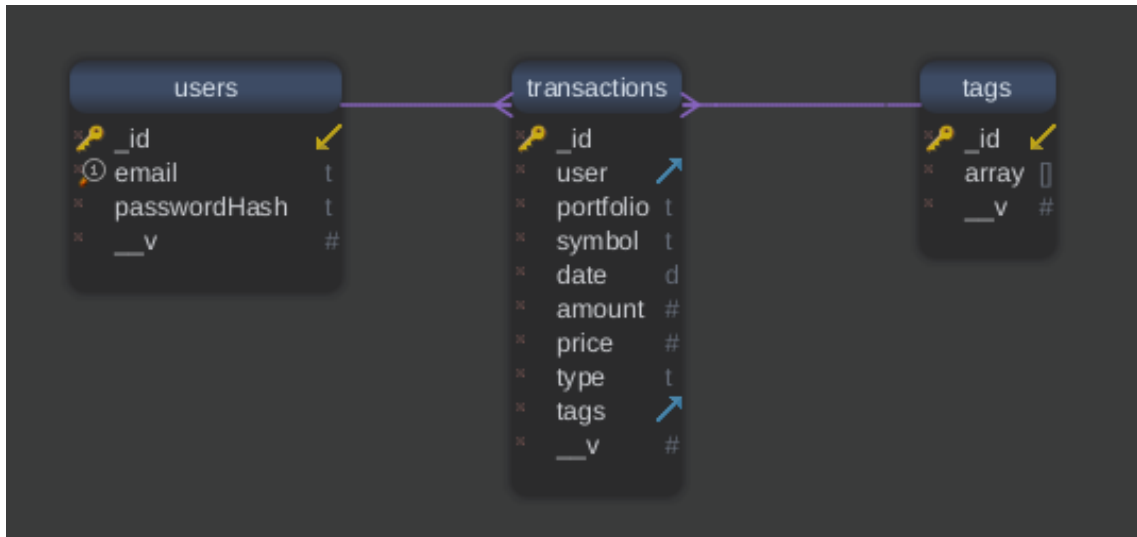


Figure 22: The database structure.

3.5.1 Users

The `users` collection stores the account information of a user. The `email` field is unique for every account and to obscure the registered user's password, only a hash derived from the actual password is stored in a document.

User schema		
<i>Field</i>	<i>Data type</i>	<i>Description</i>
<code>_id</code>	<code>ObjectId</code>	Primary key
<code>email</code>	<code>String</code>	Unique field that stores the user's email
<code>passwordHash</code>	<code>String</code>	A hash derived from the original password

3.5.2 Transactions

A transaction document is described in the table below. Portfolios and positions don't exist on the schema level, they are always aggregated from the transactions. Transactions which have the same **user** value are owned by the same user. Transactions owned by the same user can be collected into a portfolio if they have the same value in the **portfolio** field. A position can be calculated from transactions which have the same **user**, **portfolio** and **symbol** values. The **tags** field has a reference to a separate tags document to reduce the complexity of managing a different instance of tags array for every transaction. This way, changing the tags field for a position will update only one tags document instead of an array type field for all associated transaction.

Transaction schema		
<i>Field</i>	<i>Data type</i>	<i>Description</i>
_id	ObjectId	Primary key
user	ObjectId	ID reference for the owner's user document
portfolio	String	The name of a portfolio
symbol	String	The ticker symbol of a cryptocurrency
date	Date	The date of the transaction
amount	Number	The number of units of a cryptocurrency
price	Number	The price per unit of a cryptocurrency
type	String Enum	The value can be "BUY" or "SELL"
tags	ObjectId	ID reference for the position's tags document

3.5.3 Tags

A tags document contains a single **array** field which is a set of strings. *MongoDB* database doesn't guarantee the uniqueness of the values inside the array, therefore it's ensured on the application level while creating or updating a document.

Tags schema		
<i>Field</i>	<i>Data type</i>	<i>Description</i>
_id	ObjectId	Primary key
array	String[]	An array of unique strings

3.6 Express REST API service

The RESTful server is the central point between the subsystems, it's responsible for maintaining communication between the database, the market data API and the client application. The control flow is illustrated in Figure 23. The client will send a request to one of the predefined Uniform Resource Identifiers (URIs) to address a

resource. The request will be passed to the middlewares and the *body-parser* library will parse its body, making it available under the `req.body` property, then the parsed request will be passed on to the route handlers. The router defines which controller methods will be executed with the incoming request. The controllers can either terminate the flow by sending a response to the client or after successful execution they forward the request to the next controller defined in the router. The controllers can also initiate a request to the *CryptoCompare API*, query the database for records or modify the request. After all the necessary controller methods were completed the client will receive the server's response.

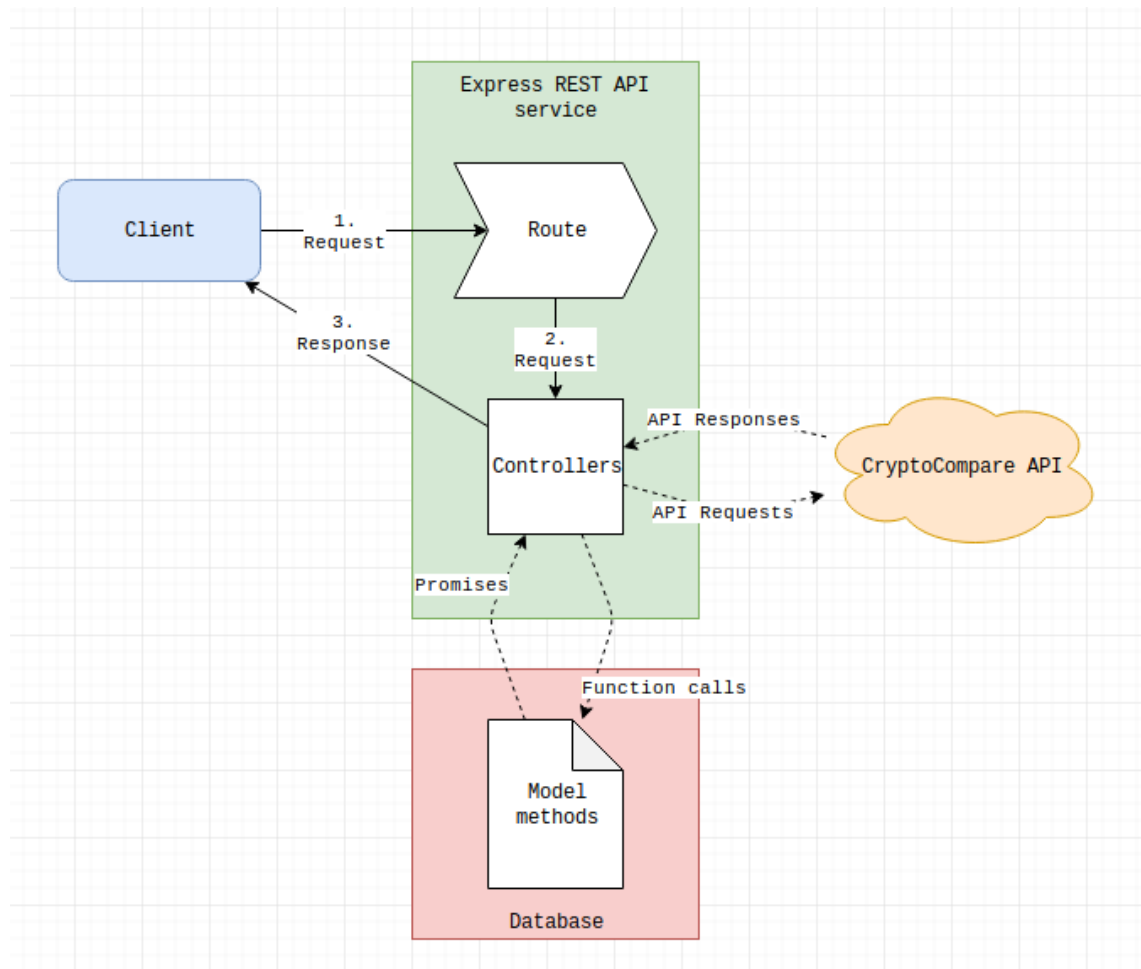


Figure 23: Control flow of the application. The dashed lines show potential data queries.

This is the list of important packages used for the development of backend server:

- **bcrypt**: Library to hash passwords
- **body-parser**: *Node.js* body parsing middleware
- **Moment.js**: Lightweight date library for parsing, validating, manipulating, and formatting dates

- **mongoose**: *MongoDB* object modeling tool designed to work in an asynchronous environment
- **node-fetch**: Light-weight module that brings `window.fetch` to *Node.js*
- **Passport.js**: Authentication library for *Node.js*
- **validator**: Library of string validators and sanitizers

3.6.1 Models

The *mongoose* Object Data Modeling (ODM) library for *MongoDB* manages relationships between data, provides schema validation, translates *JavaScript* objects between the representation of these objects in the database. Models are higher-order constructors that accept a schema and create an instance of a document. Additional methods can be defined on these models to create complex queries, parse data in hooks before database actions, add helper functions or inject virtual fields.

User

`backend/src/models/User.js`

The user model contains the user schema and methods for safe password storage and validation. A user document can be created by supplying email and password values for the model. The lowercase email will be saved after all whitespaces were trimmed, this process ensures a clean user collection and only those user documents will be persisted which have a unique email value. When a user model is created with a password, that input password won't be saved and a hashed password – computed with the *bcrypt* [24] library – will be stored instead. The comparison between the input password and the stored password hash is also implemented with the *bcrypt* library.

Transaction

`backend/src/models/Transaction.js`

The transaction model is the most complex model of the three. It's responsible for aggregating multiple simple transaction documents into meaningful position information, this is achieved with custom static methods defined on the model. The position aggregation pipeline is built up from multiple steps. First, transactions which match the input user ID and portfolio name are selected from the database, then *MongoDB* has to look up and populate the tags reference ID field of each transaction. If the query input had a tags argument, the transactions will be further reduced by those transactions whose populated tags array is not a subset of the input

array. Now the transactions are grouped into positions by ticker symbol and the position's number of bought coins, number of holdings and total cost are calculated.

$$T := \{ \text{Set of a position's transactions} \}$$

$$B := \text{Buy Transactions} = \{t \in T \mid t.type = \text{"BUY"}\}$$

$$S := \text{Sell Transactions} = \{t \in T \mid t.type = \text{"SELL"}\}$$

$$\text{number of bought coins} = \sum_{t \in B} t.amount$$

$$\text{holdings} = (\text{number of bought coins}) - \sum_{t \in S} t.amount$$

$$\text{total cost} = \sum_{t \in B} t.amount \cdot t.price$$

The next step is to calculate the cost of a position. The cost will show how much money was invested on average in the current open position. If there are buy transactions but the holdings equal zero then the cost will also equal zero, this means the position can be considered closed at the moment because there are no unsold units of cryptocurrency.

$$\text{cost} = \begin{cases} \text{holdings} \cdot \frac{\text{total cost}}{\text{number of bought coins}}, & \text{if number of bought coins} > 0 \\ 0, & \text{if number of bought coins} = 0 \end{cases}$$

By executing this position aggregation method for each day in an interval the historical cost of the positions can be determined.

Tags

`backend/src/models/Tags.js`

The tags model is a simple *mongoose* model with one additional helper method. The tags schema requires that its array's items should be unique and although *MongoDB* doesn't guarantee on the database level that every item in the array is different, pre hooks can be added to the model to filter out repeating tag strings before the document is persisted.

3.6.2 CryptoCompare API

`backend/src/api/cryptocompare.js`

The existing *cryptocompare* [25] library which maps a function to most of the URI endpoints on the *CryptoCompare REST API* wasn't satisfactory, therefore some of the functions were reimplemented and new features were built to make it easier to

send requests to the API. One of the major problems with the library is that it reassigns the function parameters which lead to misleading and confusing behavior because, for example, the second function call with the same object always failed due to the original date object was modified in a previous function execution. The library was also extended with the missing requests for the top lists endpoints.

3.6.3 Controllers

Most of the server logic is implemented in the controller layer. A controller will receive the forwarded HTTP request from their associated route, then they can communicate with the database to read or write data, or send additional HTTP requests to the external market data API. The last step of a controller's logic is to either send a response to the client, or if there are more controllers related to the same route after the current one, they will pass on the request to the next controller. All of the controllers are designed to be modular and contain related logic only to a particular slice of the server to increase scalability and separation of concerns. The controller implementations can be found in the `backend/src/controllers/` directory.

Authentication controller

`backend/src/controllers/authController.js`

This controller is responsible for features such as login, registration, password change and protection. Both login and protection methods use the *Passport.js* library for authentication. A successful login call execution will result in sending a signed *JWT* [26] token with logged in user's ID as the response payload. The protect method will secure its related route thus only a logged in user can access it by enforcing that a signed *JWT* token is applied in the request header's **Authorization** property. If a valid token is present then the user's ID will be attached to the request, but if the token is missing or expired the client will receive a response with 401 status code which signals that the request failed at authorization.

Validation controller

`backend/src/controllers/validationController.js`

The validation controller contains the validators for a URI, applying these validators in a route will guarantee that the rest of the controllers will receive a valid request object. The potential errors are collected on the request object which will be passed on to the main validate method, which can send an error response with 422 status code to signal that the received request is invalid or if the request is valid, it will be forwarded to the next controller.

Tags controller

`backend/src/controllers/tagsController.js`

The tags controller after receiving an already valid request object, it will execute the necessary create, update or delete database operation and replace the original string tags array with a document.

Transaction controller

`backend/src/controllers/transactionController.js`

If a request reaches one of the transaction controller's methods, all of the required information for a transaction operation will be available on the request and the change in the transaction collection will be persisted.

Portfolio controller

`backend/src/controllers/portfolioController.js`

The portfolio controller merges the position information received from the transaction model with the market data response from the *CryptoCompare API*. To calculate the current portfolio data, the controller will query the database for the user's requested portfolio's positions, then the market prices for each position will be collected. After all the promises have been resolved, the position's current value and average profit margin will be calculated. The portfolio's cost and value is the sum of its positions cost and value fields.

$$\text{current value} = \text{holdings} \cdot \text{market price}$$

$$\text{average profit margin} = \begin{cases} -1 + \frac{\text{current value}}{\text{cost}}, & \text{if } \text{cost} > 0 \\ -1, & \text{if } \text{cost} = 0 \end{cases}$$

Coins controller

`backend/src/controllers/coinsController.js`

The coins controller is a proxy for the *CryptoCompare API*, the requested market data is returned to the client after it has been parsed.

3.6.4 Routes

The routes determine how the server responds to a client request to a particular endpoint and a specific HTTP request method. Every route can have one or more controller functions, which are executed when the route is matched. A route can be protected with the auth controller's protect handler, then only requests with a valid

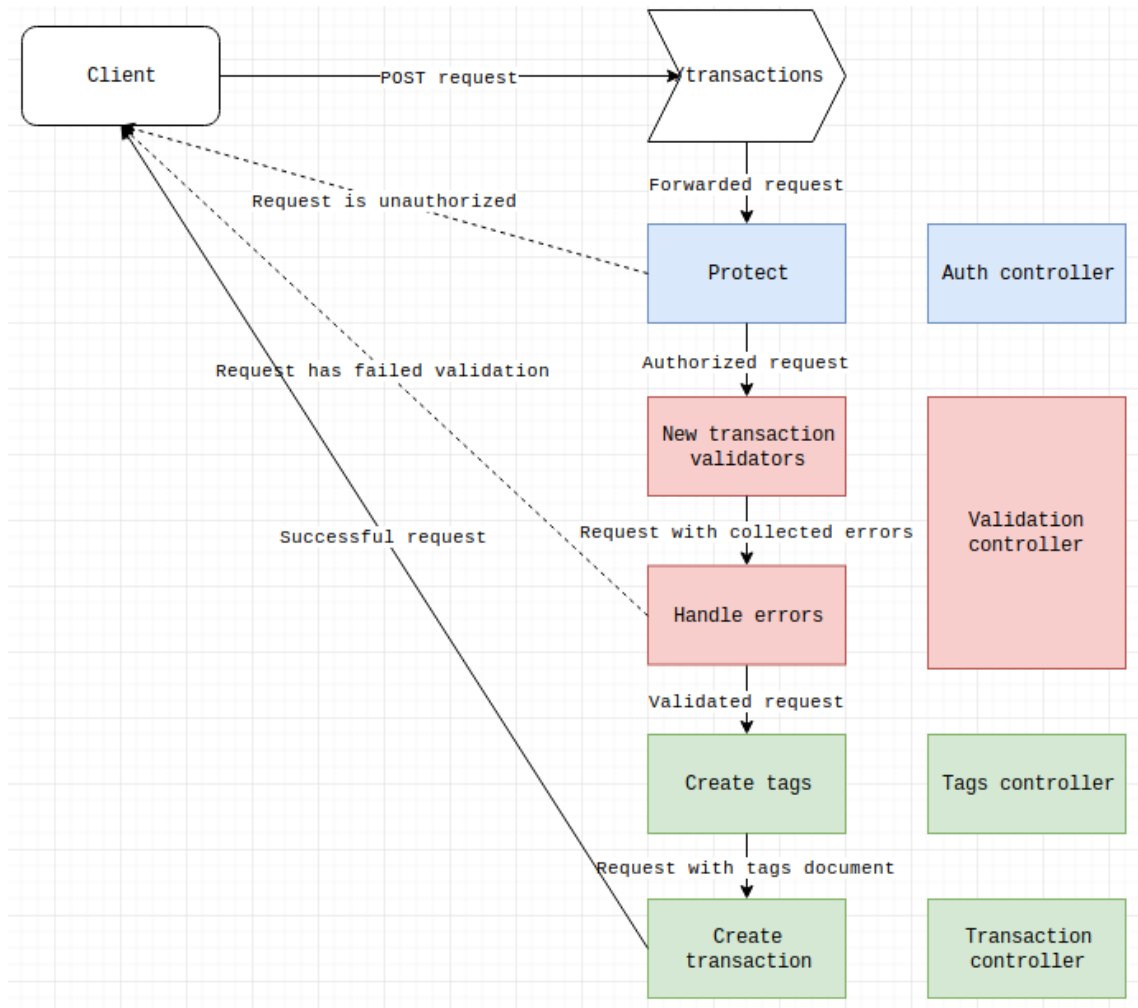


Figure 24: POST `/transactions` request handlers.

authorization header will be forwarded to the rest of the handlers. The validation handlers will only deliver valid requests so the rest of the controllers won't have to deal with potentially invalid data. An example handling of a `POST` request received on the `/transactions` path is shown in Figure 24.

Auth routes

backend/src/routes/authRoutes.js

- POST
 - /auth/register
 - * Protected: ✗
 - * Body properties: email, password, confirmPassword
 - * Description: Create a new user account
 - /auth/login
 - * Protected: ✗
 - * Body properties: email, password
 - * Description: Log in with an existing user account
 - /auth/change-password
 - * Protected: ✓
 - * Body properties: password, confirmPassword
 - * Description: Change the password for the logged in user

Coins routes

backend/src/routes/coinsRoutes.js

- GET
 - /coins
 - * Protected: ✗
 - * Description: Get the complete coin list
 - /coins/top/volume
 - * Protected: ✗
 - * Description: Get the top 10 coins by volume
 - /coins/top/market-cap
 - * Protected: ✗
 - * Description: Get the top 10 coins by market cap
 - /coins/market-data
 - * Protected: ✗
 - * Query string variables: symbols

- * Description: Get the current market data for `symbols`
- `/coins/historical`
 - * Protected: ✗
 - * Query string variables: `symbol`, `startDate`, `endDate`
 - * Description: Get the historical market data for `symbol` between `startDate` and `endDate`

Portfolio routes

`backend/src/routes/portfolioRoutes.js`

- GET
 - `/portfolios`
 - * Protected: ✓
 - * Description: Get all portfolio names for the authenticated user
 - `/portfolios/:portfolio`
 - * Protected: ✓
 - * Query string variables: `tags`
 - * Description: Get the full portfolio information for `portfolio`, filtered by `tags`
 - `/portfolios/:portfolio/historical`
 - * Protected: ✓
 - * Query string variables: `tags`, `date`
 - * Description: Get historical portfolio values since `date`, filtered by `tags`

Transaction routes

`backend/src/routes/transactionRoutes.js`

- POST
 - `/transactions`
 - * Protected: ✓
 - * Body properties: `symbol`, `portfolio`, `date`, `amount`, `price`, `type`, `tags`
 - * Description: Create a new transaction with values in the request body

- PUT
 - `/transactions/:id`
 - * Protected: ✓
 - * Body properties: `symbol`, `portfolio`, `date`, `amount`, `price`, `type`, `tags`
 - * Description: Update the transaction which has the same ID as `id` with values in the request body
- DELETE
 - `/transactions/:id`
 - * Protected: ✓
 - * Description: Delete the transaction which has the same ID as `id`

3.7 React application

The *React* application is built up from three different parts: components, contexts and hooks. The components are the reusable building blocks of the UI, the contexts hold the application data and the hooks can inject state into a component. Several additional *JavaScript* libraries were included during development, here is a list of the most important ones:

- **Chart.js**: Simple charting library
- **Create React App**: Build configuration and scripts to set up a *React* app
- **Day.js**: Fast 2kB alternative to Moment.js with the same API
- **downshift**: Primitives to build simple, flexible *React* autocomplete/dropdown/select/combobox components
- **Formik**: *React* form builder library
- **Material UI**: *React* components that implement Google's Material Design
- **normalizr**: Utility for taking *JSON* with a schema definition and returning nested entities with their IDs, gathered in dictionaries
- **Numeral.js**: Library for formatting and manipulating numbers
- **Prop Types**: Runtime type checking library for *React* props and similar objects
- **React Router**: Declarative routing library for *React*

- **React Router DOM:** DOM bindings for *React Router*
- **Yup:** Object schema validator and object parser

3.7.1 Hooks

Before *React 16.8*, reusing stateful logic was hard, despite few design patterns trying to solve this problem, they required to restructure and create wrappers around a component. The *Hooks* support was released in *React 16.8.0* and they let the developers reuse stateful logic without changing the component hierarchy and use more of *React's* features without classes. The following built-in *React* hooks [27] were used in components and in custom hooks:

- **useState:** Returns a stateful value, and a function to update it.
- **useEffect:** Side effects are not allowed inside the main body of a function component, instead an effectful function can be passed to **useEffect**, which will run after renders.
- **useContext:** Accepts a context object and returns the current context value for that context.
- **useReducer:** **useState** alternative which accepts a function of type `(state, action) => newState`; used in API hooks to handle multiple values in one state change
- **useCallback:** Returns a memoized [28] callback.
- **useMemo:** Returns a memoized value.
- **useRef:** Returns an object that will persist for the full lifetime of the component; used store specific data between effectful changes.

Several custom hooks were built to enhance common component logic. These are detailed in the following paragraphs.

API hooks

`frontend/src/hooks/useApiCallback.js`

`frontend/src/hooks/useApiOnMount.js`

These two hooks accept an API call method and the data to be sent with the request as their arguments. They use the **useEffect** hook to perform the request side effects and save the response in their state. The hooks will return their state after each iteration of state change. The **useApiOnMount** sends a request on mount or every time the data arguments change. The **useApiCallback** will only send a request if its returned callback is called.

Auth hooks

`frontend/src/hooks/useRegister.js`

`frontend/src/hooks/useLogin.js`

`frontend/src/hooks/useLogout.js`

The `useRegister`, `useLogin` hooks will store the user information in the user context after a successful register or login respectively, and the `useLogout` hook can be called to remove this user information.

3.7.2 Contexts

Contexts provide a way to pass data through the component tree without having to pass props down manually at every level. Generally, data is passed from the parent component to its child via props, but this can be cumbersome if certain props are required by many components. Shared values can be stored in these contexts without having to explicitly pass them through every level of the tree. Components which are the children of a context's **Provider** component can subscribe to context changes. The implementation of these providers will be detailed in the Components section 3.7.3. A component is subscribed to a context with the `useContext` hook which will return the current context value. Context changes will trigger a rerender for every subscribed component. These four contexts were built to hold data for different levels of the application:

- Coins context: Holds the complete coin list
- User context: Holds the user data
- Dashboard context: Holds dashboard layout information
- Portfolio context: Holds data about the selected portfolio

The composition of the contexts is displayed in Figure 25.

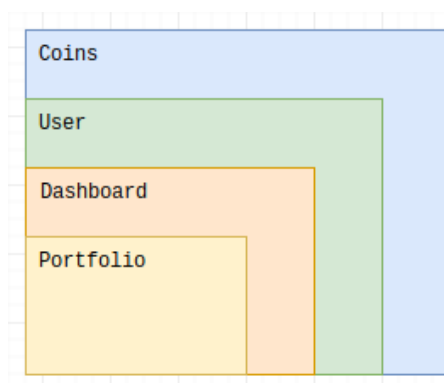


Figure 25: Composition of the context providers in each other.

3.7.3 Components

The components are semantically organized into four groups. Every context object comes with a **provider** component that allows its descendants to consume the context's data. **Containers** are components which have stateful logic or they are subscribed to a context. The simplest components are the **views**, they have no dependencies on the rest of the application, they receive data and callbacks exclusively via props and they have no state. The **pages** are top-level components which are consumed by a *React Router* component to display a page. They are composed of several other containers and views. An example dashboard component tree, with two positions and total of four transactions, is displayed in Figure 26.

User provider

`frontend/src/components/providers/UserProvider`

The User provider component exposes the current logged in user's data with two additional methods to its subscribers. After a successful login the `storeUser` method will be called, which will save the user data from the login response in local storage. The `removeUser` method removes the user data from local storage. When the provider mounts, it will try to get an existing user who was previously logged in, and if it can retrieve the user data, it will initialize the user context's user object with it, this will keep the user logged in between visits.

Dashboard provider

`frontend/src/components/providers/DashboardProvider`

This provider holds all the dashboard data and its helper methods, this grants subscriber components the ability to toggle the portfolio drawer, select a different portfolio, toggle the various dialogs and set the portfolio filter options.

Portfolio provider

`frontend/src/components/providers/PortfolioProvider`

The Portfolio provider receives the selected portfolio's name and sends a request to the server for the portfolio data.

The API returns *JSON* data that has deeply nested objects and arrays, using this data efficiently is difficult for a *JavaScript* application, therefore the response data are transformed automatically into dictionaries with the help of the *normalizr* [29] library. The *normalizr* library takes the *JSON* data and schema information about an object's ID attribute and returns nested entities with their IDs, gathered in dictionaries. The default ID attribute is `id`, but position and portfolio entities

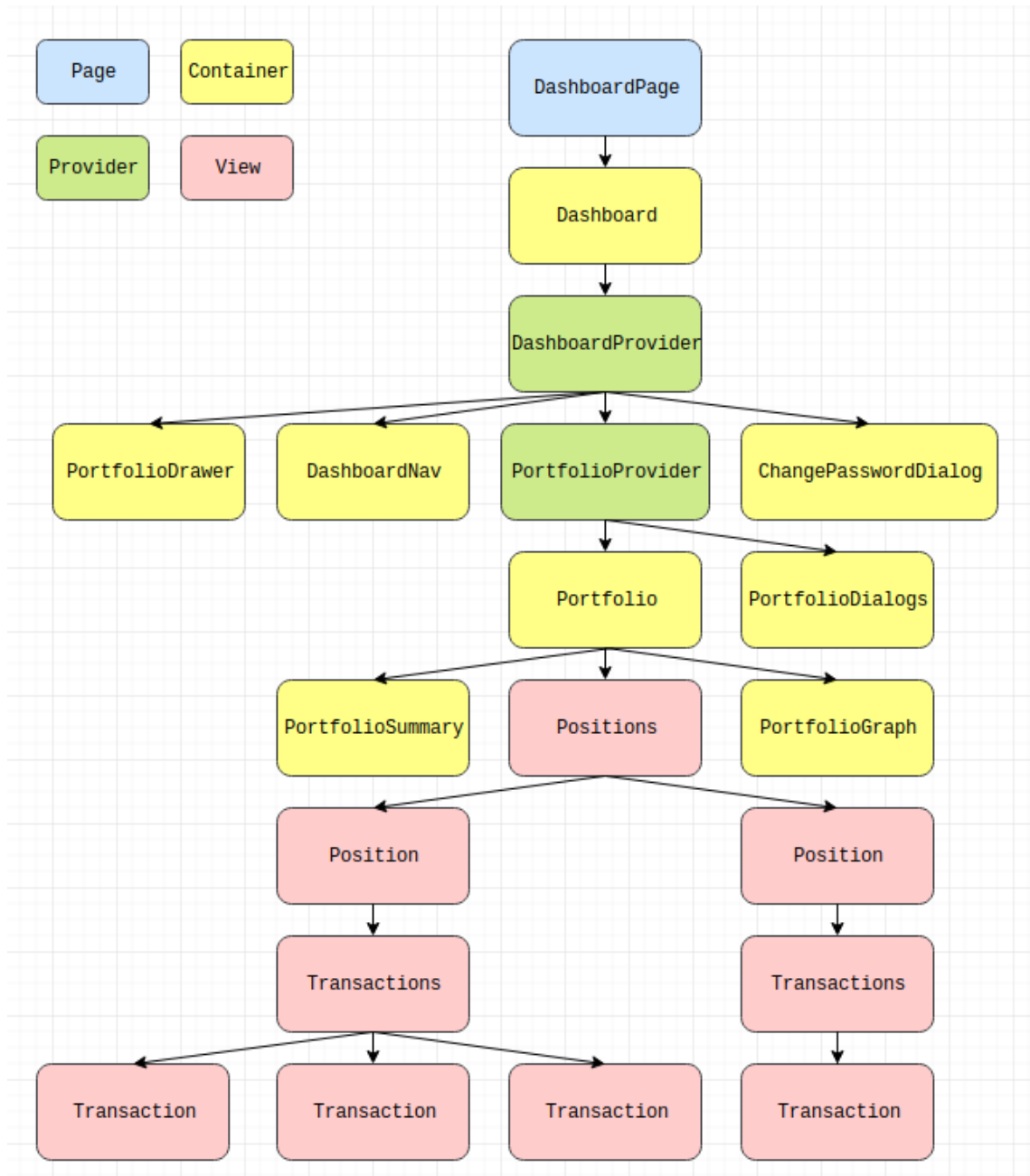


Figure 26: An example dashboard page component tree.

have no `id` fields, their `symbol` and `name` attributes are used as unique IDs instead. The relationship between the entities is shown in Figure 27. An example response data normalization is shown in Figure 28.

To further increase the efficiency of data retrieval, all getter methods are memoized with the `useMemo` and `useCallback` hooks, by storing the results of function calls and returning the cached result when the same input occur again.

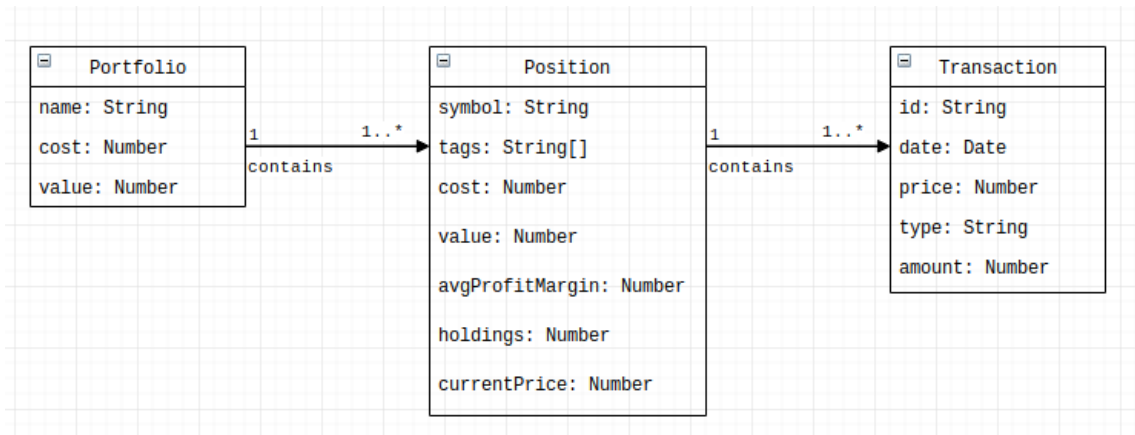
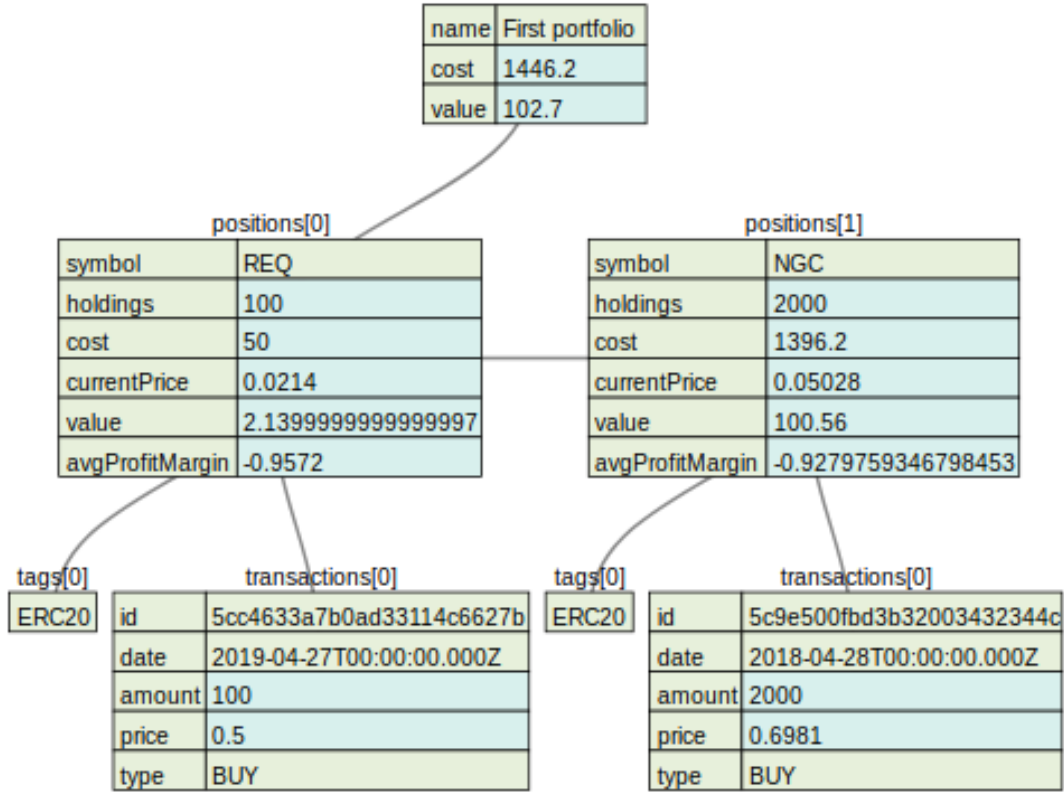


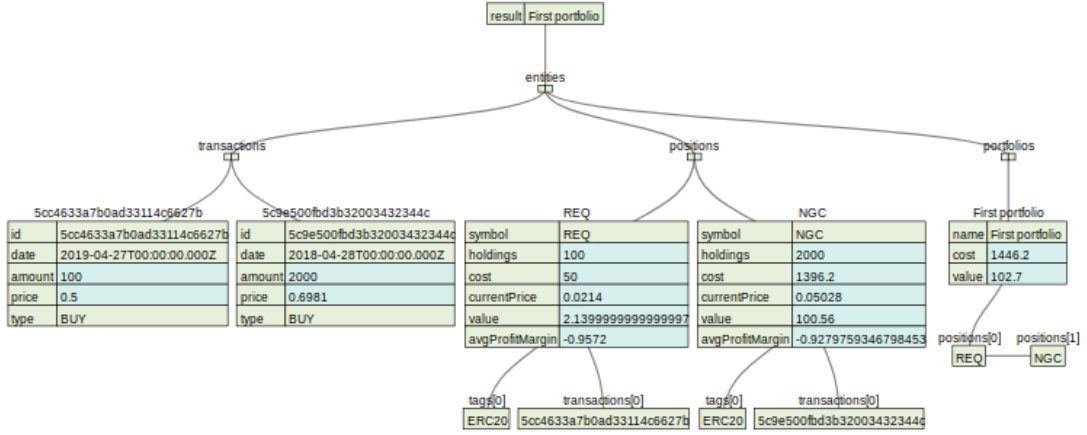
Figure 27: Entity relationship.

Rest of the components

The core logic of the *React* application is implemented in the previously described three providers. The container components subscribe to one of the context providers to receive parsed data and the callbacks for interaction, then they pass down the necessary data as props to their children.



(a) Example API response data visualized before normalization.



(b) The same response data visualized after normalization.

Figure 28: Normalization.

3.8 Testing

The tests were implemented with the *Jest* [30] *JavaScript* testing framework.

3.8.1 Integration tests

The *mongoose* models' custom hooks and complex query methods were tested combined with a live *MongoDB* connection on a test database. The purpose of this level of testing is to guarantee the validity of persisted and retrieved data.

```
PASS src/models/Transaction.test.js
Transaction
  getSymbols
    ✓ should return all of the user's symbols (44ms)
    ✓ should return symbols with matching portfolio (104ms)
    ✓ should return symbols which were added before the specified date (105ms)
    ✓ should return symbols which were added before the specified date with matching portfolio (105ms)
  getPositions
    ✓ should return the user's transactions grouped into positions (55ms)
    ✓ should return the user's transactions grouped into positions filtered by tags (52ms)
    ✓ should return positions with transactions, which were added before the specified date (42ms)
    ✓ should return positions with transactions, which were added before the specified date, filtered by tags (37ms)
  getPositionsForEachDayBetweenDates
    ✓ should return correct positions for each day between dates (52ms)
```

(a) Transaction model tests.

```
PASS src/models/Tags.test.js
Tags
  addToSet
    ✓ should add new element (372ms)
    ✓ should add multiple new elements (87ms)
    ✓ should only add new elements (72ms)
    ✓ should not have the same elements (38ms)
    ✓ should not add any elements (76ms)
```

(b) Tags model tests.

Figure 29: Integration testing results.

3.8.2 End-to-End tests

The REST API endpoints were tested with end-to-end tests to ensure that the flow of a HTTP request performs as designed from start to finish. The API is tested in a real-world scenario while communicating with the database and the *CryptoCompare API*.

```
PASS src/routes/transactionRoutes.test.js (7.819s)
transactionRoutes
  POST /transactions
    ✓ should create new transaction (386ms)
    ✓ should not create new transaction without authorization (9ms)
    ✓ should not create new transaction without portfolio (273ms)
    ✓ should not create new transaction without symbol (253ms)
    ✓ should not create new transaction with invalid symbol (1195ms)
    ✓ should not create new transaction without date (269ms)
    ✓ should not create new transaction with wrong date format (261ms)
    ✓ should not create new transaction without amount (329ms)
    ✓ should not create new transaction without price (289ms)
    ✓ should not create new transaction without type (272ms)
    ✓ should not create new transaction with wrong type (308ms)
  PUT /transactions/:id
    ✓ should update transaction (721ms)
    ✓ should not update transaction without authorization (408ms)
    ✓ should not update the portfolio of a transaction (469ms)
    ✓ should not update the symbol of a transaction (754ms)
  DELETE /transactions/:id
    ✓ should delete transaction (604ms)
    ✓ should not delete any transaction without authorization (14ms)
```

Figure 30: Transaction End-to-End testing results.

```

PASS src/routes/authRoutes.test.js
authRoutes
  POST /auth/register
    ✓ should register (689ms)
    ✓ shouldn't register without email (36ms)
    ✓ shouldn't register without password (33ms)
    ✓ shouldn't register without confirm password (39ms)
    ✓ shouldn't register without matching passwords (44ms)
  POST /auth/login
    ✓ should login (302ms)
    ✓ shouldn't login without email (2ms)
    ✓ shouldn't login without password (2ms)
    ✓ shouldn't login with wrong password (298ms)
  POST /auth/change-password
    ✓ should change the password (655ms)
    ✓ should not change not matching password (36ms)
    ✓ should not change password without authorization (9ms)

```

Figure 31: Authentication End-to-End testing results.

```

PASS src/routes/portfolioRoutes.test.js (10.146s)
portfolioRoutes
  GET /portfolios
    ✓ should return all portfolio names (102ms)
  GET /portfolios/:portfolio
    ✓ should return portfolio (455ms)
    ✓ should not return portfolio with invalid name (83ms)
    ✓ should not return portfolio without authorization (5ms)
  GET /portfolios/:portfolio/historical
    ✓ should return historical portfolio (975ms)
    ✓ should not return portfolio without date (110ms)
    ✓ should not return portfolio with invalid date (115ms)
    ✓ should not return portfolio with invalid name (111ms)
    ✓ should not return portfolio without authorization (9ms)

```

Figure 32: Portfolio End-to-End testing results.

3.8.3 Manual tests

The *React* application's behavior was tested with manual tests.

Coin browser

- The cryptocurrency search query is only executed after the user has stopped typing in the input for 1 second.
- The navigation items are correctly changed if the user is already logged in.

Detailed coin page

- The detailed coin page is accessible from the dashboard, top lists page and by visiting the symbol's URL.
- If the detailed coin page is accessed via typing in the symbol's URL and the symbol isn't found, an error message is shown.
- The "From" date field's maximum value is the previous day.
- The "To" date field's maximum value is the current day.

Registration

- "Email", "password" and "confirm password" fields show error messages after they've been touched but left empty.
- "Email" field shows error message if the email format is not correct.
- "Confirm password" field shows error message if its value doesn't equal the value in the "password" field.
- The form can't be submitted with invalid data.
- "Email" field shows error message after form submit if the email is already taken.

Login

- "Email" and "password" fields show error messages after they've been touched but left empty.
- "Email" and "password" fields show error messages after the form has been submitted with wrong credentials.
- The form can't be submitted with invalid data.
- When the user returns to the website, they are automatically identified.

Change password

- "Password", "confirm password" fields show error messages after they've been touched but left empty.
- "Confirm password" field shows error message if its value doesn't equal the value in the password field.
- The form can't be submitted with invalid data.
- Confirmation message is shown after a successful password change.

Dashboard

- Portfolio filter query request is sent only after the user hasn't been actively clicking the tags for at least 1 second.
- Selecting a tag in the input field colors the active tags in the position items.
- The "From" date field's minimum value is the date of the earliest transaction.
- The "From" date field's maximum value is the current date.
- Only one dialog can be open at the same time.
- Selecting "New portfolio" option in the drawer opens the new portfolio form.
- Selecting a different portfolio in the drawer reloads the dashboard with the selected portfolio's information.
- Clicking on a position's symbol opens its detailed page.

Transaction form

- "Coin", "amount" and "price per unit" fields show error messages after they have been touched but left empty.
- The "Date" field's maximum value is the current date.
- The "Amount" field shows error message if its value is negative.
- The "Price per unit" field shows error message if its value is negative.
- Typing in the "Coin" field shows the 5 most similar search results under it.
- Clicking on a coin in the open search results will select that cryptocurrency.
- After selecting a coin that already has a position with tags, the form's tags array is filled with those tags.

- The form can't be submitted with invalid data.
- Submitting the portfolio data is recalculated.

Update transaction form

- "Coin" field is disabled.
- The form's fields are prefilled with the appropriate transaction's data.
- The form can't be submitted with unchanged or invalid data.

Delete transaction

- Transaction is only deleted after the action is confirmed in the confirmation dialog.
- If a transaction is deleted, the portfolio data will be recalculated.

3.9 Deployment

The database is hosted on a cloud *MongoDB* service provided by *MongoDB Atlas* [31]. The working application is deployed on the *Heroku* [32] cloud platform, but before new code can be deployed, the integration and end-to-end tests must be completed, then an optimized production build has to be created from the *React* application, this is the responsibility of the *Travis CI* continuous integration service.

1. Code is pushed to the `master` branch.
2. *Travis CI* sets up a virtual environment with *Node.js* and *Yarn*.
3. *Travis CI* installs all dependencies.
4. *Travis CI* runs all tests, if any of the tests fail the deploy will terminate.
5. After all tests passed, a production bundle is built from the *React* app.
6. *Travis CI* connects to *Heroku* and deploys the application.
7. *Heroku* restarts the application.
8. The new version is available at: <https://walfo.herokuapp.com>.

3.10 Possibilities for further development

Several improvement opportunities were identified during the development and the testing phase.

One of the most obvious limitations is the lack of responsive web design. All of the features are available on mobile but the UI isn't optimized to properly fit small screen sizes.

The consistency of input transactions are entirely the user's responsibility, the application allows to create, update or delete any transaction without checking if they have any dependencies. This could be unintuitive at first for the user, but this design choice was made to allow the convenience of changing any transaction, and because this is the solution which most of the popular, similar applications, like *Blockfolio* [33] or *Delta* [34] have also chosen.

One other unintuitive feature is that tags already added to a position can't be modified or deleted, only new ones can be added. It would be useful for users to be able to modify existing tags.

The *CryptoCompare API* offers market data in multiple target currencies, like EUR or GBP, but this application only utilizes the USD market data, adding the ability to change a portfolio's currency could help users who don't want to track their portfolio in US Dollars.

On the technical side, building multiple context objects instead of using *Redux* [35] architecture proved to be the wrong choice. The deeply nested structure of the contexts and their dependencies on each other cause unnecessary rerenders in lot of the components. The frontend application doesn't have meaningful logic in it, except the dashboard operations, therefore most of the actions will send a request to the REST API which result in frequent updates to the context variables. For these reasons, one predictable state container for the entire application would have been the better choice.

4 References

- [1] J. Lansky, “Possible State Approaches to Cryptocurrencies,” *Journal of Systems Integration*, vol. 9, no. 1, p. 19–31, 2018.
- [2] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System,” p. 1–9, Oct 2008.
- [3] CoinMarketCap, “All cryptocurrencies.” <https://coinmarketcap.com/all/views/all/>. Accessed on 2019-04-23.
- [4] J. Chen, “Portfolio.” <https://www.investopedia.com/terms/p/portfolio.asp>. Accessed on 2019-04-23.
- [5] “React – A JavaScript library for building user interfaces.” <https://reactjs.org/index.html>. Accessed on 2019-04-23.
- [6] “JavaScript.” <https://developer.mozilla.org/en-US/docs/Web/JavaScript>. Accessed on 2019-04-23.
- [7] “Express - Node.js web application framework.” <https://expressjs.com/>. Accessed on 2019-04-23.
- [8] N. j. Foundation, “Node.js.” <https://nodejs.org/en/>. Accessed on 2019-04-23.
- [9] “Java.” <https://go.java/index.html?intcmp=gojava-banner-java-com>. Accessed on 2019-04-23.
- [10] “ECMAScript 6: New Features: Overview and Comparison.” <http://es6-features.org/>. Accessed on 2019-04-23.
- [11] J. Chen, “Market Capitalization.” <https://www.investopedia.com/terms/m/marketcapitalization.asp>. Accessed on 2019-04-27.
- [12] A. Hayes, “Volume.” <https://www.investopedia.com/terms/v/volume.asp>. Accessed on 2019-04-27.
- [13] A. Barone, “Ticker Symbol Definition.” <https://www.investopedia.com/terms/t/tickersymbol.asp>. Accessed on 2019-04-24.
- [14] “What Is MongoDB?.” <https://www.mongodb.com/what-is-mongodb>. Accessed on 2019-04-28.
- [15] “NOSQL Databases.” <http://nosql-database.org/>. Accessed on 2019-04-28.

- [16] E. F. Codd, “A relational model of data for large shared data banks,” *Commun. ACM*, vol. 13, pp. 377–387, June 1970.
- [17] T. Haerder and A. Reuter, “Principles of transaction-oriented database recovery,” *ACM Comput. Surv.*, vol. 15, pp. 287–317, Dec. 1983.
- [18] “Fielding Dissertation: CHAPTER 5: Representational State Transfer (REST).” https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. Accessed on 2019-04-28.
- [19] “JSON.” <https://www.json.org/>. Accessed on 2019-04-28.
- [20] “Thinking in React – React.” <https://reactjs.org/docs/thinking-in-react.html>. Accessed on 2019-04-28.
- [21] “Composition vs Inheritance – React.” <https://reactjs.org/docs/composition-vs-inheritance.html>. Accessed on 2019-04-28.
- [22] “GitHub.” Accessed on 2019-05-02.
- [23] “Sass: Syntactically Awesome Style Sheets.” <https://sass-lang.com/>. Accessed on 2019-05-02.
- [24] N. Provos and D. Mazieres, “A future-adaptable password scheme,” 03 2001.
- [25] “cryptocompare.” <https://www.npmjs.com/package/cryptocompare>. Accessed on 2019-04-30.
- [26] J. Bradley, N. Sakimura, and M. Jones, “JSON Web Token (JWT).” <https://tools.ietf.org/html/rfc7519>. Accessed on 2019-04-30.
- [27] “Hooks API Reference – React.” <https://reactjs.org/docs/hooks-reference.html>. Accessed on 2019-05-12.
- [28] “Memoization.” <https://en.wikipedia.org/w/index.php?title=Memoization&oldid=864455867>, Oct. 2018. Page Version ID: 864455867, Accessed on 2019-05-12.
- [29] P. Armstrong, “Normalizes nested JSON according to a schema..” <https://github.com/paularmstrong/normalizr>, May 2019. Accessed on 2019-05-12.
- [30] “Jest · Delightful JavaScript Testing.” <https://jestjs.io/index.html>. Accessed on 2019-05-02.
- [31] “Fully Managed MongoDB, hosted on AWS, Azure, and GCP.” <https://www.mongodb.com/cloud/atlas>. Accessed on 2019-05-02.

- [32] “Cloud Application Platform | Heroku.” <https://www.heroku.com/>. Accessed on 2019-05-02.
- [33] “Blockfolio.” <https://blockfolio.com/>. Accessed on 2019-05-02.
- [34] “Delta - The best Bitcoin, ICO & cryptocurrency portfolio tracker.” <https://delta.app>. Accessed on 2019-05-02.
- [35] “Redux.” <https://redux.js.org/>. Accessed on 2019-05-02.