



Project for the Web and Internet Engineering Course  
Sebastian Cavada, Riccardo Rigoni, Alessandro Gottardi, Luca Taddeo

## Introduction

This report provides explanation and documentation about *stonX*, a web application aimed to easily access the trading world through a clear and simple interface.

## User Stories

Some user stories that we found interesting and on which we based the development and implementation of our solution

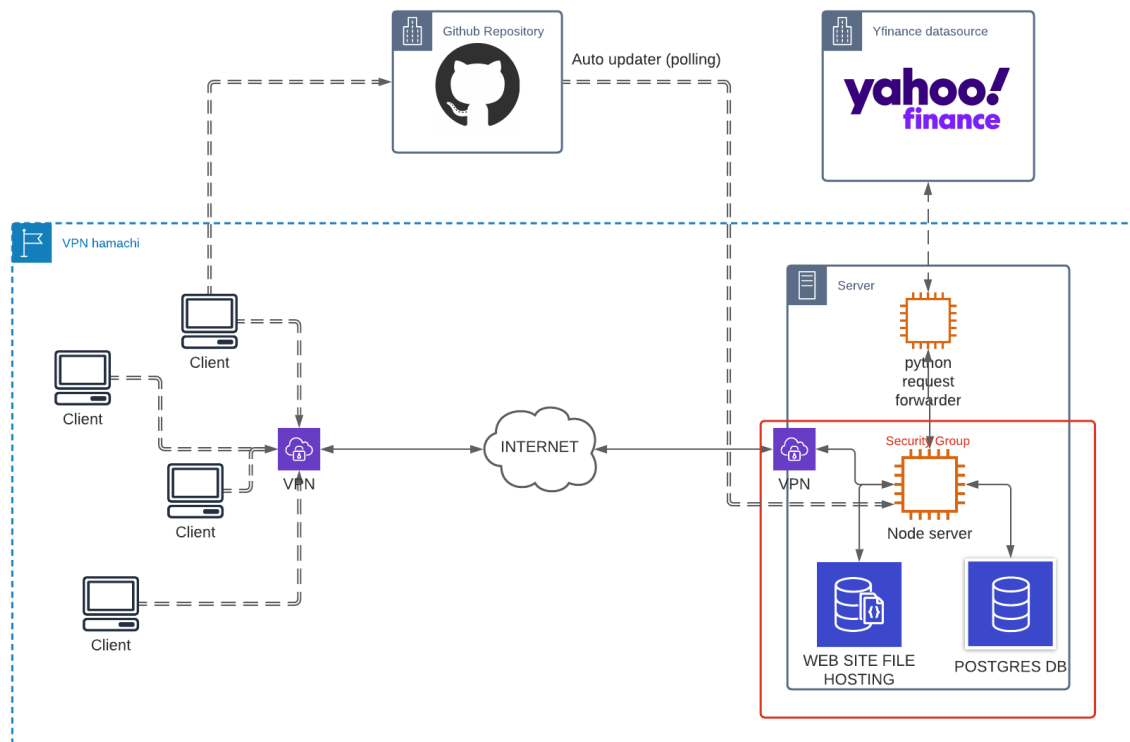
<b>Title:</b> Search stocks through many global stock markets	<b>Priority:</b> HIGH	<b>Estimate:</b> 3 weeks
<b>As a</b> <i>small investor</i>  <b>I want to</b> <i>search and visualize the data of some stocks in the form of a graph</i>  <b>so that I can</b> <i>make better investment decisions</i>		
<b>Acceptance criteria</b>  <b>Given</b> <i>a number of stocks available</i>  <b>When</b> <i>when the user carries out a search in the website</i>  <b>Then</b> <i>some results should appear also with related graphs and data</i>		

<b>Title:</b> Follow favourite stocks	<b>Priority:</b> Medium	<b>Estimate:</b> < 1 week
<b>As a</b> <i>student</i>  <b>I want to</b> <i>monitor the market</i>  <b>so that I can</b> <i>buy the stock at the right time, to maximize the profit.</i>		
<b>Acceptance criteria</b>  <b>Given</b> <i>some stocks</i>  <b>When</b> <i>the user clicks on the heart shaped button</i>  <b>Then</b> <i>the selected stock will be added to the list of favorite so it will appear in the landing page for faster access.</i>		

<b>Title: Get personalized news</b>	<b>Priority: Medium</b>	<b>Estimate: &gt; 2 week</b>
<p><b>As a retail investor</b></p> <p><b>I want to</b> <i>get the latest news on the market</i></p> <p><b>so that I can</b> <i>better understand the next moves of the market and act accordingly.</i></p>		
<p><b>Acceptance criteria</b></p> <p><b>Given</b> <i>some liked stocks</i></p> <p><b>When</b> <i>the user enters the landing page</i></p> <p><b>Then</b> <i>the news, based on the follows he has and the searches that he performed, should appear as most relevant.</i></p>		

<b>Title: See all the liked stock together</b>	<b>Priority: Low</b>	<b>Estimate: &lt; 2 week</b>
<p><b>As an investor</b></p> <p><b>I want to</b> <i>get the list of all the stock I like</i></p> <p><b>so that I can</b> <i>have a faster visual insight on all my stock.</i></p>		
<p><b>Acceptance criteria</b></p> <p><b>Given</b> <i>some liked stocks</i></p> <p><b>When</b> <i>the user enters the "my stocks area"</i></p> <p><b>Then</b> <i>the list of the stock that he/she likes will be displayed all together in a single page</i></p>		

# System architecture



The system architecture is based on a client-server pattern. Therefore, a main server hosting the web application serves multiple incoming requests from the clients that wish to interact with the app.

The primary server hosting the web application has been developed using the Express module and it has been deployed in the Node.js runtime environment. This server provides the APIs that support the web pages of the application too.

The frontend application has been developed using the ReactJs framework. Even though in the WIE course two different languages have been taught to implement front- and backend, we preferred to use a single programming language for the two application parts, hence the choice to develop both the frontend and the backend with JS-powered frameworks.

However, the data retrieval and manipulation part were not developed using JS, since the fundamental library providing the financial data (yfinance) is written in python. Therefore, we had to implement an additional server in this different programming language to be able to retrieve the data.

The implementation of this additional support data-retrieve server gave advantages to the entire system infrastructure too, since using this system architecture, the data retrieval and manipulation are not coupled to the primary Express server. Therefore, if one of them fails, the other is not affected, and furthermore, the workload for the primary server is drastically reduced. For these reasons, the additional python server turned out to be a winning choice, that made the web application faster, more robust and more reliable.

To store data of interest for the application we mainly used two databases: Postgres and Redis. The first has been used to store information about the stock and the shareholders and the second to store user session information and user personalization.

*During the whole development, the stack was not only decided on our prior knowledge of the languages/framework, but it was mainly derived from curiosity from the application and testing of new technologies. We always tried to learn something new, therefore we introduced new technologies that we found interesting to learn.*

## Development process

From the beginning of the project, we made the choice to divide ourselves into two teams, one for front-end and the other for back-end development. Implementation was carried on using a scrum-based methodology: we kept in touch with video-calls and text messages and every week we planned a one-week sprint, at the end of which a plenary meeting was planned to inform all the members about the progress and possible issues encountered.

## Frontend

As previously mentioned, the frontend implementation is based on React, a popular framework created by Facebook. Reasons for this choice lie in the availability of documentation and code examples, as well as good performance and ease of use. In particular, the framework allows writing so-called "components" that can be reused in different pages and even shared with different developers.

We strive to build an interface that is intuitive, essential, and easy to use. For this reason, Material Design was immediately chosen as the reference design in our application. We started by designing detailed mockups of each page using Figma (*ref. Mockup Images*), which have then been implemented using Material UI, together with Nivo for the chart representations. This allowed us to fulfil all the aforementioned objectives in a fast and productive way.

Communication with the back-end API was possible by using Axios, an HTTP client library with very good browsers compatibility (from Internet Explorer onwards). In order to test functionalities without having to continuously contact the server, a series of "mock-up data" was locally saved in the frontend code. Thanks to some environment variables, developers are given the ability of switching between the two different modality and to specify the backend address.

## Overall Architecture

Source code under the `src` folder have been divided into the following subdirectories:

- `api`: logic classes and files related to the client-server communication;
- `components`: view files dedicated to specific components, used in several pages;
- `context`: logic files providing a repository of data, accessible by any part of the code. Most notably, they are used to save the user state and the current theme;
- `fonts`: location of the font files used in the app ([Lato](#), under Open Font License);
- `pages`: view files describing the content of each and every page in the website;
- `utils`: other logic files, needed by various components.

Among all the files, some deserve a mention. `index` is the first element that gets rendered by React's Virtual DOM. It makes available the two context elements described before and most importantly it provides routing capabilities to the whole application (as defined in the `routes` file). `PageContainer` receives a route and a page to render (from the so-called "props") and decides whether it is needed to render header and navbar. It also listens for modifications in the user's screen width, so as to dynamically resize all the components and to provide a better user experience when using a mobile device. `MarketChart` is an adapter class over the complex but well stocked Nivo library. We tried to configure all the relevant knobs so as to hide the complexity and to expose just those parameters that were really needed. With respect to charts showing multiple series, normalization has been implemented in order to ensure that all the series are readable and well scaled.

## Backend

The backend has been developed incrementally, starting from the data retrieval and manipulation part and then developing the API endpoints making the managed data accessible.

### Data Retrieval and Manipulation

The data of interest for the application are of two types: the financial information and the user information.

The financial information is stored in a PostgreSQL database instance, and price values for the stocks are retrieved and updated from the Python server.

The user information is stored both in PostgreSQL and in Redis. User account information such as the email, the name and the password are kept in the PostgreSQL db instance together and the user session information is stored in an in-memory Redis instance. The decision to split them and exploit different technologies has been taken due to the different nature of this information. On one hand, the user account info cannot be lost and is likely to be rarely updated, therefore, the PostgreSQL fits well in this context. On the other hand, user session info is more frequently updated, and they must be associated with each request object starting from the `SessionId` stored in the cookie. Therefore, we thought that an in-memory db would fit better in this context than a SQL database, and Redis proved to be faster in responding than its SQL counterpart.

## Data Retrieval Server:

It has been written in python using the Flask framework and it is devoted to forward HTTP Requests from the application hosting server to Yahoo Finance endpoints. It acts as a middleware among the two endpoints filtering the communications and rejecting a priori Bad Requests coming from the Web Application. It also acts as a Cache that keeps relevant data once they have been retrieved from Yahoo Finance to update the financial information stored in the PostgreSQL database.

The main idea behind this System Architecture was to make different microservices in the backend so to have different software for each functionalities, decoupling each of them from the others. This approach seemed to be more effective and more reliable than its monolithic counterpart, since it seemed to scale better and to be not only more reliable, but also more responsive and efficient.

It is worth clarifying how the Python Server acts as a Cache to update the Financial Information.

Since each set of historical data retrieved from Yahoo Finance always contains the most recent available price for the related Stock, the python server always stores the last price value with its relative timestamp in a Redis storage after having returned the set of historical data to the Express server. In this way, the Redis storage has the most recent available prices for the searched stocks without the need to make an additional request to Yahoo Finance, and are the users with their requests that keeps the information up-to-date.

A Python BackgroundScheduler instance runs inside the Python server and it periodically flushes the Redis content into the PostgreSQL database, updating the stock prices.

This cache implementation has been largely debated but we agreed that it was the best solution. In fact, as the application scales, are the users that keep the information they need up-to-date, and lot of Yahoo Finance requests could be avoided exploiting this technique.

## Hosting Server:

Written in JavaScript, it handles all the requests from the client and interacts with the database or with the python server when financial or user data are to be retrieved.

The documentation for all the endpoints can be found at the following [link](#). The api are logically divided into folders, following the structure of the website and this approach has been found intuitive and effective both for the frontend and backend teams.

The developed API is a RestFul API, and the user information is kept among different HTTP requests exploiting Cookies.

We used the Node.js Sequelize module as the main interface to the SQL database, and this resulted in a huge amount of time saved, since the queries were automatically generated. The library was complete as we didn't need to write any native query even for the most complex situations.

It also offered some ready to use functions, for example adding a new entry without an id, which was automatically generated.

This server also acted as hosting to the web-site. So there was not the necessity to create another special purpose for just the web-site.

## Technologies

- Redis
- Cookies + sessions express-session
- Postgres
- Nodejs
- Sequelize
- Express
- Python
- Waitress (python enqueueur)
- yfinance
- postman (documentation)

## Project management

During the project the group divided itself into 2 different groups. One tackling the frontend and the other the backend. To keep ourselves in synchro, we extensively used most of the functionalities of github. We tried many of them, rejected some and stuck with others.

We created an organization, which is also a folder for more repositories, since we have more than one service to be built in parallel. Therefore this method helped us to keep things nice and tidy. Moreover since we had to keep in sync also the gitlab repositories, we added another endpoint on the push, so that gitlab would also be updated.

Branches were widely used by both teams, in order to avoid conflicts since most of the time we would work in parallel. This was not the only way to avoid code redundancy as the communication between the teams was overall always present.

The website was built in react and tested on a development purpose server (the computer of the web developer most of the times). When some changes were ready to be deployed, a build of the site was made and directly uploaded into the main branch in the node server. It was safe to do so, even though in a production environment we would have preferred to implement some github actions for continuous deployment over the other repository, so to run also some small test.

### Migration

Since the number of components is not very small, the migration could become somewhat a bit difficult. In order to limit possible errors generated from incorrect migration, we developed a standard procedure to be followed and we introduced some configuration files through which the different components could be dynamically customized.

Follows a description of the procedure to migrate the different app components:

- Databases

The database migration would be the first step of the migration. Since none of the data is kept in the cloud, a dump of both PostgreSQL and Redis has to be performed. With the dump the new db instance in the different locations could be repopulated.

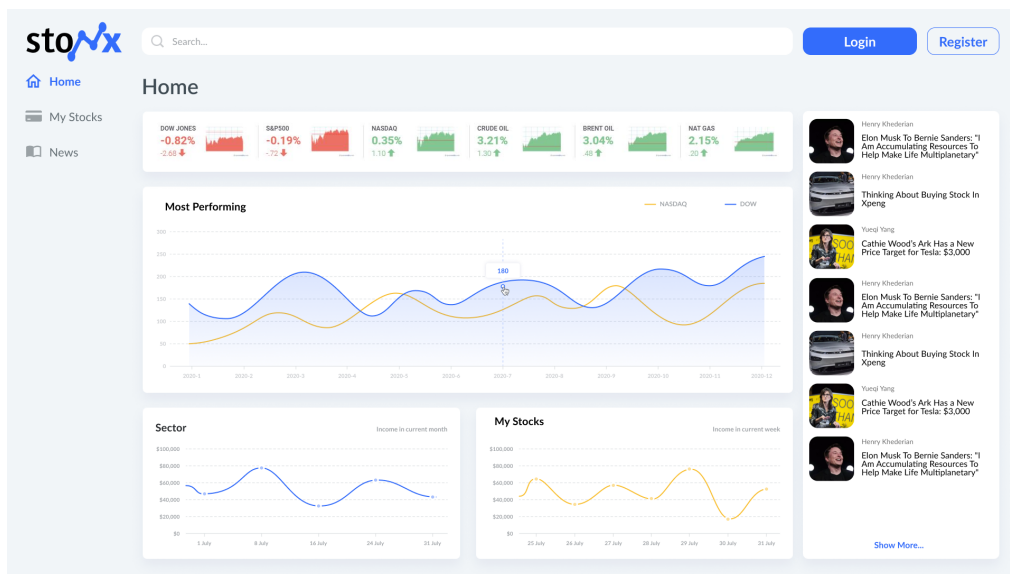
The software has to be installed on the new machine and then the import can start.

- Server

The services that have to be moved are 2. The forwarder server (or namely the server in python) can be moved at any time, but best if it is at first. Then nodejs can follow. To perform this action, we can just download the related repositories and install the libraries. The last step in this process would be to configure the .config file in nodeJS with the right endpoints and credentials for databases and python server.



# Mockup Images



The login page has a header with the stoNx logo and a search bar. The main heading is 'Access to your account'. It contains two input fields for 'Username' and 'Password'. Below these is a link: 'Don't have an account? Click here to [register!](#)'. At the bottom are two buttons: a blue 'Login' button and a white 'Cancel' button with a blue border.

The registration page features the stoNx logo and a search bar. The main heading is 'Create a new account' with a note '\* field is mandatory!'. It includes input fields for 'First Name\*', 'Last Name\*', 'Date Of Birth\*', 'Country', 'Email Address\*', 'Confirm Email Address\*', 'Password\*', and 'Confirm Password\*'. A section titled 'Select your interests!' contains checkboxes for various categories: Energy, Materials, Industrials, Utility, Healthcare, Financials, Consumer Discretionary, Consumer Staples, Information Technology, Communication Services, and Real Estate. At the bottom, there is a link: 'Already have an account? Click here to [login!](#)'. The page concludes with a blue 'Register' button and a white 'Cancel' button with a blue border.



Q TI

TSLA.MI	Tesla, Inc.	+ 120.2%
TSLA.MI	Tesla, Inc.	- 120.2%
TSLA.MI	Tesla, Inc.	+ 120.2%

stoNx

Q Search...

Home

My Stocks

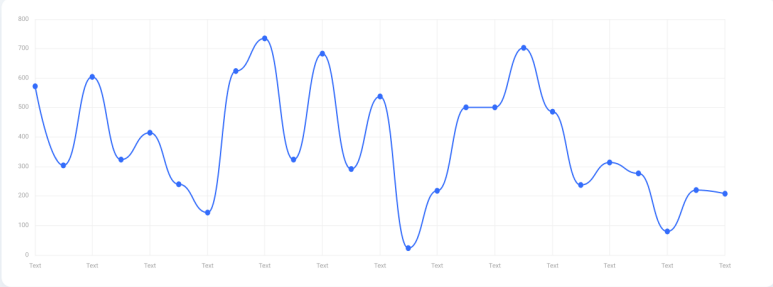
News

TSLA.MI

Tesla, Inc.

800 USD ▲0.13 (0.20%)

Nasdaq, dd/mm/yyyy hh:mm



Sector	Consumer Cyclicals	Market Cap	\$628.58B
Industry	Electric (Alternative) Vehicles	Net Debt & Pref	(\$7.64B)
Data 1	Value 1	Entprs. Value	\$620.93B
Data 2	Value 2	Beta	2.03X
Data 3	Value 3	Borrow Cost	0.25%

Henry Khederian

Elon Musk To Bernie Sanders: "I Am Accumulating Resources To Help Make Life Multiplanetary"

Henry Khederian

Thinking About Buying Stock In Xpeng

Yunqi Yang

Cathie Wood's Ark Has a New Price Target for Tesla: \$3,000

Henry Khederian

Elon Musk To Bernie Sanders: "I Am Accumulating Resources To Help Make Life Multiplanetary"

Henry Khederian

Thinking About Buying Stock In Xpeng

Yunqi Yang

Cathie Wood's Ark Has a New Price Target for Tesla: \$3,000

Henry Khederian

Elon Musk To Bernie Sanders: "I Am Accumulating Resources To Help Make Life Multiplanetary"

Show More...

Mario Rossi

Watching 5 stocks

▼

Dark Mode

Share

Logout