FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

STOCK

[Stock Prediction and Sentiment Analysis System]

Harun Ćerim, Ilda Balliu, Kaan Yöş, Mahran Emeiri

Prague, Czech Republic, 2020

Project name: Stock Price Prediction and Sentiment Analysis System [STOCK]

Authors: Harun Ćerim, Ilda Balliu, Kaan Yöş, Mahran Emeiri

Supervisor: doc. RNDr. Irena Holubová, Ph.D., Department of Software Engineering

Abstract: The aim of this project is to develop a tool that helps users gain insight on the stock market, by having easy access to the companies' current position, prices, outside affecting factors and also giving them an opportunity to build prediction models and customize them to their needs and interests, regardless of their knowledge in the field.

We would like to especially thank our project supervisor, Doc. RNDr. Irena Holubová, Ph.D. for her time and guidance on this journey. We would like to also give thanks to our friends and peers who took time to use our tool and gave us valuable feedback on our it as a possible target users of our platform, giving us a clearer view of the usage of STOCK in the real market.

Table of Contents

1. Introduction

This is a detailed documentation of the STOCK project, containing information about the tool itself as a whole and its use, the front end, back end and machine learning modules, explaining how these modules were built, how they work with each other to fulfill the purpose of the project and how to further extend and use them in the future.

1.1. The Team

~~Foivos Gkoulis~~	~~Software and Data Engineering, MFF UK~~
Harun Cerim	Software and Data Engineering, MFF UK (Back end application)
Ilda Balliu	Software and Data Engineering, MFF UK (Front end application and help with back end application)
Kaan Yöş	Software and Data Engineering, MFF UK (Machine Learning module)
Mahran Emeiri	Software and Data Engineering, MFF UK (Front end application and help with Machine Learning module)

1.2. Structure

This documentation starts with the Overview in section 2 where we discuss some similar tools, the use cases of our tool and the work timeline of our project. Section 3 describes the implementation overview with system architecture diagrams. Section 4 describes the Front-end module, including Installation, Components, Authentication, Data flow and more. In section 5, we describe the Back-end module with its Architecture, Core, Infrastructure, API and other components. The Machine Learning module is described in section 6 with details regarding all the models we used. Section 7 describes Integrations with other tools during development and we conclude the document with some ideas for future expansion and implementation of the project in section 8.

2. Overview

STOCK is a tool where users can gain more insight on the stock market and how their favorite companies are positioned in it. The main features it provides are:

- Sentiment analyzed articles using Vader.
- Company profile with historical data and predicted prices using Recurrent Neural Network.
- Building Machine Learning models for price prediction with full customizability.

However, it goes without discussion that there are similar tools out there which provide similar services. A few of them are described and discussed in the next section.

2.1. Related Works

While shaping our idea of this project, we researched on what already exists in the market. Two of the most popular tools we found are Market Watch and Wallet Investor.

MarketWatch is an American financial information website that provides business news, analysis, and stock market data [1]. It provides a large amount of information, from real time stock prices information, to articles and watchlists, but they only focus on providing information and don't make any predictions on the data they provide.

Wallet Investor is a tool that offers cryptocurrency, stocks, forex, fund, and commodity price predictions by Machine Learning. Wallet Investor's cryptocurrency and other forecasts are based on changes in the exchange rates, trade volumes, volatilities of the past period, and other important economic aspects. The accuracy of the prediction depends on the quantity and the quality of the data, so it is also difficult to anticipate anything in the case of newer cryptocurrencies. [2]. However, there is no information on what Machine Learning models they are using to build their predictions.

By analyzing these two tools and other similar ones, we came to the conclusion that their services cover different parts of our tool, but none of them provides them all. This was the main reason why we added several Machine Learning models for the user to build and customize, not only to see their results as Wallet Investor does. It is worth mentioning, that as a university software project, STOCK should not be seen as a competitor of any present tool and that

it has its own limitations in terms of the amount of data and the companies the user can analyze, but this can also be seen as a potential expansion in the future. Our main goal is to give the user the opportunity to have everything in one place and easily get a grasp of the stock market.

Additionally, we found a tool for financial data visualization and analysis called Wallmine. They provide various tools for displaying financial information, more precisely stocks, forex and cryptocurrencies that can help financial exerts making smarter financial decisions [3]. Moreover, they offer information for most of the stock markets same as Wallet Investor and Market Watch. However, the tool lacks Machine Learning capabilities and does not focus on stock price predictions while one of the STOCKs main focuses is exactly that.

2.2. Use Cases

When we first defined the idea of the STOCK application, we had three types of users in mind:

- Unexperienced users, which includes users that don't have any knowledge of the stock market or Machine Learning, but they still want to get some information with easy indicators on how current events affect the prices.

- Casual/Standard users, which includes users that have limited knowledge of the stock market and Machine Learning, but not enough technical skills to be able to build models themselves.

- Experienced users, which includes users that have more advanced knowledge of the stock market and Machine Learning models building and customization, but they lack the ability to build them without technical skills, and/or need different types of information and analytical visualizations in one place.

While we were working on building the tool, we asked 5 students from our faculty to use it and give us their review on it, specifically students attending the Introduction to Machine Learning course. The Analysis view in STOCK, described in section 3.4.7., supports a major part of this course, such as Regression Machine Learning models, Decision Tree, Random Forest and Recurrent Neural Networks. The main feedback we received was that it is a very easily usable tool even for people that don't have much knowledge regarding the stock market or Machine Learning models. For these users which are currently studying Machine Learning, it simplifies the process of creating models that have different levels of complexity from simple linear models to recurrent neural networks.

We found that it was particularly useful in the context of the latest events with the COVID-19 (Coronavirus) which is an evolving situation and everyday there is impact on the prices of stock market companies. For example, Apple, Facebook, Amazon, Microsoft and Google lost more than $238 billion as reported on February 24th, 2020. Articles related to this were also retrieved and sentiment analyzed in our system giving indicators if they were positive, negative or neutral and then reflected in the charts of current week's prices as shown in the two Figure 1.

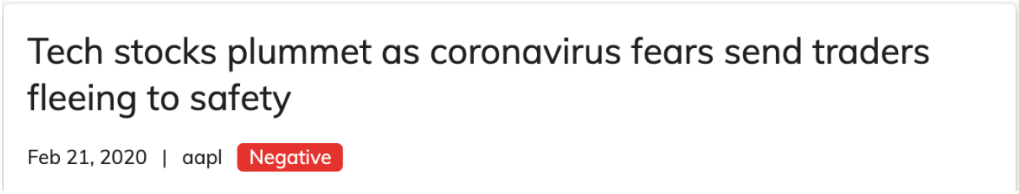

Figure 1: Analyzed article of coronavirus

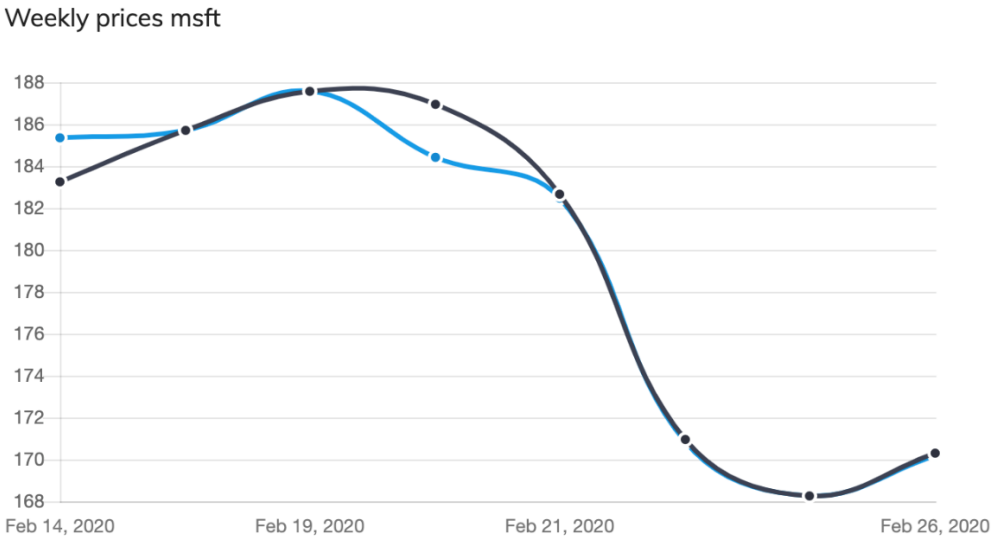

Figure 2: Weekly opening and closing prices affected by coronavirus

2.3. Project Schedule

Our work on this project is the result of 6 stages, from the initial idea around December 2018 to the final delivery, which are described in more detail in the next sections.

2.3.1. Stage 1

This stage describes the work we were doing from December 2018 to March 2019. It is worth mentioning that during this time, we had another member in our team, Foivos Gkoulis, but for personal reasons he had to leave in

January and from January 2019 to May 2019 we were only 3 members working on the project while looking for another one.

On our initial idea for this project we wanted to build a tool where users could see how the stock prices of the companies in the stock market were being affected by the articles or tweets written about them. However, after several discussions with our supervisor, we realized that we could extend it even more by building Machine Learning models to predict the prices, but the user would only see their results and would not be able to customize them. After we finalized our idea, we sent the proposal to the committee, and by 1st of April it got approved and we were ready to start working on the detailed specification. Also, while working on the proposal, the Authentication and Database modules were also being worked on and prepared for further development.

2.3.2. Stage 2

This stage describes the work we were doing from April 2019 to May 2019. During this time, our idea got defined and clearer and we were working on the detailed specification after the proposal approval and confirming the official start date to 1st of June 2019. In May, another member joined our team, Mahran Emeiri, and started contributing to the project from the detailed specification to its finalization.

2.3.3. Stage 3

This stage describes the work we were doing from June 2019 to July 2019. During this time, we already had some of the modules in place, but we were more focused on the detailed specification and the defense with the committee. We also prepared a mock version of our application so that there would be no confusion regarding the functionalities of STOCK.

2.3.4. Stage 4

This stage describes the work we were doing from August 2019 to October 2019. We submitted the detailed specification by 1st of August and we continued working on building the tool, specifically the Data Provider Services, Machine Learning module and Client application.

2.3.5. Stage 5

This stage describes the work we were doing from October 2019 to December 2019. In the beginning of October, we had the detailed specification defense with the committee, and more than discussing in detail the project, also offered new opportunities on how to extend our tool. Rather than building Machine Learning models and showing the results to the user, we decided to offer the user the freedom to build and customize the Machine Learning models according to their preferences and what they are more interested in tweaking. This added the Analysis page to STOCK application, where the user can choose from a list of models, can customize

all the corresponding parameters, see the results, compare, copy them and more. By this time, we already had a working web application, we were building the first machine learning models and the back end was already supporting most of the promised features.

2.3.6. Stage 6

This stage describes the work we were doing from December 2019 to February 2020. By the end of December, the main functionalities of the application were already working and ready to be tested. Since the coding was almost done, we were mainly focused on testing the application from a technical point of view as well as a user point of view and also documenting all the work we did in the past months.

3. Implementation Overview

This section is a high-level overview of the system, its main components, their responsibilities and how they communicate with each other.

3.1. System Architecture

The system is divided into smaller implementation modules as depicted in Figure 3.

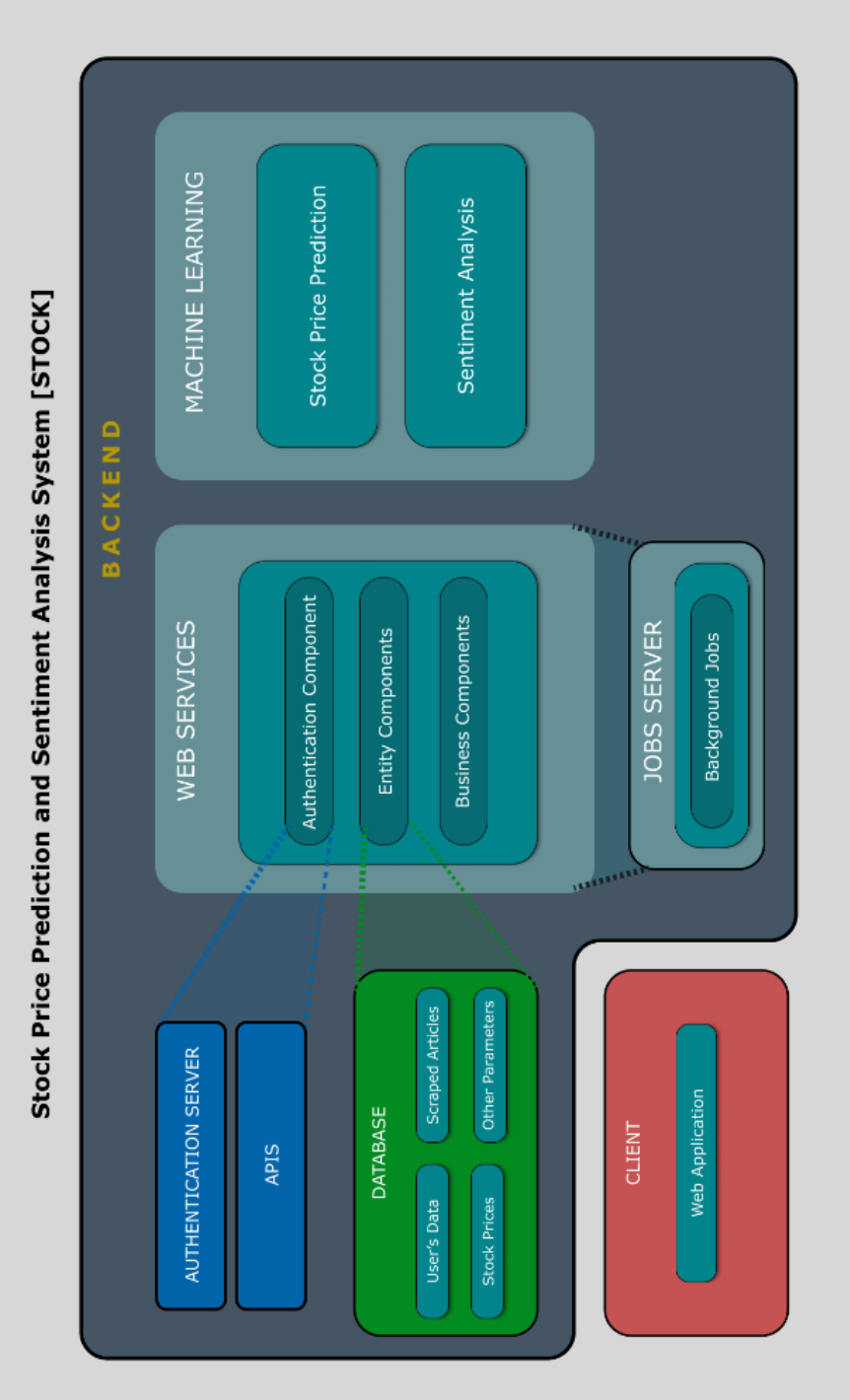

Figure 3: Decomposition View

- Client Module (Front End)

A web application that allows users to interact with STOCK to get the latest information about the stock market and build Machine Learning models to predict future prices.

- Authentication Server Module

This module is responsible for authentication and authorization of users in order to access resources and functionalities.

- Database Module

This module is used to store users' data, financial articles, stock prices, companies' information, result of default and user created machine learning models.

- Machine Learning Module

This module is the core part of the application which will be responsible for creating a default prediction model for company profile page and it allows users to create custom prediction models.

- Web Services Module

This module is a middleware responsible for dataflow and business logic between other modules.

3.2. Data Flow

This section describes how all the parts of the system communicate with each other to send and receive data.

For example, when the user creates a custom model with tuned parameters, the request goes to backend. Backend validates the requests and stores the parameters in the database. After that backend schedules a job to builds the model and sends the parameters to machine learning module. Machine learning module creates the model and sends back the results to the backend. After the results are stored in the database they can be retrieved by the frontend upon request.

Figure 4: High Level Data Flow

4. Front End Module

This section is intended for parties who would like to have more information regarding the implementation and code structure of the STOCK client application.

4.1. Installation and Prerequisites

To install and use this software, you should first install the following technologies with the corresponding versions.

- yarn v1.17.x [4]
- nodejs v10.15.x [5]
- docker v19.x [6][1]

To run this application locally without Docker, navigate to the source folder where you have saved the project and run the following commands.

```
$ npm install
$ npm run start
```

Snippet 1: Running the Application

Now the application will be running in localhost:3000 on your browser. To change the default port, you should go to the package.json file in the main folder of the project, in the "scripts" section locate "start" and change the value "export PORT=3000" to "export PORT=[INTENDED_PORT]" on Mac or Linux machines or to "set PORT=[INTENDED_PORT]" on Windows machine. Then run the previous commands again and navigate to localhost:[INTEDED_PORT] on your browser.

You can also access a live version with the latest changes at this URL: https://www.predictfuturestocks.com/

To run the application using Docker, run the following commands in the command line:

```
$ docker build -t stocks-front .
$ docker run -p 80:80 stocks-front
```

Snippet 2: Running the Application with Docker

[1] Install only if you want to run the application using Docker.

To run the application on a different port, run:

```
$ docker run -p [INTENDED_PORT]:80 stocks-front
```

Snippet 3: Running the Application on a Different Port

4.2. Project Structure

```
├── src # root project folder
    ├── @fuse # theme components
    ├── @lodash # theme library for object manipulation
    ├── app # main source code folder
        ├── api # accessing data from backend
        ├── functions # utility shared functions
        ├── fuse-configs # basic configuration for the theme
        ├── fuse-layouts # layout configuration for the theme
        ├── locales # supported languages
        ├── main # custom components
        ├── store # data manipulation
        ├── App.js # main component of the app
        ├── AppContext.js # creation of React Context
        ├── locale.js # locale configuration
        ├── validationService.js # data validation functions
    ├── styles # main component styling
    ├── history.js # creating browser history for routing
    ├── index.js # main entry point of the application
    ├── react-chartjs-2-defaults.js # react chartjs library support
    ├── react-table-defaults.js # react table library support
├── .gitignore # git config file to remove files from commit
├── package-lock.json # automatically generated file by npm
├── package.json # project development dependecies
├── README.md # description and general instructions for the project
├── tailwind.js # css framework
├── purge-tailwindcss.js # automatically generated files by tailwind
├── yarn-error.log # log error by yarn library
├── yarn.lock # automatically generated file by yarn
```

Snippet 4: Project Structure

4.3. Dependencies

1. *React v16.8.3*
   The main library of the web application is React which is a JavaScript library used for building user interfaces. It is maintained by Facebook and a community of individual developers and companies. [7] We used React to create the project as a Single Page Application (SPA).

2. *Material UI v3.9.2*
   Material UI is a library that provides reusable component for React development. [8] It implements the Google Material Design guidelines.

3. *Chartjs v2.7.3*
   Chartjs is a free open-source JavaScript library for data visualization. In this project it is used to visualize information such as stocks prices in the form of charts. [8]

4. *Fuse Theme*
   Fuse is a React & Material Design admin template. We used it a skeleton for the project. [9]

5. *Redux v4.0.1*
   Redux is an open source JavaScript library used for managing application state. [10] We used it for managing the state of the components and handling data manipulation throughout the application.

6. *React Redux v6.0.1*
   React Redux is a UI "binding" library to tie Redux together with the UI framework. React Redux is the official UI binding library for React [11], which is why we used it in the application.

7. *React Thunk*
   Redux Thunk is a middleware for Redux, to extend the Redux store abilities and allows us to write async logic that interacts with the store. [12]

4.4. Data Flow

To display the information necessary, the UI communicates with the server through an API. However, to serve the information properly in the UI and update the state and changes accordingly, Redux is used.

The logic used is the same for all components, so for simplicity only ArticleDetails component is explained in Snippet 5.

```
├── src
    ├── app
        ├── api
        # articles api calls
            ├── articles.api.js
            ...
        ├── store
            ├── actions
                ├── custom
                # store actions for articles
                    ├── articles.actions.js
                    ...
            ├── reducers
                ├── custom
                # store reducers for articles
                    ├── articles.reducer.js
        ...
```

Snippet 5: Article Details API calls and store actions structure

As mentioned in the Components chapter in section 2.4, in the Article Details component the user is able to see a preview of related articles. In this page, user can see details of a selected article, content and indicator (Positive/Negative/Neutral). This page doesn't need authentication.

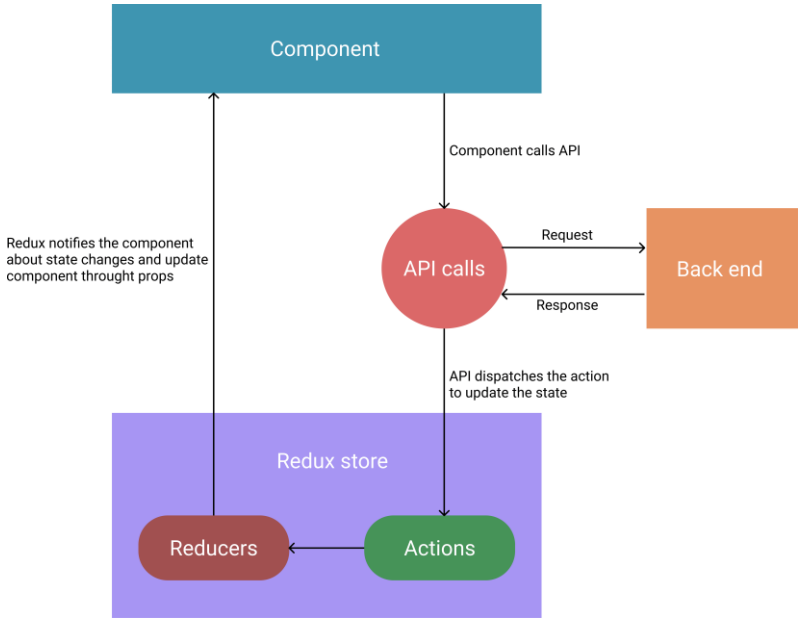

Figure 5: Communication within components

When we navigate to the Article Details page, upon component mounting getSingleArticle is triggered to get the data from the server. On success it dispatches an action to update the state. After the state is updated, we receive singleArticleDetails through the props to the component itself.

4.5. Components

This is an overview of the components, their routes and how to add new pages. In the following chapters, we will go through each component in more details, this is just a guide on how to add new components to the application in the future.

To add a new component, first we create the corresponding component folder under /main following the structure Snippet 6.

```
import React, { Component } from "react";
        import { FusePageSimple } from "@fuse";

        class NewComponent extends Component {
        ...
              render() {
              const { classes } = this.props;
              return (
              <FusePageSimple
                 content={
                 // content of the page
                 }
              />
              );
           }
```

```
        }
        function mapStateToProps({ custom }) {
              return { link the component to state store };
        }

        const mapDispatchToProps = dispatch =>
              bindActionCreators({ link the component to action },
                    dispatch
              );
        export default connect(mapStateToProps,
mapDispatchToProps)(NewComponent);
```

Snippet 6: Component structure

The next step is configuring the route to the newly created component in NewComponentConfig.js file as Snippet 7:

```
        import { FuseLoadable } from "@fuse";

        export const NewComponentConfig = {
        settings: {
            layout: {
            config: {}
            }
        },
        routes: [
            {
            path: "/newcomponent-path",
            component: FuseLoadable({
                loader: () => import("./NewComponent")
            })
            }
        ]
        };
```

Snippet 7: Route configuring

Then, in the /app/fuse-config/routesConfig.js we import the file we created:

```
  import { NewComponentConfig } from "route-
to/newcomponent/NewComponentConfig.js";
```

Snippet 8: Import component configuration

and add the NewComponentConfig to the routes object:

```
  const routeConfigs = [
       ...
       NewComponentConfig
  ]
```

Snippet 9: Add configuration to routes

The last step is to add in the /app/fuse-configs/navigationConfig.js the Snippet 10:

```
const navigationConfig = [{
      id: "applications",
      title: "Applications",
      ...
      children: [ ... {
          id: "newcomponent",
          title: "newComponent",
          type: "item",
          url: "/newcomponent-path",
          icon: "icon",
          showOnLogin: true | false | null
              // decide if the menu item is available
              // for all users or only authenticated ones
              // true : for logged in
              // false: for not logged in
              // null: for both
                  },]}
 ]
```

Snippet 10: Add new component to side menu

4.5.1. Dashboard

The dashboard is the main entry point of the application. It is available without authentication.

Structure

```
├── src
    ├── app
        ├── main
            ├── dashboard
                ├── widgets
                    # class component to display latest stocks
                    ├── Table.js
                # basic configuration of dashboard routes and settings
                ├── DashboardConfig.js
             # class component used as a container for the dashboard page
                ├── Dashboard.js
```

Snippet 11: Dashboard folder structure

Dashboard.js

This component doesn't have any custom methods, it makes use of the React Component Lifecycle.

Property	Description
articles	Array of articles
stocksList	Array of the latest stocks list
chartList	Array of latest stocks for a specific company to be displayed in a chart.
chartCurrentCompany	Selected company to be shown the chart for.
locale	Page localisation
isLoggedIn	Indicator if the current user is logged in

Table 1: Dashboard component properties

Table.js

This is a subcomponent used in Dashboard to display the latest stocks for all companies.

Property	Description
data	Stocks list coming from the parent component (Dashboard)

Table 2: Component Properties

4.5.2. Companies

This page is not available without authentication to the platform. It allows users to manage their favorite companies.

Structure

```
├── src
    ├── app
        ├── main
            ├── companies
                ├── widgets
                     # class component to display list of companies
                     ├── CompaniesList.js
                     # class component item used by companies list
                     ├── CompanyItem.js
                # basic configuration of companies' routes and settings
                ├── CompaniesConfig.js
            # class component used as a container for the companies page
                ├── Companies.js
```

Snippet 12: Companies folder structure

Companies.js

This component has custom methods and it also makes use of the React Component Lifecycle.

Property	Description
companies	List of all available companies in the system.
userSettings	Logged in user settings, including the list of their favourite companies.
locale	Page localisation
isLoggedIn	Indicator if the current user is logged in

Table 3: Companies component properties

CompaniesList.js

This is a subcomponent used in Companies to display the list of all companies.

Method	Description
setFilter	Method used for real time search of companies in the list.
manageCompanySelect	Method for selection of companies in the list.
manageCompanies	Method for handling add/delete of companies list.

Table 4: CompaniesList component methods

Property	Description
userSettings	Logged in user settings, including the list of their favourite companies.

Table 5: CompaniesList component properties

CompanyItem.js

This is a subcomponent used in CompaniesList to display a single company.

Methods	Description
loadCompany	Redirecting the user to the selected company page.

Table 6: CompanyItemt component methods

Property	Description
company	Object with information about selected company

Table 7: CompanyItem component properties

4.5.3. Company Details

This page is available without authentication to the platform. It allows users to see the details of a specific company.

Structure

```
├── src
    ├── app
        ├── main
            ├── company-details
                ├── widgets
                     # class component to display charts
                     ├── BarChart.js
                # class component used to show information about the company
                # (name, description, founding etc.)
                     ├── Details.js
                # class component item to show the historical data of stocks
                     ├── Table.js
                # basic configuration of company details routes and settings
                ├── CompanyDetailsConfig.js
          # class component used as a container for the company details page
                ├── CompanyDetails.js
```

Snippet 13: Company details folder structure

CompanyDetails.js

This component doesn't have any custom methods, it makes use of the React Component Lifecycle.

Property	Description
stocksList	List of historical stock data
companyDetails	Information about the company (description, name etc.)
locale	Page localisation
isLoggedIn	Indicator if the current user is log

Table 8: CompanyDetails component properties

BarChart.js

This is a subcomponent used in Companies to display charts in the company details page.

Property	Description
title	Title of the widget (chart)
data	Data that will be rendered by the widget (chart)

Table 9: BarChart component properties

Details.js

This is a subcomponent used in CompaniesList to display information about the company such as official name, founding date, CEO, description etc.

Method	Description
handleFilter	Method used to filter the historical data according to a date range set by the user.
parseRows	Converts the response object into the structure needed for display in UI.

Table 10: Details component methods

Property	Description
locale	Page localisation.

Table 11: Details component properties

4.5.4. Articles Details

From the Dashboard page, user is able to see a preview of related articles. In this page, user can see details of a selected article, content and indicator (Pos/Neg/Neu). This page doesn't need login.

Structure

```
├── src
    ├── app
        ├── main
            ├── article-details
          # class component used as a container for article details page
                ├── ArticleDetails.js
           # basic configuration of articles details routes and settings
                ├── ArticleDetailsConfig.js
```

Snippet 14: Article Details folder structure

ArticleDetails.js

This component doesn't have any custom methods, it makes use of the React Component Lifecycle.

Property	Description
articleDetails	Current article details
articles	Array of articles related to the current company
locale	Page localisation
isLoggedIn	Indicator if the current user is logged in

Table 12: ArticleDetails component properties

4.5.5. History

This page is not available without authentication to the platform. It allows users to see their search history within the platform.

Structure

```
├── src
    ├── app
        ├── main
            ├── history
                ├── widgets
                    # class component item used to show search history items
                     ├── SearchHistory.js
                # basic configuration of history routes and settings
                ├── HistoryConfig.js
                # class component used as a container for the history page
                ├── History.js
```

Snippet 15: History folder structure

History.js

This component has no custom methods, but it makes use of the React Component Lifecycle.

Property	Description
historyList	List of the users search history items
locale	Page localisation
isLoggedIn	Indicator if the current user is logged in

Table 13: History component properties

SearchHistory.js

This is a subcomponent used in History to display the searched items.

Method	Description
loadCompany	Method used to navigate to the company details page from the list.

Table 14: SearchHistory component methods

Property	Description
data	The searched items to be displayed

Table 15: SearchHistory component properties

4.5.6. Profile

This page is available only with authentication to the platform. It allows users to see their profile information and edit it.

Structure

```
├── src
    ├── app
        ├── main
            ├── profile
                ├── widgets
            # class component item used to show contact information of user
                    ├── ContactInfo.js
            # class component item used to show account information of user
                    ├── AccountInfo.js
                # basic configuration of profile routes and settings
                ├── ProfileConfig.js
                # class component used as a container for the profile page
                ├── Profile.js
```

Snippet 16: Profile folder structure

Profile.js

This component has no custom methods but it makes use of the React Component Lifecycle.

Property	Description
locale	Page localisation
isLoggedIn	Indicator if the current user is logged in

Table 16: Profile component properties

ContactInfo.js

This is a subcomponent used in Profile to display the contact information of the user.

Method	Description
updateUserInfo	Method used to update user information.

Table 17: ContactInfo component methods

Property	Description
userInfo	Object that holds the user information details

Table 18: ContactInfo component properties

AccountInfo.js

This is a subcomponent used in Profile to display user information related to the account (update password, delete account).

Method	Description
validatePreviousPassword	Method to validate user's previous password if it matches, right pattern etc.
validateNewPassword	Method to validate user's new password if it matches the required pattern
validateConfirmPassword	Method to validate that user wrote the same password to confirm it.
updatePassword	Method to handle the password change request.
canBeSubmitted	Method to check if user can submit the form to change password
showDeleteModal	Showing a confirmation pop up to confirm that user wants to delete the password.

Table 19: AccountInfo component methods

Property	Description
modalLocale	Widget localisation

Table 20: AccountInfo component properties

4.5.7. Analysis

This page is not available without authentication to the platform. It allows users to create, view and compare their machine learning models.

Structure

```
├── src
    ├── app
        ├── main
            ├── analysis
                # components and widgets used to create a new model
                ├── add-model
                # components and widgets used to compare models
                ├── compare-models
                # components and widgets used to view a specific model
                ├── view-model
                # widgets used in analysis view of all models created
                ├── widgets
                     # component that provides filtering of user's models
                     ├── Filter.js
                     # component that lists all models of the user
                     ├── Table.js
                # class component used as a container for the analysis page
                ├── Analysis.js
                # basic configuration of analysis routes and settings
                ├── AnalysisConfig.js
```

Snippet 17: Analysis folder structure

Analysis.js
This component has no custom methods, but it makes use of the React Component Lifecycle.

Property	Description
userModels	List of the users created models
locale	Page localisation
isLoggedIn	Indicator if the current user is logged in

Table 21: Analysis component properties

Filter.js
This is a subcomponent used in Analysis to filter the list of created models based on model company, model type, creation date and status. The filtering is done on-the-fly when the user selects the company/type/date/status using customhandlers.

Method	Description
clearFilters	Method used to reset all user filters.

Table 22: Filter component methods

Property	Description
companies	List of all available companies
models	List of all model types
userModelsFilter	Object used for filtering the models list

Table 23: Filter component properties

Table.js
This is a subcomponent used in Analysis to list all created models of the user based on user's filters, and giving the user the options of deleting and cloning a model.

Method	Description
showDeleteModal	Modal to confirm that user wants to delete a model.

Table 24: Table component methods

Property	Description
userModelsFilter	Object used for filtering the models list
data	List of models to be displayed
locale	Page localisation

Table 25: Table component properties

4.5.8. Add model

This page is not available without authentication to the platform. It allows users to create a new model.

Structure

```
├── src
    ├── app
        ├── main
            ├── analysis
                ├── add-model
                    ├── widgets
                        # component used to select the dataset and model
                        # title
                        ├── DataControl.js
                        # component used to select features for the model
                        ├── FeaturesControl.js
                        # component used to select the feature to be
                        # predicted
                        ├── FeatureToPredict.js
                        # component used to select the model type and its
                           # hyperparameters
                        ├── ModelControl.js
                    # class component used as a container for the add model
                    # page
                    ├── AddModel.js
                    # basic configuration of add model routes and settings
                    ├── AddModelConfig.js
```

Snippet 18: Add model folder structure

AddModel.js
This component has custom methods and it makes use of the React Component Lifecycle.

Method	Description
saveModel	Saves the user model and sends it to the backend for building.

Table 26: Add model component methods

Property	Description
createdModel	Object used to save the user selections for the model
hyperParams	Object used to dynamically load the hyperParameters of each model type
locale	Page localisation
isLoggedIn	Indicator if the current user is logged in

Table 27: Add model component properties

DataControl.js
This is a subcomponent used in Add Model to select the dataset to be used, i.e. company and add a title for the model.

Property	Description
companies	List of all available companies
createdModel	Object used to save the user selections for the model
locale	Page localisation

Table 28: DataControl component properties

FeaturesControl.js
This is a subcomponent used in Add Model to select the features to be used in the model building.

Property	Description
createdModel	Object used to save the user selections for the mod
locale	Page localisation

Table 29: FeaturesControl component properties

FeatureToPredict.js
This is a subcomponent used in Add Model to select the feature the user wants to predict.

Property	Description
createdModel	Object used to save the user selections for the mod

locale	Page localisation

Table 30: FeatureToPredict component properties

ModelControl.js
This is a subcomponent used in Add Model to display the model type selected and its features.

Property	Description
models	List of all available models
createdModel	Object used to save the user selections for the model
hyperParams	Object used to dynamically load the hyperParameters of each model type
locale	Page localisation

Table 31: ModelControl component properties

4.5.9. Compare models

This page is not available without authentication to the platform. It allows users to compare two selected models.

Structure

```
├── src
    ├── app
        ├── main
            ├── analysis
                ├── compare-models
                    # class component used as a container for the compare
                    # model page
                    ├── CompareModels.js
                    # basic configuration of compare model routes and
                    # settings
                    ├── CompareModelConfig.js
```

Snippet 19: Compare models folder structure

CompareModels.js
This component has no custom methods but it makes use of the React Component Lifecycle. It shares the widgets in the `/widgets` subfolder of View Model component described in section 2.7.4.

Property	Description
isLoggedIn	Indicator if the current user is logged in.
modelById	Source model of the compare
comparedModel	Target model of the compare
locale	Page localisation

Table 32: CompareModels component properties

4.5.10. View model
This page is not available without authentication to the platform. It allows users to view a model in detail.

Structure

```
├── src
    ├── app
        ├── main
            ├── analysis
                ├── view-model
                    ├── widgets
                        # component used to show the selected dataset
                        ├── DataControl.js
                        # component used to show the selected features for
                        # the model
                        ├── FeaturesControl.js
                        # component used to show the selected feature to be
                        # predicted
                        ├── FeatureToPredict.js
                        # component used to show the selected model type and
                        # its hyperparameters
                        ├── ModelControl.js
                        # component used to show title summary, status and
                        # actions (compare, export)
                        ├── Header.js
                        # component used to show the model results
                        ├── Predictions.js
                    # class component used as a container for the view model
                    # page
                    ├── ViewModel.js
                    # basic configuration of view model routes and settings
                    ├── ViewModelConfig.js
```

Snippet 20: View Model folder structure

ViewModel.js
This component has no custom methods but it makes use of the React Component Lifecycle.

Property	Description
modelById	The model the user selected for display
locale	Page localisation
isLoggedIn	Indicator if the current user is logged in

Table 33: ViewModel component properties

DataControl.js
This is a subcomponent used in View Model to show the dataset that the user selected to be used in the model building, e.g. company.

Property	Description
companies	List of all available companies
company	The selected company for the model
locale	Page localisation

Table 34: DataControl component properties

FeaturesControl.js
This is a subcomponent used in View Model to display the selected features of the model.

Property	Description
features	List of selected features
locale	Page localisation

Table 35: FeaturesControl component properties

FeatureToPredict.js
This is a subcomponent used in View Model to display the selected feature to be predicted.

Property	Description
value	Selected feature
locale	Page localisation

Table 36: FeatureToPredict component properties

Header.js
This is a subcomponent used in View Model to display the model title, status and actions such as compare and export the model to PDF.

Method	Description
export	Exports the model to PDF
compare	Change route to model comparison view

Table 37: Header component methods

Property	Description
userModels	List of all user models
name	Title of the selected model
state	State of the selected model (Created, Pending, Failed)
locale	Pagelocalisation

Table 38: Header component properties

ModelControl.js
This is a subcomponent used in View Model to display the model type selected and its features.

Property	Description
models	List of all available models
type	Type of the selected model
parameters	Selected model hyperParameters
locale	Page localisation

Table 39: ModelControl component properties

Predictions.js
This is a subcomponent used in View Model to display the model results.

Property	Description
result	Results of the model
locale	Page localisation

Table 40: Predictions component properties

4.6. Authentication

This is an overview of the authentication widgets and logic.

4.6.1. Structure

```
├── src
    ├── app
        ├── main
            ├── auth
                ├── forgot-password
                     # diplays the forgot password form
                     ├── ForgotPasswordPage.js
                     # basic configuration of forgot password routes and
                     # settings
                     ├── ForgotPasswordPageConfig.js
                ├── login
                     # diplays the login form
                     ├── LoginPage.js
                     # basic configuration of login routes and settings
                     ├── LoginPageConfig.js
                ├── mail-confirm
                     # diplays the mail confirmation form
                     ├── MailConfirmPage.js
                    # basic configuration of mail confirmation routes and
                     # settings
                     ├── MailConfirmPageConfig.js
                ├── register
                      # diplays the registration form
                     ├── RegisterPage.js
                # basic configuration of registration routes and settings
                     ├── RegisterPageConfig.js
                ├── reset-password
                      # diplays the reset password form
                     ├── ResetPasswordPage.js
                     # basic configuration of reset password routes and
                     # settings
                     ├── ResetPasswordPageConfig.js
```

Snippet 21: Authentication folder structure

4.6.2. ForgotPasswordPage.js

This is a container for the forgot password form.

Method	Description
canBeSubmitted	Method to check if the form is correctly filled and can be submitted
validateUsername	Method to check if the username matches the pattern required.

Table 41: ForgotPasswordPage component methods

Property	Description
locale	Page localisation.
isLoggedIn	Indicator if the user is logged in.

Table 42: ForgotPasswordPage component properties

4.6.3. LoginPage.js

This is a container for the login form.

Method	Description
canBeSubmitted	Method to check if the form is correctly filled and can be submitted
validateUsername	Method to check if the username matches the pattern required.
validatePassword	Method to check if the password matches the pattern required.

Table 43: LoginPage component methods

Property	Description
locale	Page localisation.
isLoggedIn	Indicator if the user is logged in.

Table 44: LoginPage component properties

4.6.4. RegisterPage.js

This is a container for the registration form.

Method	Description
canBeSubmitted	Method to check if the form is correctly filled and can be submitted
validateConfirmPassword	Method to validate the password confirmation.
validateEmail	Method to validate email matches the pattern required.
validateUsername	Method to check if the username matches the pattern required.
validatePassword	Method to check if the password matches the pattern required.

Table 45: RegisterPage component methods

Property	Description
locale	Page localisation.
isLoggedIn	Indicator if the user is logged in.

Table 46: RegisterPage component properties

4.6.5. MailConfirmPage.js

This is a container for the mail confirmation form.

Method	Description
canBeSubmitted	Method to check if the form is correctly filled and can be submitted
validateConfirmationCode	Method to validate the pattern of the confirmation code sent to the user email.

Table 47: MailConfirmPage component methods

Property	Description
locale	Page localisation.
isLoggedIn	Indicator if the user is logged in.

Table 48: MailConfirmPage component properties

4.6.6. ResetPasswordPage.js

This is a container for the reset password form.

Method	Description
canBeSubmitted	Method to check if the form is correctly filled and can be submitted
validateConfirmPassword	Method to validate the password confirmation.
validateConfirmationCode	Method to validate the pattern of the confirmation code sent to the user email.
validateUsername	Method to check if the username matches the pattern required.
validatePassword	Method to check if the password matches the pattern required.

Table 49: ResetPasswordPage methods

Property	Description
locale	Page localisation.
isLoggedIn	Indicator if the user is logged in.

Table 50: ResetPasswordPage component properties

4.7. Shared widgets

An overview of widgets shared by multiple components in the project.

Structure

```
├── src
    ├── app
        ├── main
            ├── shared-widgets
                ├── articles
                     # diplays a list of articles
                     ├── Articles.js
                ├── breadcrumbs
                     # displays breadcrumbs to show the user the location
                      # within the platform
                     ├── Breadcrumbs.js
                ├── charts
                     # displays data in a chart
                     ├── Charts.js
                # custom loaders for all components
                ├── loaders
```

Snippet 22: Shared components folder structure

4.7.1. Articles.js

Class component to display a list of articles.

Property	Description
widgetTitle	Title of the articles widget
data	Object with the articles to be displayed.

Table 51: Articles component properties

4.7.2. Breadcrumbs.js

Class component to display the current navigation level.

Property	Description
locale	Page localisation
data	Object with the current navigation level within the platform.

Table 52: Breadcrumbs component properties

4.7.3. Charts.js

Class component to display different data objects into charts.

Property	Description
data	Object with the data to be displayed.
title	Title of the current chart.

Table 53: Charts component properties

4.7.4. Loaders

On this folder, there are static components to display a loading pattern for different components of the project. For examples, there is ChartLoader.js which displays a loading pattern to imitate the chart widget. There are custom loaders depending on the type of the widget, including TableLoader, SmallChartLoader, CompanyDetailsLoader, ArticlesBodyLoader etc., but they only contain HTML markup and CSS styling to imitate the corresponding widget, so the code is not displayed here.

4.8. Data manipulation logic

This is an overview of the data manipulation logic and back end communication for all components with an example for the SingleArticle page. The other components follow the same logic and workflow.

4.8.1. API Call

To communicate with the back end we are using the axios [9] library which is a promise based HTTP client for the browser. For easier accessing, we defined an instance with the base URL and other common configurations for the requests in Snippet 23.

```
const instance = axios.create({
      baseURL: "https://www.predictfuturestocks.com/backend/api/",
      headers: {'language': 'en'}
  });
```

Snippet 23: axios instance

This instance is then used by all requests to the API simply by importing it where necessary.

4.8.2. articles.api.js

This file contains all the methods responsible for communicating with the API to get data about articles. getSingleArticle calls the /article endpoint and on success dispatches an action to send back the data to the ui and an action to indicate that the process is not loading anymore for better user experience.

```
import instance from "./axios.instance";
import { articlesSingle, articlesSingleLoading } from"../store/actions";
/**
 * get details of a single artcile
 * @param id the id of the wanted article
 */
export const getSingleArticle = id => {
   return dispatch => {
                dispatch(articlesSingleLoading(true));
                instance
                .get("articles", {
                params: {
                id
                }
                })
                .then(res => {
                dispatch(articlesSingle(res.data));
                dispatch(articlesSingleLoading(false));
                })
                .catch(er => {
                dispatch(articlesSingleLoading(false));
                });
   };
};
```

Snippet 24: getSingleArticle API call

4.8.3. articles.actions.js

We have defined two actions to manipulate the state of the single article page, ARTICLES_SINGLE and ARTICLES_SINGLE_LOADING to update the state accordingly.

```
export const ARTICLES_SINGLE = "[ARTICLES] SINGLE";
export const ARTICLES_SINGLE_LOADING = "[ARTICLES] SINGLE LOADING";
export function articlesSingle(articleDetails) {
    return {
        type: ARTICLES_SINGLE,
        articleDetails
    };
}
export function articlesSingleLoading(loading) {
    return {
        type: ARTICLES_SINGLE_LOADING,
        loading
    };
}
```

Snippet 25: Store actions to get a single article

4.8.4. articles.reducers.js

The reducer logic is responsible for handling the actions being dispatched from the API call and updating the state of the component.

```
import * as Actions from "app/store/actions/custom";

const initialState = {
    ...
    articleDetails: null,
    singleArticleLoading: true
};
const handle = function (state = initialState, action) {
    switch (action.type) {

        ...

        case Actions.ARTICLES_SINGLE: {
            return {
                ...state,
                articleDetails: { ...action.articleDetails }
            };
        }
        case Actions.ARTICLES_SINGLE_LOADING: {
            return {
                ...state,
                singleArticleLoading: action.loading
            };
        }
        default: {
            return state;
        }
    }
};
```

Snippet 26: Store reducers to get a single article

All other components follow the same logic and structure, depending on the types of actions for each component.

4.9. ValidationService.js

This is a validator class with several methods to make sure that the user types the expected input for the corresponding type of input field.

Method	Description
emailValidator	Validates user email with a RegEx pattern matching the usual email (examples@domain.com)

usernameValidator	Validates username with a RegEx pattern that requires between 5 and 15 uppercase and lowercase letters, numbers and underscore.
passwordValidator	Validates password with a RegEx pattern that requires at least one lowercase letter, one uppercase letter, one number and a total of at least 8 characters.
stringMatchValidator	Validates if two strings are identical.
confirmationCodeValidator	Validates if a string is exactly 6 characters long.

Table 54: ValidationService class methods

4.10. Localization

This is an overview of the localization of the whole project as well as a guide on how to add more languages or strings.

Structure

```
├── src
    ├── app
        ├── locales
            # the file that contains the english strings
            ├── en.js
            # the file that contains the czech strings
            ├── cz.js
        # the library configurations
        ├── locale.js
        ├── store
            ├── custom
                # language managment
                ├── locale.reducer.js
```

Snippet 27: Localisation components and files structure

4.10.1. Locale.js

For the localisation of the project in English and Czech language, we are using the react-localization [10] library in Snippet 27, using the en.js and cz.js files with the translated strings correspondingly for each language.

```
import LocalizedStrings from "react-localization";
  import en from "./locales/en";
  import cz from "./locales/cz";

  let locale = new LocalizedStrings({ en, cz });
  export default locale;
```

Snippet 28: Localisation library configuration

To add new languages, first we create a new file "[languagecode].js" under the locales folder, with all the strings for each section of the application, following the same structure. Then import the created file into locale.js as :

```
import [languagecode] from "./locales/[languagecode]";
```

Snippet 29: Importing new language strings

and update the localisation constructor of the react-localisation library.

```
let locale = new LocalizedStrings({ en, cz, [languagecode] });
```

Snippet 30: Update localization library constructor

The last step is updating the store reducer initial state in locale.reducer.js file:

```
const initialState = {
        ...
    localesCodes: [
        'en',
        'cz',
        [languagecode]
    ]
}
```

Snippet 31: Update locale store reducer

The strings files for each language follow a simple JSON object structure, with objects containing strings for each section. Snippet 32 is an example of the login page localised strings in Czech language, and all other languages being added should follow the same structure.

```
    const cz = {
        languages: {
            en: "English",
            cz: "Čeština",
            language: "Jazyk"
        },
        loginPage: {
            title: "PŘIHLÁSIT SE NA VÁŠ ÚČET",
            username: "Uživatelské jméno",
            password: "Heslo",
            rememberMe: "Zapamatuj si mě",
            forgotPassword: "Zapomenuté heslo?",
            login: "Přihlásit se",
            or: "NEBO",
            noAccount: "Nemáte účet?",
            register: "Vytvořit účet",
            invalidUsername: "Neplatné uživatelské jméno",
            invalidPassword: "Neplatné heslo"
        },
    }
```

Snippet 32: Login page localization strings example

4.11. Unit Testing

To perform unit testing of the application, the jest and Enzyme libraries are used. Jest is a JavaScript Testing Framework [11]. Enzyme is a JavaScript testing utility for React that makes it easier to test React Components' output, developed by AirBnb [12].

In the performed tests, we covered the shared widgets among major components, such as Articles displaying, Charts, Breadcrumbs, Tables etc.

4.11.1. Articles Testing

To write the tests we create a testing file on the same directory as the target component. In this case, the file is called Article.test.js. It is a library requirement to follow the same naming convention in all new test suites. The imports in the testing file look similar to the Snippet 33:

```
// configurations
import React from "react";
import Adapter from "enzyme-adapter-react-16";
import { configure, shallow } from "enzyme";

// component to be tested
import Articles from "./Articles";

// any children components to be asserted
import ArticleWidgetLoader from "../loaders/ArticleWidgetLoader";
import ListItem from "@material-ui/core/ListItem";
```

```
// linking jest to Enzyme
configure({ adapter: new Adapter() });
```

Snippet 33: Tests imports and configuration

The next step is start writing the test on the file. For simplicity, Snippet 34 shows only two examples, but it can be extended according to the component and expected results. Specifically, our Articles component expects two properties: loading which is an indicator to display a loading view until the data is loaded from the API and data which are the articles that we want to display. The first test checks if loading view is rendered when loading is false and there are no date to be displayed. The second test checks if the data is displayed when there is data but loading is false.

```
describe("<Articles />", () => {

  it("Should render loaders if loading is false but not data", () => {
    const wrapper = shallow(<Articles loading={false} />);
    expect(wrapper.find(ArticleWidgetLoader)).toHaveLength(1);
  });

  it("Should not render loaders if loading is false and there is data", ()
=> {
    const data = [
      {
        id: "54008c7e-9d83-4dbd-9ad3-52d82e2266b6",
        createdAt: "2020-02-29T16:00:00Z",
        companySymbol: "msft",
        title: "Fin24.com | Cloud computing: invisible, versatile and highly
             profitable",
        polarity: "Positive"
      }
    ];
    const wrapper = shallow(<Articles loading={false} data={data} />);
    expect(wrapper.find(ListItem)).toHaveLength(1);
  });
});
```

Snippet 34: First unit test

To run the test, the command below is used:

```
$ npm run test
```

Snippet 35: Running tests command

After running the tests, a summary of the results will be shown in the terminal, indicating passed or failed tests and possible reasons why they failed, if so. Below is an example output.

```
PASS  src/app/main/shared-widgets/articles/Articles.test.js
  <Articles />
    ✓ Should render loaders if loading is false but not data (8ms)
    ✓ Should not render loaders if loading is false and there is data (7ms)
Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        3.527s
Ran all test suites related to changed files.
```

Snippet 36: Test results

5. Back End Module

Backend currently runs in Azure AppService [13] at the endpoint: https://stockapi20200306052122.azurewebsites.net. APIs can be accessed at the path /api. Jobs can be accessed with SecretKey at the path /jobs. Backend is deployed to AppService through Visual Studio 2019 with zero configuration.

5.1. Structure

```
├── StockSolution # VISUAL STUDIO solution
    ├── Core # solution folder
        # projects
        ├── Stock.Application
        ├── Stock.Domain
        ├── Stock.Utilities
    ├── Infrastructure # solution folder
        # projects
        ├── Stock.Components
        ├── Stock.Jobs
        ├── Stock.Persistence
    ├── Presentation # solution folder
        # project
        ├── Stock.Api
    ├── Tests # solution folder
        # projects
        ├── Stock.IntegrationTests
        ├── Stock.UnitTests
```

Snippet 37: Back end solution folder structure

5.2. Architecture

The backend module is built using the Clean Architecture pattern [14] and is visualized in the Figure 6.

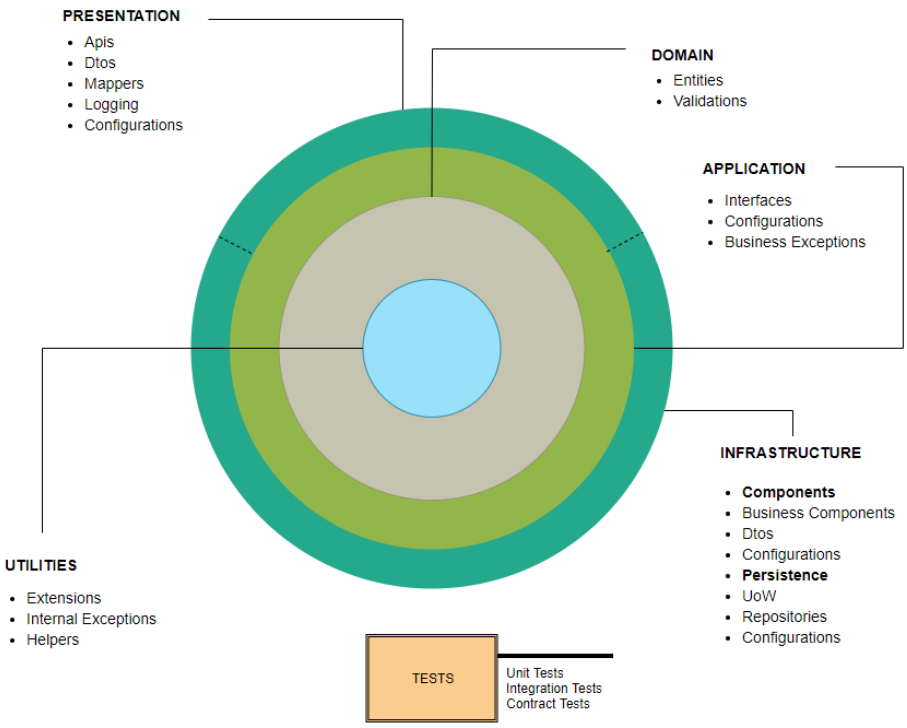

Figure 6: Backend architecture

The idea behind the Clean architecture is that one should depend on the contract/interface and not the implementation itself. Therefore, when the implementation changes, other parts of the system are not impacted. This is accomplished by Dependency Injection (DI) and Inversion of Control (IoC). Also, Domain is the main part of every system and is therefore put in the center and all other layers depend on it. Application is the layer which provides interfaces and contracts and the implementation layers Infrastructure, Persistence and Presentation depend on it and not on each other. Application layer depends on the Domain layer as well.

Figure 7 shows the architecture and structure of the backend from a high-level point of view. Each of these parts will be described in detail in the following sections.

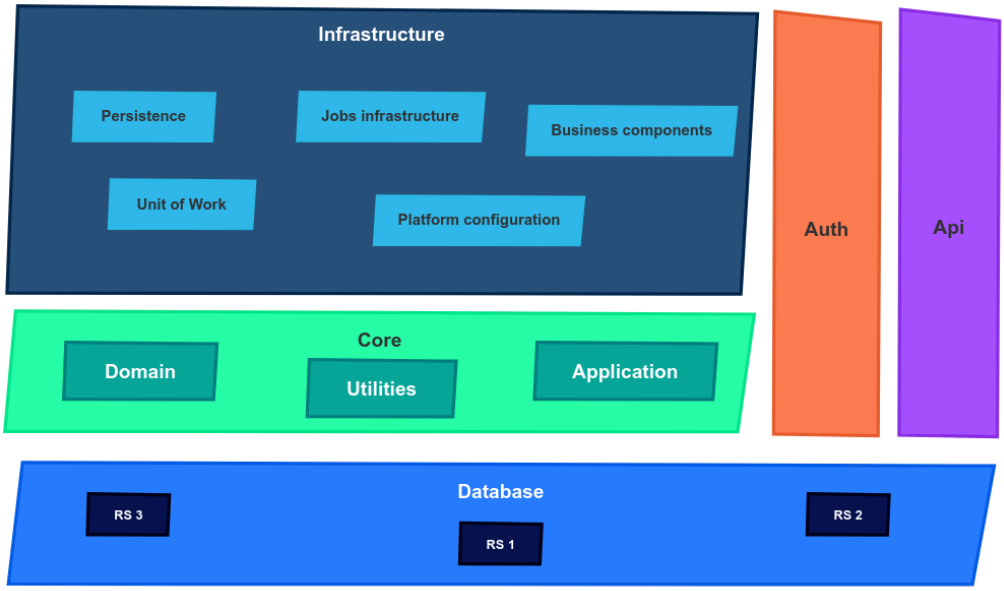

Figure 7: Backend units

Core is in the center of the backend, Presentation is the Representational State Transfer (REST) API [15] application where other units are libraries. In the Infrastructure, we can see that no units depend on each other as they depend only on the interfaces defined in the Core. Presentation does not directly depend on Infrastructure but rather on its interfaces defined in the Core and the link between them is because those implementations needs to be registered in the configuration of the REST API in order for the DI to find them during IoC.

5.3. Core

5.3.1. Domain

This section focuses on the Domain layer and entities it contains.

5.3.2. Structure

```
├── StockSolution # VISUAL STUDIO solution
    ├── Core # solution folder
        # projects
       ...
        ├── Stock.Domain
            ├── Entities
                # entity interfaces
                ├── Interfaces
                # Machine Leardning parameters
                ├── ML Model Parameters
                # Validation classes
                ├── Validations
                # Entities
                ├── Articles.cs
                ├── Company.cs
                ├── MLModel.cs
                ├── Stocks.cs
                ├── User.cs
          ├── Extensions
              # Helper methods
              ├── EntityExtensions.cs
       ...
```

Snippet 38: Domain folder structure

As mentioned in the Architecture section 4.2, Domain is the center of the whole system. It lies in the center of the Core and all other layers depend on it and contains entities (domain objects) of the whole system. It also defines business rules and validations for each entity.

```
public interface IEntity
{
    Guid Id { get; }

    DateTime CreatedAt { get; }

    Guid Version { get; }

    void Validate(IEntity entity);
}
```

Snippet 39: Base entity interface

As we can see from Snippet 39, IEntity has three properties that all entities need to implement. Id is the identifier of the object in the database. CreatedAt is the date of the creation of the object (varies between entities, some entities have it as date of the creation of the object in our system and some as the time of creation in external resources, e.g Stock entity has the date of that specific stock).Version is the internal property of that object which is used for optimistic concurrency. Domain comprises of these entities: Article, Company, Machine Learning (ML) Model, Stock, User. Each of these entities have specific validation classes. Snippet 40 shows the validation class definition of the ML Model.

```
internal sealed class MlModelValidator : AbstractValidator<MlModel>
```

Snippet 40: Validation class of ML Model

Moreover, IEntity's properties have only getters. This means that all entities are immutable, and their properties can be defined only once upon creation.

5.4. Application

This section focuses on the Application layer and interfaces that it defines.

5.4.1. Structure

```
├── StockSolution # VISUAL STUDIO solution
    ├── Core # solution folder
        # projects
       ...
        ├── Stock.Application
            #
            ├── Check
                # Business exception handler
                ├── Business.cs
            ├── Exceptions
                # Business exception
                ├── BusinessException.cs
            # collection of interfaces
            ├── Infrastructure
                ├── Components
                ├── Jobs
                ├── Persistence
       ...
```

Snippet 41: Application solution folder

Application is a business layer of the system. It provides interfaces and contracts to the rest of the system layers that need to implement them. By doing so, we allow applications to be developed in a component-oriented way because C# is Component Oriented Programming (COP) language. Each layer that needs to access functionality from another layer will depend on the interface that the Application layer provides, e.g. StocksComponent from

Infrastructure layer relies on IStockRepository interface from the Application layer rather than StocksRepository from the Persistence layer. Therefore, if there are changes to be made in the implementation of some parts of the system, the rest is not impacted as it relies on the interface. By doing this we are also able to test each module separately and therefore implement Test Driven Development (TDD) [16]. This is accomplished by DI and IoC. Application layer provides interfaces for the Infrastructure layer: Components, Persistence and Jobs.

Not all interfaces will be described in detail here but only the base one that shows the logic behind this design. The list of Persistence interfaces includes: IArticlesRepository, ICompaniesRepository, IMLModelsRepository, IStocksRepository, IUsersRepository and the base one IRepository which is shown in Snippet 42.

```
public interface IRepository<TEntity> where TEntity : class, IEntity
{
    Task<ITry<Guid>> TryAddAsync(
        IClientSessionHandle session,
        Func<Unit, TEntity> action
    );

    Task<ITry<IEnumerable<Guid>>> TryAddManyAsync(
        IClientSessionHandle session,
        Func<Unit, ICollection<TEntity>> action
    );

    Task<IOption<TEntity>> GetAsync(
        IClientSessionHandle session,
        Func<FilterDefinitionBuilder<TEntity>, FilterDefinition<TEntity>>
filterDefinitionPredicate,
        Func<ProjectionDefinitionBuilder<TEntity>,
ProjectionDefinition<TEntity>> projectionDefinitionPredicate = null
    );

    Task<IOption<IEnumerable<TEntity>>> GetManyAsync(
        IClientSessionHandle session,
        Func<FilterDefinitionBuilder<TEntity>, FilterDefinition<TEntity>>
filterDefinitionPredicate = null,
        Func<ProjectionDefinitionBuilder<TEntity>,
ProjectionDefinition<TEntity>> projectionDefinitionPredicate = null
    );

    Task<ITry<Guid>> TryUpdateAsync(
        IClientSessionHandle session,
        Func<FilterDefinitionBuilder<TEntity>, FilterDefinition<TEntity>>
filterDefinitionPredicate,
        Func<UpdateDefinitionBuilder<TEntity>, UpdateDefinition<TEntity>>
updateDefinitionPredicate
    );
```

```
    Task<ITry<Unit>> TryDeleteAsync(
        IClientSessionHandle session,
        Func<FilterDefinitionBuilder<TEntity>, FilterDefinition<TEntity>>
filterDefinitionPredicate
    );
}
```

Snippet 42: IRepository interface

As we can see, there are 6 basic CRUD [17] operations defined on top of this base interface. We have GET, GET MANY, WRITE, WRITE MANY, UPDATE and DELETE. Operations that perform WRITE operation on top of the database return ITry that indicates that the corresponding operation may fail and that we should get information why it failed. READ operations return IOption [18] which says that either there are some values or there are no values. Again, this response needs to be handled as it is forced by the compiler. The only difference is that read operation should not fail and if it does, it is an issue on our side and Internal exception is raised and logged accordingly. All specific interfaces for entities inherit from this interface and do not have additional operations.

Another interface that is worth mentioning is ITransaction interface.

```
public interface ITransaction
{
    IBusinessContext BusinessContext { get; }

    Task<ITry<TTransactionResult>>
TryTransactionAsync<TTransactionResult>(Func<IClientSessionHandle,
Task<TTransactionResult>> action);

    Task<ITry<Unit>> TryTransactionAsync(Func<IClientSessionHandle, Task>
action);

    ITry<Unit> TryTransaction(Action<IClientSessionHandle> action);
}
```

Snippet 43: ITransaction interface

It provides transactional operations as well as BusinessContext which is an interface that provides all business components. By using this interface, you can wrap all database related operations in a transaction context. By that, you ensure that either all operations succeed or none do. It provides ACID [19] operations on top of the database. There are different methods for different use cases (async or sync context), but they serve the same purpose. They start the session and commit only upon all successful operations. Otherwise, they abort everything. This interface supports multi-document transactions across all database replicas.

From the Components part we will describe only IStocksComponent as others follow the similar pattern. The list of all components includes: IArticlesComponent, ICompaniesComponent, IAuthComponent, IMLModelsComponent, IMailingComponent, IStocksComponent, IUsersComponent.

```
public interface IStocksComponent
{
    Task<IEnumerable<Domain.Entities.Stock>> GetManyAsync(
        IClientSessionHandle session,
        DateTime from = default,
        DateTime to = default,
        string companySymbol = default
    );

    Task<IEnumerable<SimpleLatestStockResponse>>
GetLatestAsync(IEnumerable<string> companies);

    Task AddNewAsync(IClientSessionHandle session, Range range);
}
```

Snippet 44: IStocksComponent interface

In this layer, we return specific types not wrapped in ITry or IOption anymore because we want to throw exceptions if a certain operation fails and therefore making sure that the whole transaction fails. There are three methods defined in this interface. First two are used by the Stocks Application Programming Interface (API) which is described in Presentation section 4.7. and the last one is used by a Stocks Job. We return Task only when working with jobs as we don't process the response after the operation finishes but we do log exceptions that are thrown if the operation fails.

Application layer also defines interfaces for jobs. The base interface is IJob:

```
public interface IJob
{
    void Create();
}
```

Snippet 45: IJob interface

All other interfaces (IArticlesJob, ICompaniesJob, IMLModelsJob, IDatabaseJob, IStocksJob) implement this interface and Create method with a specific scope and type.

5.5. Utilities

This layer acts as a Stock library with helper and extension methods.

5.5.1. Structure

```
├── StockSolution # VISUAL STUDIO solution
    ├── Core # solution folder
        # projects
       ...
        ├── Stock.Utilities
            ├── Attributes
                # description attribute for enum fields
                ├── FieldDescriptionAttribute.cs
            ├── Check
                # internal exception handler
                ├── Internal.cs
            ├── Enumerations
                # supported languages
                ├── Language.cs
            ├── Exceptions
                # developer/contract exception
                ├── InternalException.cs
            ├── Extensions
                # helper methods
                ├── CollectionExtensions.cs
                ├── Description.cs
                ├── Extensions.cs
            ├── Globalization
                # collection of keys
                ├── Keys
                # translation infrastructure
                ├── Globalization.cs
                # localization handler
                ├── Localization.cs
            ├── Helpers
                # helper wrapper functions
                ├── Functions
                # testing infrastructure
                ├── Tests
            ├── ML
                # sentiment analysis
                ├── Sentiment.cs
          ...
```

Snippet 46: Utilities folder structure

Utilities is an optional layer that we decided to add because it provides some helpful extension methods and internal exception types that we use across the whole application. It also contains the sentiment analysis functionality which relies on VaderSharp [20] library that uses Vader [21].

It also provides Globalization infrastructure that we use across the whole application. When a new translation needs to be added, it should be added to all languages with the specified key that needs to be the same in all languages and translation for the specific language. New key needs to be added the Keys class as well. If the message is dynamic, meaning it has some values that need to be injected during runtime, it can be specified using curly braces {}. e.g " *My name is {name}.*" The value inside curly braces is not important. Later, the key will be used like in Snippet 47.

```
Localization.Get(Keys.SomeKey, Language.En)
```

Snippet 47: Static localized message retrieval

The default value of the language is English and does not need to be specified. If you need to inject some values during runtime, the key for the message "*My name is {name} and my surname is {surname}."* will be used like:

```
Localization.Get(Keys.SomeKey, "John", "Doe").
```

Snippet 48: Dynamic localized message retrieval

There is also a base class for tests with test extension methods that enforce Arrange-Act-Assert (AAA) principle

There is also a base class for tests with test extension methods that enforce Arrange-Act-Assert (AAA) principle which is later used in test project as following:

```
public abstract class Test
{
    protected static void UnitTest<TArrangeResult, TActResult>(
        Func<Unit, TArrangeResult> arrange,
        Func<TArrangeResult, TActResult> act,
        Action<TActResult> assert
    )
    {
        var arrangeResult = arrange(Unit.Value);
        var actResult = act(arrangeResult);
        assert(actResult);
    }

    // ...
}
```

which is later used like:

```
public class DescriptionTests : Test
{
    [Fact]
    public void Enum_Field_Description_Equals_GetDescription_Result()
    {
        UnitTest(
            _ => TestEnum.Test1,
            p => p.GetDescription().SafeEquals("Test1"),
            Assert.True
        );
    }

    // ...
}
```

Throughout the system we use InternalException which is defined as following:

```
public class InternalException : Exception
{
    public InternalErrorType ErrorType { get; }

    public InternalException(InternalErrorType type, string message)
        : base(message)
    {
        ErrorType = type;
    }
}

public enum InternalErrorType
{
    InvalidOperation,
    ForbiddenOperation,
    IncorrectBehavior
}
```

This exception represents incorrect behavior that can occur. It is a developer mistake and is logged in Sentry as it represents a bug.

5.6. Infrastructure

Infrastructure is an implementation layer of the Application layer. It comprises of three units which are Components, Persistence and Jobs.

5.6.1. Structure

```
├── StockSolution # VISUAL STUDIO solution
    ├── Infrastructure # solution folder
        # projects
       ...
        ├── Stock.Components # implements business logic
        ├── Stock.Persistence # provides database communication logic
        ├── Stock.Jobs # jobs infrastructure
          ...
```

Snippet 49: Infrastructure folder

5.6.2. Components

Components represent the business layer on top of entities. For each entity, we have a corresponding component and each of them is responsible for manipulating entities by using either Persistence or Jobs. We currently have the following components: ArticlesComponent, CompaniesComponent , MlModelsComponent, StocksComponent and UsersComponent. Additionally, we have components for authentication/authorization and mailing , respectively AuthComponent and MailingComponent.

5.6.3. Persistence

Persistence comprises of three parts, Database, Repositories and UnitOfWork [22]. Database provides database configuration and migration capabilities. The Repository and UnitOfWork are intended to create an abstraction layer between the data access layer and the business logic layer of an application. Implementing them can make your application more flexible regarding implementation changes and integrations with external services without breaking changes in the codebase and simplifying the testing process.

5.6.4. Jobs

For every long running process in our system such as acquiring articles, stock prices, company information and building machine learning models, we use background jobs. In our system we have two types of jobs, recurring and fire-and-forget. A recurring job, depending on the job itself, runs in a repeated fashion such as articles, stock prices and company information acquiring and building the default recurrent neural network jobs. A fire-and-forget job runs once unless it fails for some reason after which it is rescheduled to run for a specified amount of times, such as jobs to build custom machine learning models. There is also a job for database migration and historical stock prices which are recurring jobs, but they are only executed manually by an administrator who has access to the job server dashboard.

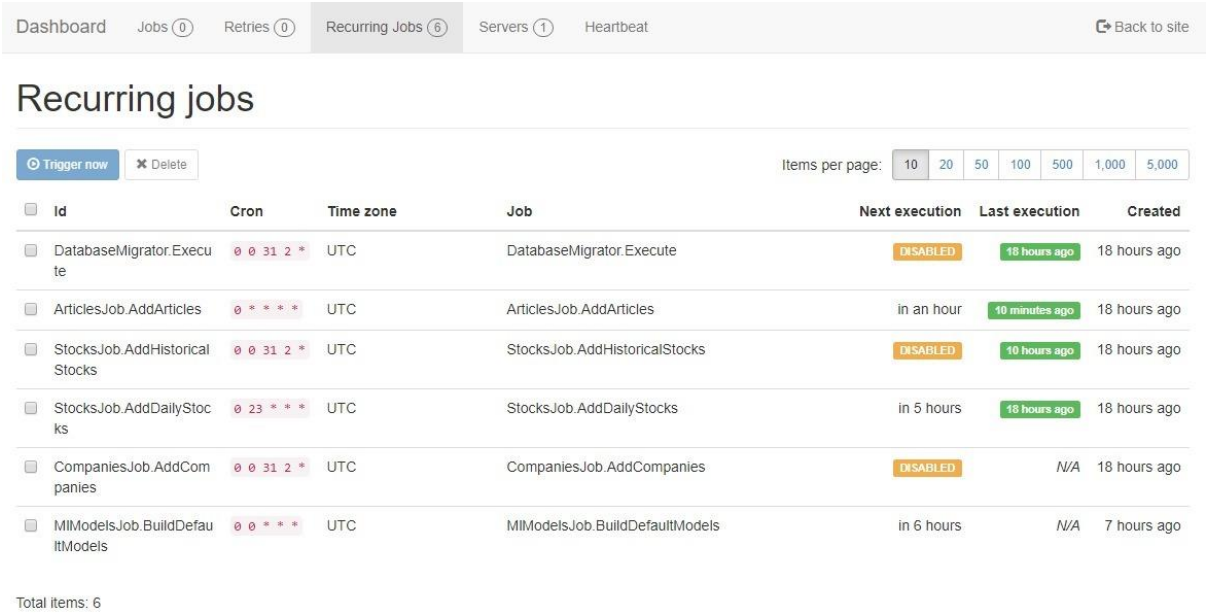

Figure 8: Jobs Dashboard

5.7. Presentation

The Presentation layer in the backend application is the API. The API follows the REST style and is done in ASP.NET Core Web API 3.1 using C# 8. The API section 5.1. explains the details of the implementation and the endpoints it provides.

5.7.1. API

This section focuses on the available API endpoints. It also describes how it makes use of the infrastructure it is built on.

The API acts as the interface to the frontend application to interact with the rest of the system. It provides available endpoints in the form of REST APIs. Each endpoint will be described in the Endpoints section 5.2. with a request/response example and the actual flow together with the Frontend and the ML will be described in the Integrations section 6.

The endpoints are available directly by navigating to https://www.predictfuturestocks.com/backend/swagger. The base URL of the endpoints is https://www.predictfuturestocks.com/backend/api/.

Auth endpoints

POST | Register | /auth/sign-up

API endpoint used to start the registration process.

Request		
Property	**Type**	**Description**
username **REQUIRED**	string	Valid username (min 5 characters long).
password **REQUIRED**	string	Valid password (min 8 characters, at least 1 uppercase and 1 lowercase letter, at least one number).
confirmPassword **REQUIRED**	string	Password confirmation.
email **REQUIRED**	string	Valid email address (Will be required in the next process).

Response	
Code	**Body**
200 OK	Use the verification code sent to your email.
400 **BAD REQUEST**	Depends on the request (email not valid, username too short, etc).
409 **CONFLICT**	Depends on the request e.g. Username already exists, User with provided email already exists

POST | Confirm registration | /auth/confirm-sign-up

API endpoint used to confirm the registration process.

Request		
Property	**Type**	**Description**
username **REQUIRED**	string	Valid username used in the sign up request.
verificationCode **REQUIRED**	string	Verification code sent to the email provided in the previous request.

Response	
Code	**Body**
200 OK	Sign-up successful. You are good to go.
403 FORBIDDEN	Invalid verification code.
404 NOT FOUND	User with this username does not exist.

POST | Register | /auth/sign-in

API endpoint used to sign in.

Request		
Property	**Type**	**Description**
username REQUIRED	string	Username used in the registration process.
password REQUIRED	string	Password used in the registration process.

Response	
Code	**Body**

200
OK
	{
	"accessToken": "",
	"expiresIn": "",
	"idToken": "",
	"newDeviceMetadata", "",
	"refreshToken": "",
	"tokenType":""
	}

401 UNAUTHORIZED	Incorrect username or password.

POST | Forgot password | /auth/forgot-password

API endpoint used to begin reset password process.

Request		
Property	**Type**	**Description**
username REQUIRED	string	Valid username used in the sign up request.

Response	
Code	**Body**
200 OK	Use the verification code sent to your email.
400 BAD REQUEST	The Username field is required.
404 NOT FOUND	User with this username does not exist.

POST | Reset password | /auth/reset-password

API endpoint used to confirm the password reset.

Request		
Property	**Type**	**Description**
username REQUIRED	string	Username of the user.
newPassword REQUIRED	string	New valid password
confirmationCode REQUIRED	string	Confirmation code sent to the email of the user.

Response	
Code	**Body**
200 OK	Reset password successful.
400 BAD REQUEST	Password requirements not fulfilled. (Response depends on type of requirement not met).
403 FORBIDDEN	Invalid verification code.
404 NOT FOUND	User with this username does not exist.

POST | Change password | /auth/change-password

API endpoint used to change password of the authenticated user.

Request		
Headers		
Property	**Type**	**Description**
AccessToken **REQUIRED**	string	Access token.
Authorization **REQUIRED**	string	Id token
Body Parameters		
Property	**Type**	**Description**
newPassword **REQUIRED**	string	Valid new password.
oldPassword **REQUIRED**	string	Previous password.

Response	
Code	**Body**
200 OK	Change password successful.
400 **BAD REQUEST**	Access or Id token undefined. New password related, same password rules as in the registration process.
401 **UNAUTHORIZED**	Invalid Access Token (if id token is incorrect there will be no message).

POST | Update token | /auth/update-token

API endpoint used for updating token.

Request		
Headers		
Property	**Type**	**Description**
Authorization **REQUIRED**	string	Id token
Body Parameters		

Property	Type	Description
refreshToken REQUIRED	string	Refresh token returned from the sign-in endpoint.

Response

Code	Body

200
OK
	{
	"accessToken": "",
	"expiresIn": "",
	"idToken": "",
	"newDeviceMetadata", "",
	"refreshToken": "",
	"tokenType":""
	}

400 BAD REQUEST	The RefreshToken field is required.
401 UNAUTHORIZED	Invalid Refresh Token (if id token is incorrect there will be no message).

POST | Change password | /auth/change-password

API endpoint used to logout the user.

Request

Headers

Property	Type	Description
AccessToken REQUIRED	string	Access token.
Authorization REQUIRED	string	Id token

Response

Code	Body
200 OK	Sign out successful.
400 BAD REQUEST	Access or Id token undefined.
401 UNAUTHORIZED	Access Token has been revoked. Invalid Access Token.

Open endpoints

Open endpoints are the collection of endpoints that do not require authenticated users.

Articles

GET | Get articles | /articles/all

API endpoint used to retrieve a list of articles for the specified parameters.

Request		
Property	**Type**	**Description**
companySymbol OPTIONAL	string	Company from which you want articles.
from OPTIONAL	string	Starting date from when you want articles.
from OPTIONAL	string	Ending date until when you want articles.

Response	
Code	**Body**
200 OK	[{ "id": "213423", "createdAt": "2020-02-19T16:00:00Z", "companySymbol": "amzn", "title": "Amazon researchers train AI to rewrite queries for better spoken language understanding", "polarity": "Neutral" }, ...] .
404 NOT FOUND	There are no items.

GET | Get articles | /articles

API endpoint used to retrieve all information of the specified article.

Request		
Property	**Type**	**Description**
id OPTIONAL	string	Id of the article.

Response	
Code	**Body**

200
OK
	{
	"id": "342",
	"createdAt": "2020-02-19T16:00:00Z",
	"version": "084368a3-5379-411b-a2f7-5fc3b16fe99e",
	"companySymbol": "amzn",
	"source": "VentureBeat",
	"title": "Amazon researchers train AI to rewrite queries for better spoken language understanding",
	"image":
	"https://cloud.iexapis.com/v1/news/image/a8028089-b97e-4fc0-8cba-b1ad1ffcf0e7",
	"link":
	"https://cloud.iexapis.com/v1/news/article/a8028089-b97e-4fc0-8cba-b1ad1ffcf0e7",
	"content": "In a new preprint paper, researchers describe an AI system that rewrites queries to reduce spoken language understanding errors in assistants like Alexa.",
	"polarity": "Neutral"
	}

404 NOT FOUND	Item not found.

Stocks

GET | Get stocks | /stocks/all
API endpoint used to retrieve a list of stocks for the specified parameters.

Request		
Property	**Type**	**Description**
companySymbol OPTIONAL	string	Company from which you want articles.
from OPTIONAL	string	Starting date from when you want articles.
from OPTIONAL	string	Ending date until when you want articles.

Response	
Code	**Body**

200
OK
	// If the company symbol is not specified.
	[
	{
	"companySymbol": "orcl",
	"createdAt": "4/25/2003 4:00:00 PM",
	"open": 11.96,
	"high": 12.02,
	"low": 11.77,
	"close": 11.79,
	"volume": 26159361
	},
	...
	]
	
	// If the company symbol is specified.
	[
	{
	"createdAt": "4/1/2003 4:00:00 PM",
	"open": 24.46,
	"high": 24.7,
	"low": 24.25,
	"close": 24.35,
	"volume": 49803234
	},
	...
	].

404 NOT FOUND	There are no items.

GET | Get latest stocks | /stocks/latest

API endpoint used to retrieve latest/current stock price of the companies currently in our system. There are no optional or required parameters for this request.

Response	
Code	**Body**

200
OK
	[
	{
	"companySymbol": "aapl",
	"content": {
	"open": 318.58,
	"high": 319.99,
	"low": 317.31,
	"close": 317.7,
	"volume": 25458115
	}
	},
	...
	]

404 NOT FOUND	There are no items.

Companies

GET | Get company | /companies

API endpoint used to retrieve a list of available companies in our system. There are no optional or required parameters for this request.

Response	
Code	**Body**
200 OK	[{ "name": "Amazon.com, Inc.", "symbol": "amzn" }, { "name": "Tesla, Inc.", "symbol": "tsla" }, ...]
404 NOT FOUND	There are no items.

GET | Get stocks | /companies

API endpoint used to retrieve information for the specified company.

Request		
Property	**Type**	**Description**
companySymbol OPTIONAL	string	Company for which you want to retrieve information.

Response	
Code	**Body**

200
OK
	{
	"id": "a49650ce-0e9c-49ea-8b98-3de8ce2e5a5e",
	"createdAt": "2020-02-16T16:00:00Z",
	"version": "b535f724-70c6-432d-a48a-
	28b207d2b467",
	"symbol": "msft",
	"name": "Microsoft Corp.",
	"ceo": "Satya Nadella",
	"description": "Microsoft Corp. engages in
	...
	The company was founded by Paul Gardner
	Allen and William Henry Gates III in 1975
	and is headquartered in Redmond, WA.",
	"tags": [
	"Technology Services",
	"Packaged Software"
	],
	"industry": "Packaged Software",
	"employees": 144000,
	"website": "http://www.microsoft.com",
	"city": "Redmond",
	"country": "US"
	}

404 NOT FOUND	Item not found.

Endpoints that require authenticated/authorized users

5.7.2. Users

GET | Get user basic information | /users/info

API endpoint used to retrieve users basic information.

Request		
Headers		
Property	**Type**	**Description**
AccessToken **REQUIRED**	string	Access token.
Authorization **REQUIRED**	string	Id token

Response	
Code	**Body**

200
OK
	{
	"firstName": "John",
	"lastName": "Doe"
	}

400 **BAD REQUEST**	Access or Id token undefined.
401 **UNAUTHORIZED**	Invalid Access Token (if id token is incorrect there will be no message).

GET | **Get users settings** | /users/settings

API endpoint used to retrieve user's settings.

Request		
Headers		
Property	**Type**	**Description**
AccessToken **REQUIRED**	string	Access token.
Authorization **REQUIRED**	string	Id token

Response	
Code	**Body**

200
OK
	{
	"favoriteCompanies": [
	{
	"symbol": "msft",
	"name": "Micrsoft Inc."
	},
	...
	],
	"language": "en"
	}

400 BAD REQUEST	Access or Id token undefined.
401 UNAUTHORIZED	Invalid Access Token (if id token is incorrect there will be no message).

GET | Get users settings | /users/search-history

API endpoint used to retrieve user's search history.

Request		
Headers		
Property	**Type**	**Description**
AccessToken REQUIRED	string	Access token.
Authorization REQUIRED	string	Id token

Response	
Code	**Body**

200
OK
	{
	"searchedCompanies": [
	{
	"searchedAt": "2020-0222T16:37:58.268Z",
	"symbol": "msft",
	"name": "Microsoft Inc."
	},
	...
	]
	}

400 BAD REQUEST	Access or Id token undefined.
401 UNAUTHORIZED	Invalid Access Token (if id token is incorrect there will be no message).

PUT | Update user's basic information | /users/info

API endpoint used to update user's basic information.

Request		
Headers		
Property	**Type**	**Description**
AccessToken **REQUIRED**	string	Access token.
Authorization **REQUIRED**	string	Id token
Body Parameters		
Property	**Type**	**Description**
firstName **REQUIRED**	string	Name.
lastName **REQUIRED**	string	Surname.

Response	
Code	**Body**

200
OK
	{
	"firstName": "John",
	"lastName": "Doe"
	}

400 **BAD REQUEST**	Access or Id token undefined. Name or surname must be defined. Validation errors (short name, long name, etc.)
401 **UNAUTHORIZED**	Invalid Access Token (if id token is incorrect there will be no message).

PUT | Update user settings | /users/settings

API endpoint used to update user's settings.

Request		
Headers		
Property	**Type**	**Description**
AccessToken **REQUIRED**	string	Access token.
Authorization **REQUIRED**	string	Id token
Body Parameters		
Property	**Type**	**Description**
favoriteCompanies **REQUIRED**	array	List of companies (symbol, name)
language **REQUIRED**	string	Preferred language.

Response	
Code	**Body**

200
OK
	{
	"favoriteCompanies": [
	{
	"symbol": "msft",
	"name": "Micrsoft Inc."
	},
	...
	],
	"language": "en"
	}

400 **BAD REQUEST**	Access or Id token undefined. Favorite companies or language must be defined. Some of specified companies are not supported. Specified language not supported.
401 **UNAUTHORIZED**	Invalid Access Token (if id token is incorrect there will be no message).

PUT | Update user's search history | /users/search-history

API endpoint used to update user's search history.

Request		
Headers		
Property	**Type**	**Description**
AccessToken **REQUIRED**	string	Access token.
Authorization **REQUIRED**	string	Id token
Body Parameters		
Property	**Type**	**Description**
searchedCompany **REQUIRED**	object	Company (symbol, name)

Response	
Code	**Body**

200
OK
	{
	"searchedCompanies": [
	{
	"searchedAt": "2020-02-2T16:37:58.268Z",
	"symbol": "msft",
	"name": "Microsoft Inc."
	},
	...
	]
	}

400 **BAD REQUEST**	Access or Id token undefined. Searched company must be defined. Some of specified companies are not supported.
401 **UNAUTHORIZED**	Invalid Access Token (if id token is incorrect there will be no message).

DELETE | Delete user | /users

API endpoint used to delete the user.

Request		
Headers		
Property	**Type**	**Description**
AccessToken **REQUIRED**	string	Access token.
Authorization **REQUIRED**	string	Id token

Response	
Code	**Body**
200 OK	Delete account successful
400 **BAD REQUEST**	Access or Id token undefined.
401 **UNAUTHORIZED**	Invalid Access Token (if id token is incorrect there will be no message).

Machine Learning Models

POST | Add model | /ml

API endpoint used to add new machine learning model for scheduled build.

Request		
Headers		
Property	**Type**	**Description**
AccessToken **REQUIRED**	string	Access token.
Authorization **REQUIRED**	string	Id token
Body Parameters		
Property	**Type**	**Description**
name **REQUIRED**	string	Name of the model.
type **REQUIRED**	string	Type of the model.

parameters **REQUIRED**	object	Model parameters object depending on the model type and desired results..

Response	
Code	**Body**

200
OK
	{
	"id": "8abb3b32-81fa-4d6c-9d82-9d927beeb4b1",
	"message": "Your model is scheduled for building. Building will start in 1 minutes."
	}

400 **BAD REQUEST**	Access or Id token undefined. Model type not supported. Validation exceptions (depending on the object sent in the request).
401 **UNAUTHORIZED**	Invalid Access Token (if id token is incorrect there will be no message).

GET | Get models | /ml/all

API endpoint used to get a list of models under specified conditions.

Request		
Headers		
Property	**Type**	**Description**
AccessToken **REQUIRED**	string	Access token.
Authorization **REQUIRED**	string	Id token

Response	
Code	**Body**
200 OK	[{ "id": "8abb3b32-81fa-4d6c-9d82-9d927beeb4b1", "createdAt": "2020-02-22T16:00:00Z" "type": "linear", "state": "Pending" }, ...]
400 BAD REQUEST	Access or Id token undefined.
401 UNAUTHORIZED	Invalid Access Token (if id token is incorrect there will be no message).
404 NOT FOUND	There are no items.

GET | Get model | /ml

API endpoint used to get a specified model.

Request		
Headers		
Property	**Type**	**Description**
AccessToken REQUIRED	string	Access token.
Authorization REQUIRED	string	Id token
Query Parameters		
Property	**Type**	**Description**
id OPTIONAL	string	Id of the model.

Response	
Code	**Body**

200
OK
	{
	"id": "8abb3b32-81fa-4d6c-9d82-9d927beeb4b1",
	"createdAt": "2020-02-22T16:00:00Z"
	"type": "linear",
	"parameters": {
	"randomState": 1,
	"fitIntercept": false,
	"normalize": false,
	"companySymbol": "msft",

	"forecastOut": 10, "features": ["Open", "High"], "featureToPredict": "Open", "testSize": 20 }, "state": "Pending", (//Created, Failed) "result": "Pending" (//Result) }
400 **BAD REQUEST**	Access or Id token undefined.
401 **UNAUTHORIZED**	Invalid Access Token (if id token is incorrect there will be no message).
404 **NOT FOUND**	Item not found.

GET | Get default model | /ml/default

API endpoint used to get a default model.

Request		
Headers		
Property	**Type**	**Description**
Authorization **REQUIRED**	string	Id token
Query Parameters		
Property	**Type**	**Description**
companySymbol OPTIONAL	string	Stock market symbol of the company.

Response	
Code	**Body**
200 OK	Model build result.
401 **UNAUTHORIZED**	
404 **NOT FOUND**	Item not found.

DELETE | Delete model | /ml

API endpoint sed to delete a specified model.

Request		
Headers		
Property	**Type**	**Description**
AccessToken **REQUIRED**	string	Access token.
Authorization **REQUIRED**	string	Id token
Query Parameters		
Property	**Type**	**Description**
id OPTIONAL	string	Id of the model.

Response	
Code	**Body**
200 OK	Delete model successful.
400 **BAD REQUEST**	Access or Id token undefined.
401 **UNAUTHORIZED**	Invalid Access Token (if id token is incorrect there will be no message).
404 **NOT FOUND**	Item not found.

Backend provides Swagger [23] user interface (UI) as well. Swagger is an implementation of the Open API Specification [24]. By using Swagger, it is possible to see all the available endpoints and test them directly without the need of REST client.

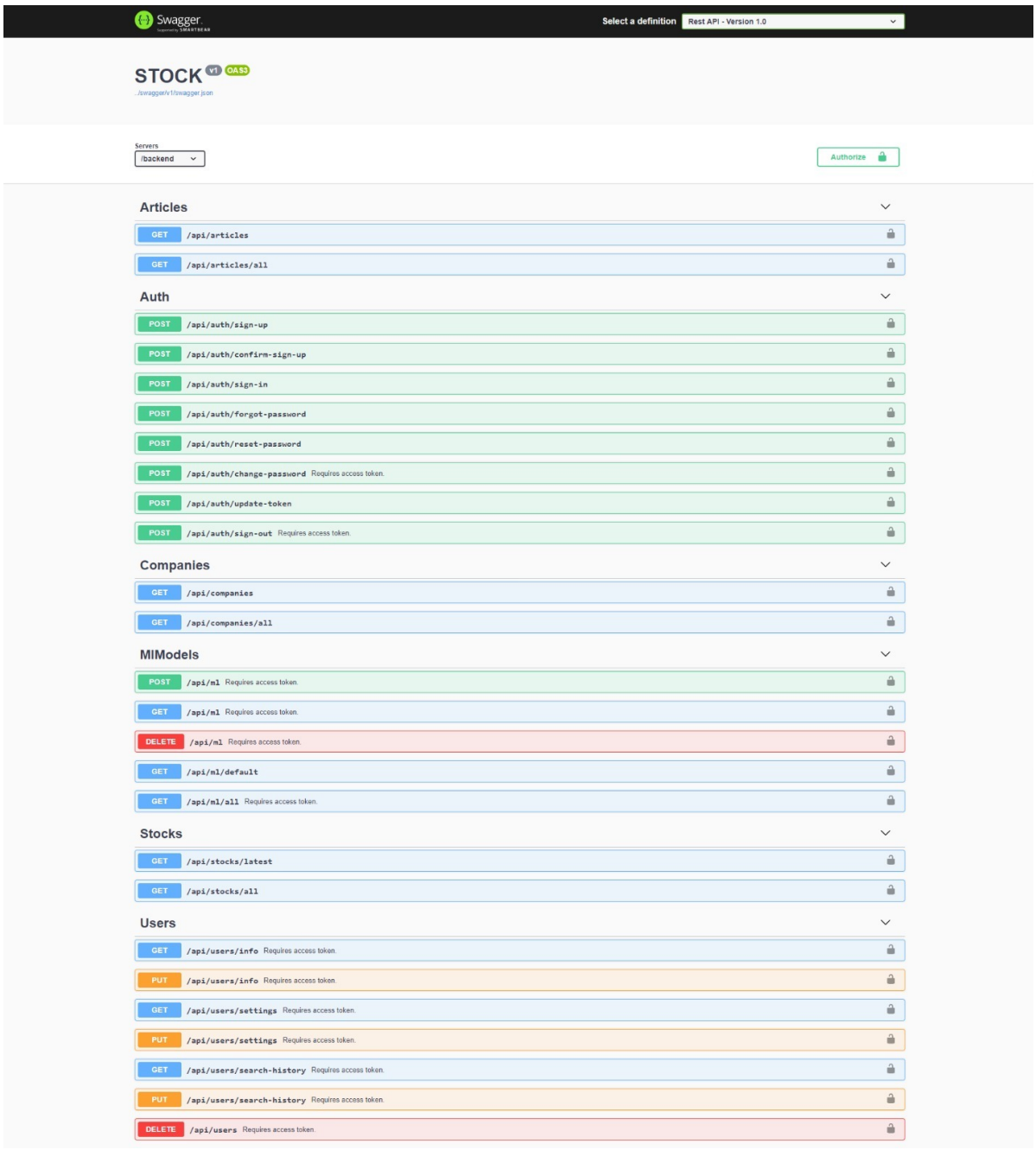

Figure 9: Swagger User Interface

5.8. Request flow

The diagram in Figure 10 describes the flow of the Create Model request in Section that is processed in the following way:

The client sends the request with the required information, in this case model type, model parameters as body parameters and idToken and AccessToken as headers. The API directly communicates with Cognito (pure auth related operations go through AuthComponent). If the idToken or AccessToken are not valid, 401 or 403 error codes are returned respectively. If the authentication is successful, the API communicates with the Domain layer calling MlModel create method. If the model cannot be built due to validation errors, ValidationException is returned to the API and then to the client. If the model building is successful, the API sends the created model to the MlModelsComponent that calls the underlying MlModelsRepository with a constructed insert command. That command is later sent to the Database. If the insert operation is unsuccessful, an error is returned to the MlModelsComponent which maps the error to a user-friendly error and returns it to the API. If the operation is successful, MlModelsRepository returns the id of the model to the MlModelsComponent which then schedules a build, where it sends the request for a job creation to the corresponding Job of that same model depending on the number of currently processed jobs on the remote server. It then sends the response with id and message containing also the scheduled time of building, which is then passed to the API where it processes the response, logs information and returns the result to the client.

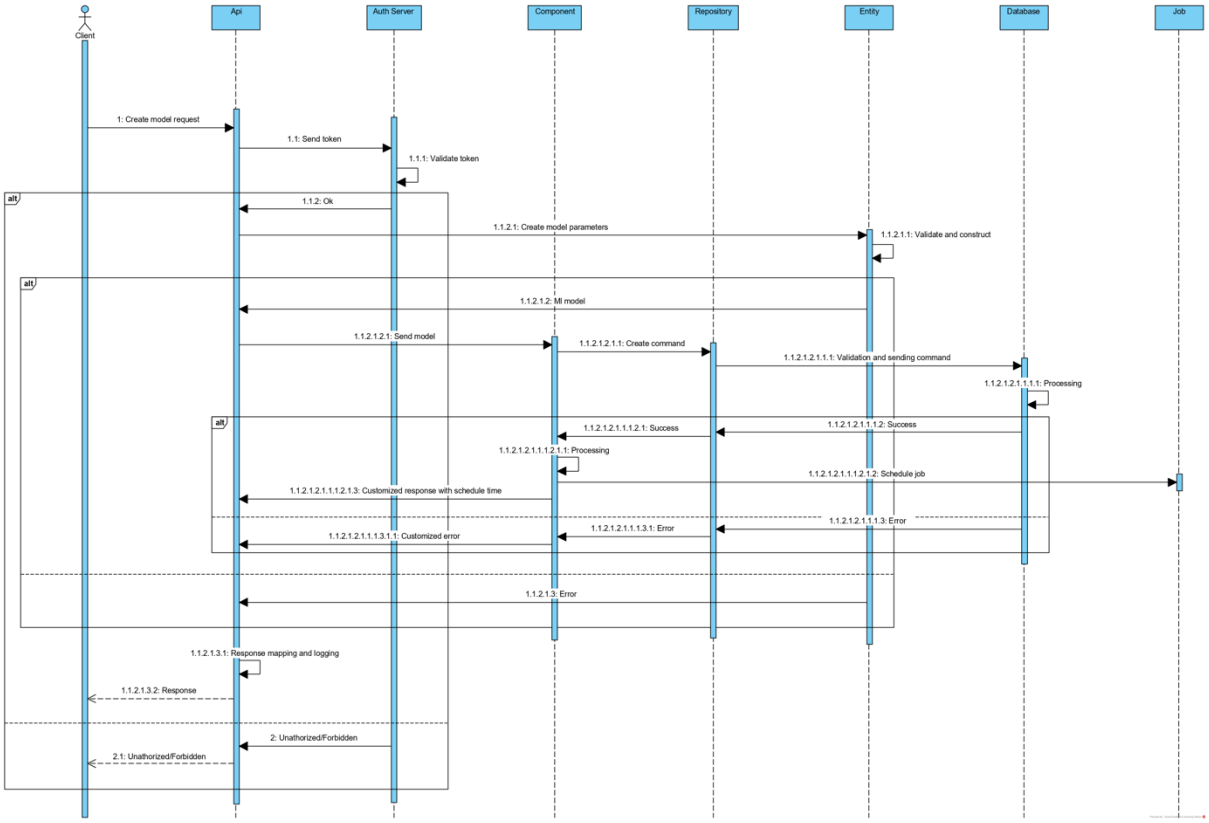

Figure 10: Request flow

6. Machine Learning Module

Machine Learning Module (ML) is responsible for creating stock predictions. It is designed as a separate module and it is hosted under Linux Virtual Machine. That allows the module scale up easily in case of a heavy load in the future.

6.1. Structure

Each prediction model has its own source code in order to have easier modification, and scaling. Individual models can be modified easily without disturbing data manipulation or any connection regarding the data sources. Detailed project structure is shown in Snippet 50.

```
stock_project  # Root Project Folder
├── app.py
├── regression_algorithms # Folder contains Regression Algorithms
│   ├── decision_tree_model.py
│   ├── lr_bayesian_ridge_model.py
│   ├── lr_default_model.py
│   ├── lr_elastic_model.py
│   ├── lr_lasso_lars_model.py
│   ├── lr_lasso_model.py
│   ├── lr_ridge_model.py
│   ├── random_forest_model.py
│   └── regression_model.py
├── rnn_algorithms # Folder contains Neural Network Algorithm
│   └── rnn_lstm_model.py
├── stock.ini
├── stock_ml_environment.yml
├── stock.sock
├── utils # Folder contains Helper Classes
│   ├── data_util.py
│   ├── linear_regression_helper.py
│   ├── neural_network_helper.py
└── wsgi.py
```

Snippet 50: ML Code Structure

As seen in Snippet 50, module has three main folders which contain regression and neural network models as well as the shared utility methods that is used by models. Currently, module provides nine different ways to predict stock prices. Users can create and customize each model in order to have their own prediction results

6.2. Workflow

Machine Learning Part only focuses on creating user customized prediction models and predicting future stock prices.

Project has two types of stock prediction. First one is called Default Stock Prediction which predicts stock prices by using Recurrent Neural Network with custom parameters that is assigned by development team. This model is responsible to show forecast stock prices when user goes to company dedicated page on the web site. Second type is called Custom Stock Prediction which essentially gives the user the power of creating prediction models by configuring their own parameters.

In order Custom Stock Prediction to work, user needs to create a custom prediction model with or without custom parameters. High level workflow diagram is shown in Figure 12.

Figure 11: High Level Workflow Diagram

After Machine Learning Module receives the model type and its parameters, it initializes the custom model, assigns its parameters, train and test the model and at the end, creates future stock price predictions. Results are converted to a JSON Object and send back to Backend Module.

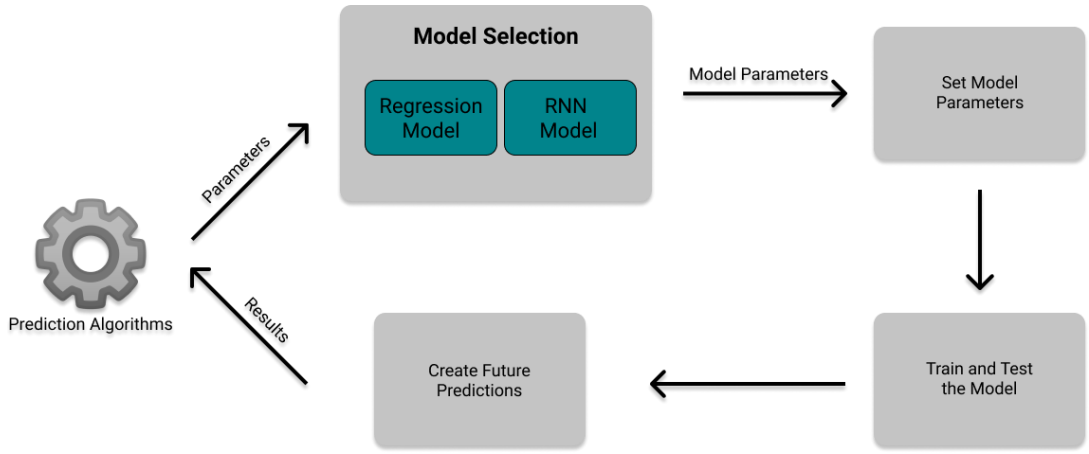

Figure 12: Low Level Workflow Diagram

Before the code description, remember that stock market prices are highly unpredictable since they are affected by different independent factors. This means that there is actually no consistent pattern or a silver bullet for stock

price prediction. Thus, we let our users to **customize** their prediction models and **compare** their own results.

6.3. Code

In this section, source code for the stock prediction is described with details; including code structure and parameter definition with an example source code.

Machine Learning Codes are divided into three main folders as follows:

- *Regression Algorithms Folder* contains eight different regression models to be used for stock prediction:
  - Decision Tree
  - Random Forest
  - Standard Regression
  - Bayesian Ridge Regression
  - Elastic Regression
  - Lasso Regression
  - Lasso Lars Regression
  - Ridge Regression

- *Neural Network Algorithms Folder* contains one Recurrent Neural Network Model to be used for stock prediction:
  - Recurrent Neural Network with Long-Short Term Memory

- *Utility Folder* contains one helper class to manage data retrieval, two helper class for regression and neural network related methods.

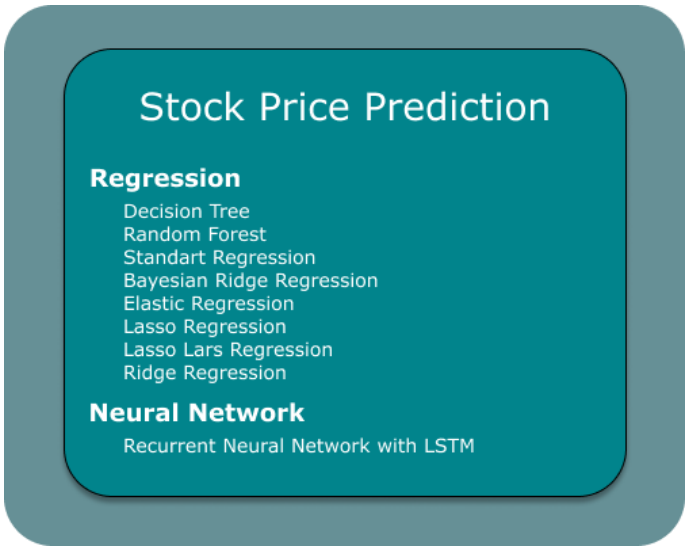

Figure 13: Module Overview

6.3.1. Regression Models

STOCK Project contains eight different regression models for prediction purposes. Each regression model has common and unique parameters depending on their type and inherits the Model class located under model.py.

Classes and Code Explanation

- model.py has three main methods to run and a constructor for prediction model parameters.

Constructor

```
self.parameters = {
    'companySymbol': 'orcl',

    'features': ['high', 'low', 'open', 'volume'],
    'featureToPredict': 'close',
    'forecastOut': 7,
    'testSize': 0.25,
    'randomState': 42,
    'precision':   3,

    'alpha': 0.95,
    'fitIntercept': False,
    'normalize': False,
    'maxIteration': 8192,
    'tol': 0.01,
    'preCompute': True,
    'warmStart': True,
    'positive': False,
    'selection': 'random',
    'alpha2': 1e-06,
    'lambda': 1e-06,
    'lambda2': 1e-06,
    'computeScore': True,
    'l1Ratio': 10.85,
    'eps': 0.01,
    'fitPath': True,
    'solver': 'auto',
    'criterion': 'friedman_mse',
    'splitter': 'best',
    'maxDepth': 512,
    'maxFeatures': 5,
    'minImpurityDecrease': 0.0,
    'minSamplesLeaf': 1,
    'minSamplesSplit': 2,
    'maxLeafNodes': None,
    'preSort': True,
    'bootstrap': True,
    'estimators': 1024,
    'oobScore': False,
}
```

Snippet 51: Parameters Used for Regression Model

self.parameters is a dictionary which contains all the customizable parameters for the regression models.

Split Data

Historical stock prices must be divided into pieces for the prediction. These pieces will be used during the training and evaluation the regression model as well as predicting the future stock prices. This task is assigned to split_data method.

```
def split_data(self):
    self.X_train, self.X_test,
    self.y_train, self.y_test,
    self.x_forecast = prepare_data(company_symbol=self.parameters['companySymbol'],
                                   result_feature=self.parameters['featureToPredict'],
                                   features=self.parameters['features'],
                                   forecast_out=self.parameters['forecastOut'],
                                   test_size=self.parameters['testSize'],
                                   random_state=self.parameters['randomState'])
```

Snippet 52: Implementation of Split Data Method

During its execution, split_data calls prepare_data method which serves as a helper method. prepare_data requires six parameters in order to operate:

- *company_symbol*
  Stock market symbol of the company that user would like to make prediction on.

- *result_feature*
  One value regarding what user would like to make prediction (Close, Open stock prices etc..). User can only predict one value per model since it is a regression model.

- *features*
  List of features that user wants to use during the prediction. It can be one feature or all the features in the dataset. ['high', 'low', 'open', 'volume'].

- *forecast_out*
  Represents how many days in the future user wants to make prediction. Important part here is that test data must be bigger than forecast out days. Otherwise, algorithm does not have sufficient data to make prediction.

- *test_size*
  Test dataset size.

- *random_state*
  The seed of the pseudo random number generator that selects a random feature to update. If random state is an integer value, the seed used by the random number generator is the one set for random_state variable. If None, the random generator is the Random State instance used by np.random.

Fit and Evaluate the Model
After user selects custom parameters are assigned to the prediction model, fit_evaluate method runs. Method is responsible for fitting the user selected model, evaluating it and predicting the future stock prices.

fit_evaluate method uses:

- fit()
- get_model_performance()
- predict()
- get_future_predictions()
- dictionary_to_json()

helper methods to perform its functionality. These methods are described in Section 5.3.3 of the document.

Shared Methods Between Regression Models
For simplicity, common methods which is implemented for each model is described under this section in order not to repeat the same descriptions.

train_model method is responsible of training the prediction model with its specified parameters. It initializes the prediction model, set its parameters and calls the create_model method. In Snippet 53, it is shown an example code for Decision Tree Model.

```
def train_model(params):
    model = DecisionTreeModel()
    model.parameters = model.set_parameters(parameter_dict=params)
    model.split_data()

    return model.create_model()
```

Snippet 53: Implementation of train_model Method

As it is shown, method creates Decision Tree Model, sets its parameters, apply data manipulation and creates the prediction model. All nine prediction models have train_model method to create their own prediction model.

create_model is responsible of creating the prediction models with the given parameters by using sklearn library [25]. Each prediction models

have various parameters which can be unique to the model. create_model method for each prediction model is for dedicated model section.

Shared Parameters Between Regression Models
For simplicity, common methods which is implemented for each model is described under this section in order not to repeat the same descriptions.

Each regression model has various parameters that are configurable by user. In this section all the common parameters that are used for all regression models are described:

- *criterion*
  The function to measure the quality of a split. Supported criteria are mse for the mean squared error, which is equal to variance reduction as feature selection criterion and minimizes the L2 Loss using the mean of each terminal node. friedman_mse, which uses mean squared error with Friedman's improvement score for potential splits. mae for the mean absolute error, which minimizes the L1 loss using the median of each terminal node.

- *max_depth*
  The maximum depth of the tree. If None then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split sample.

- *max_features*
  The number of features to consider when looking for the best split. Cannot be bigger than the number of features in the dataset.

- *min_samples_split*
  The minimum number of samples required to split an internal node. If int , then consider min_samples_split as the minimum number. If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.

- *min_samples_leaf*
  The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches. This may have effect of smoothing the model, especially in regression. If int, then consider min_samples_leaf as the minimum number. If float, then min_samples_leaf is a fraction and ceil(min_samples_leaf * n_samples) are the minimum number of samples for each node.

- *max_leaf_nodes*
  Grow a tree with max_leaf_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.

- *min_impurity_decrease*
  A node will be split if this split induces a decrease of the impurity greater than or equal to this value.

- *random_state*
  The seed of the pseudo random number generator that selects a random feature to update. If random state is an integer value, the seed used by the random number generator is the one set for random_state variable. If None, the random generator is the Random State instance used by np.random.

- *fit_intercept*
  Whether to calculate the intercept for the model. If set to False, no intercept will be used in calculations. The seed of the pseudo random number generator that selects a random feature to update. If random state is an integer value, the seed used by the random number generator is the one set for random_state variable. If None, the random generator is the Random State instance used by np.random.

- *normalize*
  This parameter is ignored when fit_intercept is set to False. If False, then the regressor will be normalized before regression by subtracting the mean and dividing by the L2-norm.

- *alpha*
  Constant that multiples the penalty terms. Default is 1.0.

- *pre_compute*
  Whether to use a pre-computed Gram matrix to speed up calculations. The Gram matrix can also be passed as argument. For sparse input this option is always True to preserve sparsity.

- *warm_start*
  When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution.

- *selection*
  If set to random, a random coefficient is updated every stage rather than looping over features sequentially by default. This often leads

to significantly faster convergence especially when tol is higher than 1e-4.

- *max_iter*
  The maximum number of iterations.

Unique parameters are described under dedicated model section.

Decision Tree Model

Decision Tree [26] class has two methods to run. create_model method is for creating the model and train_model method is for training the model. Model can get nine different parameters.

Create Model Method
Method uses DecisionTreeRegressor method to create the model with the given parameters. Once the model is ready, fit_evaluate method is called to fit and evaluate the model, so user can check how successful her/his model is.

- **DecisionTreeRegressor():** Method from sklearn library to create Decision Tree Model. All its parameters can be modified by the user.

- **fit_evaluate():** Helper Method that is described in Section 5.3.3 of the documentation. It allows the created model to be fit, evaluated and predict the future stock prices. Then, results will be transformed into a JSON Object and sent to the backend.

```
def create_model(self):
    # Fitting regression to the training data
    self.MODEL = DecisionTreeRegressor(criterion=self.parameters['criterion'],
                                       splitter=self.parameters['splitter'],
                                       max_depth=self.parameters['maxDepth'],
                                       min_samples_split=self.parameters['minSamplesSplit'],
                                       min_samples_leaf=self.parameters['minSamplesLeaf'],
                                       max_features=self.parameters['maxFeatures'],
                                       max_leaf_nodes=self.parameters['maxLeafNodes'],
                                       min_impurity_decrease=self.parameters['minImpurityDecrease'],
                                       presort=self.parameters['preSort'])

    return self.fit_evaluate()
```

Snippet 54 : Parameters for Decision Tree Model

- *splitter*
  The strategy used to choose the split at each node. Supported strategies are **best,** to choose the best split, and **random**, to choose the best random split.

- *presort*
  Whether to presort the data to speed up the finding of best splits in fitting. For the default settings of a decision tree on large datasets, setting this to true may slow down the training process. When using either a smaller dataset or a restricted depth, this may speed up the training.

Random Forest Model

Random Forest [27] Model can get twelve different parameters which three of them unique for Random Forest Model which described in Snippet 55:

```
def create_model(self):
        # Fitting regression to the training data
        self.MODEL = RandomForestRegressor(
                                        bootstrap=self.parameters['bootstrap'],
                                        criterion=self.parameters['criterion'],
                                        max_depth=self.parameters['maxDepth'],
                                        max_features=self.parameters['maxFeatures'],
                                        max_leaf_nodes=self.parameters['maxLeafNodes'],
                                        min_impurity_decrease=self.parameters['minImpurityDecrease'],
                                        min_samples_leaf=self.parameters['minSamplesLeaf'],
                                        min_samples_split=self.parameters['minSamplesSplit'],
                                        n_estimators=self.parameters['estimators'],
                                        oob_score=self.parameters['oobScore'],
                                        random_state=self.parameters['randomState'],
                                        warm_start=self.parameters['warmStart'])
        return self.fit_evaluate()
```

Snippet 55: Parameters of Standard Regression Model

- *bootstrap*
  Whether *bootstrap* samples are used when building trees. If False, then the whole dataset is used to build each tree.

- *n_estimators*
  The number of trees in the forest.

- *oob_score*
  Whether to use out-of-bag samples to estimate the R^2 on unseen data.

Standard Regression Model

This is the basic Linear Regression [28] Model that has two configurable parameters. Model is created by using LinearRegression method from sklearn library.

```
def create_model(self):
        # Fitting regression to the training data
    self.MODEL = LinearRegression(fit_intercept=self.parameters['fitIntercept'],
                                  normalize=self.parameters['normalize'],

  return self.fit_evaluate()
```

Snippet 56: Parameters of Standard Regression Model

Bayesian Ridge Regression Model

Bayesian Ridge [29] Model has nine configurable parameters. Model is created by using BayesianRidge method from sklearn library.

```
def create_model(self):
        # Fitting regression to the training data
        self.MODEL = BayesianRidge(n_iter=self.parameters['maxIteration'],
                    tol=self.parameters['tol'],
                    alpha_1=self.parameters['alpha'],
                    alpha_2=self.parameters['alpha2'],
                    lambda_1=self.parameters['lambda'],
                    lambda_2=self.parameters['lambda2'],
                    compute_score=self.parameters['computeScore'],
                    fit_intercept=self.parameters['fitIntercept'],
                    normalize=self.parameters['normalize'])

  return self.fit_evaluate()
```

Snippet 57: Parameters of Bayesian Ridge Model

- *tol*
  Stop the algorithm if weight has converged.

- *alpha_1*
  Shape parameter for the Gamma Distribution prior over the alpha parameter.

- *alpha_2*
  Inverse scale parameter (rate parameter) for the Gamma Distribution prior over the alpha parameter.

- *lambda_1*
  Shape parameter for the Gamma Distribution prior over the alpha parameter.

- *lambda_2*
  Inverse scale parameter (rate parameter) for the Gamma Distribution prior over the alpha parameter.

- *compute_score*
  If True, compute the log marginal likelihood at each iteration of the optimization. Default is False.

Elastic Regression Model

Elastic Ridge Model has eleven configurable parameters. Model is created by using BayesianRidge method from sklearn library.

```
def create_model(self):
    # Fitting regression to the training data
    self.MODEL = ElasticNet(alpha=self.parameters['alpha'],
                            l1_ratio=self.parameters['l1Ratio'],
                            fit_intercept=self.parameters['fitIntercept'],
                            normalize=self.parameters['normalize'],
                            precompute=self.parameters['preCompute'],
                            max_iter=self.parameters['maxIteration'],
                            tol=self.parameters['tol'],
                            warm_start=self.parameters['warmStart'],
                            positive=self.parameters['positive'],
                            random_state=self.parameters['randomState'],
                            selection=self.parameters['selection'])

  return self.fit_evaluate()
```

Snippet 58: Parameters for Elastic Regression Model

- *alpha*
  Constant that multiples the penalty terms. Default is 1.0.

- *l1_ratio*
  Stop the algorithm if weight has converged. Default is 1.e-3.

- *tol*
  The tolerance for the optimization. If the updates are smaller than tol, the optimization code checks the dual gap for optimality and continues until it is smaller than tol.

- *positive*
  When set to True, forces the coefficients to be positive.

Lasso Lars Regression Model

Lasso Lars [30] Model has ten configurable parameters. Model is created by using Lasso method from sklearn library. Lasso model is an optimized version of Elastic Regression with **l1_ratio = 1.0**.

```
def create_model(self):
        # Fitting regression to the training data
        self.MODEL = LassoLars(alpha=self.parameters['alpha'],
                               fit_intercept=self.parameters['fitIntercept'],
                               normalize=self.parameters['normalize'],
                               precompute=self.parameters['preCompute'],
                               max_iter=self.parameters['maxIteration'],
                               eps=self.parameters['eps'],
                               fit_path=self.parameters['fitPath'],
                               positive=self.parameters['positive'])

    return self.fit_evaluate()
```

Snippet 59: Parameters of Lasso Regression Model

- *alpha*
  Constant that multiples the penalty terms. Default is 1.0.

- *eps*
  The machine-precision regularization in the computation of the Cholesky diagonal factors. Increase this for very ill-conditioned systems. Unlike the tol parameter in some iterative optimization-based algorithms, this parameter does not control the tolerance of the optimization.

- *fit_path*
  If True, the full path is stored in the coef_path_ attribute. If you compute the solution for a large problem or many targets, setting fit_path to False will lead to a speedup, especially with a small alpha.

- *positive*
  Under the positive restriction the model coefficients will not converge to the ordinary-least-squares solution for small values of alpha.

Ridge Regression Model
Ridge [31] Model has eight configurable parameters. Model is created by using Ridge method from sklearn library.

```
def create_model(self):
        # Fitting regression to the training data
        self.MODEL = Ridge(alpha=self.parameters['alpha'],
                           fit_intercept=self.parameters['fitIntercept'],
                           normalize=self.parameters['normalize'],
                           max_iter=self.parameters['maxIteration'],
                           tol=self.parameters['tol'],
                           solver=self.parameters['solver'],
                           random_state=self.parameters['randomState'])

    return self.fit_evaluate()
```

Snippet 60: Parameters of Ridge Regression

- *alpha*
  Regularization strength; must be a positive float. Regularization improves the conditioning of the problem and reduces the variance of the estimates. Larger values specify stronger regularization.

- *tol*
  Precision of the solution.

- *Solver*
  Solver to use in the computational routines:
  - ***auto*** chooses the solver automatically based on the type of data.

  - ***svd*** uses a Singular Value Decomposition of X to compute the Ridge coefficients. More stable for singular matrices than 'cholesky'.

  - ***cholesky*** uses the standard scipy.linalg.solve function to obtain a closed-form solution.

- ***sparse_cg*** uses the conjugate gradient solver as found in scipy.sparse.linalg.cg. As an iterative algorithm, this solver is more appropriate than 'cholesky' for large-scale data (possibility to set `tol` and `max_iter`).

- ***lsqr*** uses the dedicated regularized least-squares routine scipy.sparse.linalg.lsqr. It is the fastest and uses an iterative procedure.

- ***sag*** uses a Stochastic Average Gradient descent, and 'saga' uses its improved, unbiased version named SAGA. Both methods also use an iterative procedure and are often faster than other solvers when both n_samples and n_features are large. Note that 'sag' and 'saga' fast convergence is only guaranteed on features with approximately the same scale. You can preprocess the data with a scaler from sklearn.preprocessing.

All last five solvers support both dense and sparse data. However, only 'sag' and 'sparse_cg' supports sparse input when `fit_intercept` is True.

6.3.2. Neural Network Models

STOCK Project contains one neural network model for prediction purposes. Between all neural networks, recurrent neural network is the most suitable solution for stock prediction.

Classes and Code Explanation

Recurrent Neural Networks [32] are designed to solve sequence prediction problems. Those sequences can be words in a sentence, company production values or stock prices. In our case, we are trying to predict stock prices.

We used Many to One RNN [33] Model which produces one output value after training over multiple input values.

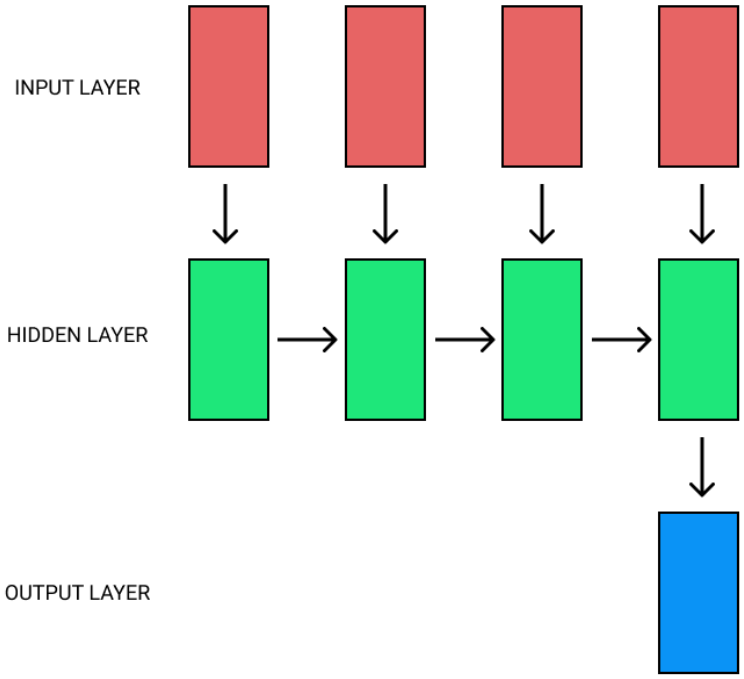

Figure 14: Many to One Recurrent Neural Network Model

When algorithm creates prediction for the next day, this day is added to dataset as a **real** value. On the next iteration, algorithm makes a prediction for the second next day in the future by considering already predicted value. This will allow us to forecast for next day or multiple days in the future.

Assume that user wants to forecast stock values of a Company A for next 5 days. Algorithm will retrieve historical stock prices from backend module and pre-process the data. This part will be explained in details in Section 5.3.3. During the pre-process, historical data will be divided to sequences. Those sequences will be used as an input to the model and at the end algorithm will output the stock price of next day in the future. Since user asked for five days of forecasting, prediction phase will iterate five times by adding each prediction to dataset as a real value. Refer to Figure 18 for better understanding.

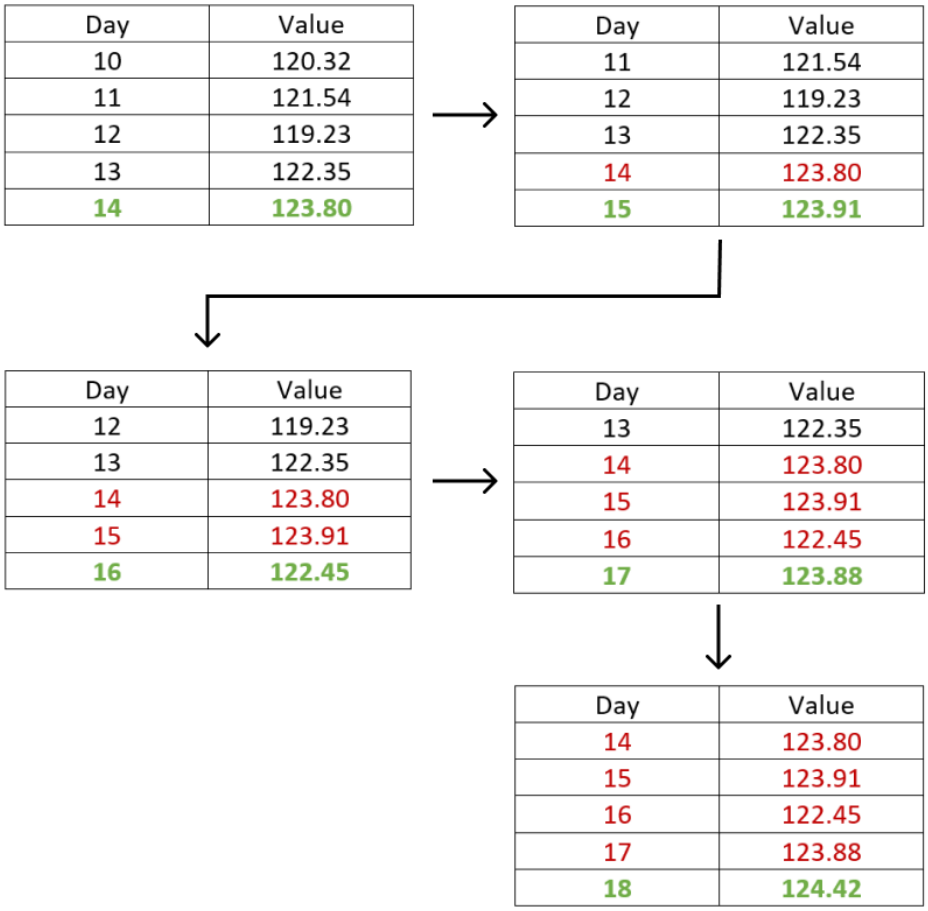

Figure 15: Future Value Prediction Example

Above example shows predicting stock prices of Company A for five days in the future. Algorithm has historical stock price data up to Day 13. After predicting stock price for Day 14, algorithm put this stock value to the dataset as a **real** value. Then, algorithm runs again for the next prediction and uses the historical stock prices up to Day 14 and so on.

Note that; Green text indicates new prediction and Red text indicates previously predicted value which inserted to dataset as a real value.

Recurrent Neural Network Parameters

Recurrent Neural Network Model has various parameters that are configurable by user. Parameters definition as following:

- *seqLen*
  Stock Prices must be pre-processed in a way that it represents each seqLen days of stock as one sequence. Each sequence should start from the second element of the previous sequence.

SEQ_1	1	2	3	4	5	6	7	8	9
SEQ_2	2	3	4	5	6	7	8	9	10
SEQ_3	3	4	5	6	7	8	9	10	11
SEQ_4	4	5	6	7	8	9	10	11	12

Figure 16: Sequence Example

- *forecastOut*
  Indicates how many days in the future user wants to make prediction. Size of the test data must be higher than forecastOut days since it is not possible to predict future days if there is not enough data.

- *features*
  List of features that user wants to use during the prediction. It can be one feature or all the features in the dataset.

- *featureToPredict*
  One value regarding what user would like to make prediction. User can only predict one value per model.

- *testSize*
  Size of the test dataset.

- *scaler*
  Scaler [34] object to scale values in order neural network to converge faster. Implemented scalers are as follows:

  - **StandardScaler**: Removes the mean and scales the data to unit variance.

  - **MinMaxScaler**: Rescales the data set such that all feature values are in the range [0, 1].

  - **MaxAbsScaler: Absolute** values are mapped in the range [0, 1]. On positive only data, this scaler behaves similarly to MinMaxScaler.

  - **RobustScaler:** Centering and scaling statistics of this scaler are based on percentiles and are therefore not influenced by a few number of very large marginal outliers.

- *epochs*
  Number of times that network will iterate over the training data.

- *batchSize*
  The batch size limits the number of samples to be shown to the network before a weight update can be performed.

- *networkLayers*
  Number of layers that will be used to build the network.

  - *First Network Layer*
    Input layer of the RNN. It will take the input with the given data and batch size.

  - *Hidden Network Layer*
    This layers will basically help RNN to learn the training data better. Input for the hidden network layer comes from the first network layer.

  - *Last Network Layer*
    Every single layer before the last layer will be having return_sequences = true. So, the next layer (if it is not the last layer) can process the values. Last layer is the only layer that will not have return_sequences parameter on.

  - *Dense Network Layer*
    This layer will get the value from the last network layer. Layer will output a single prediction value.

- *networkUnits*
  Number of neurons that will used for each layer of the network except the Dense Layer. Dense layer need only 1 Neuron to operate since it is an output layer.

- *dropOut*
  Dropout is a way to prevent over-fitting. It consists in randomly setting a fraction rate of input units to 0 at each update during training time, which helps prevent over-fitting.

- *batchNormalization*
  Normalizes the activation of the previous layer at each batch.

- *metrics*
  There are three different metrics that can be used for model evaluation. User can choose up to 3 evaluation metrics.

  mse : Mean Square Error
  mae : Mean Absolute Error

mape: Mean Absolute Percentage Error

- *optimizer*
  User can choose six different optimizers for RNN. There can be only one optimizer applied for one model. Implemented optimizers are as follows:
  Adam, Adamax, Nadam, SGD, RMSprop, Adadelta

- *learningRate*
  The amount the weights are updated during training.

- *beta*
  The exponential decay rate for the first moment estimates.

- *beta2*
  The exponential decay rate for the second moment estimates.

- *epsilon*
  A small number that prevents division by zero.

- *decay*
  After each update, the weights of the neural network are multiplied by a factor slightly less than one. This stops the weights from growing too large.

- *amsgrad*
  Whether to apply the AMSGrad variant.

- *scheduleDecay*
  Drops the learning rate by a given factor in every few epochs.

- *momentum*
  Parameter that accelerates SGD in the relevant direction and dampens oscillations.

- *nesterov*
  Whether to apply Nesterov Momentum.

- *rho*
  The decay factor or the exponentially weighted average over the square of the gradients.

Recurrent Neural Network Model

Constructor

self.parameters is a dictionary contains all the customizable parameters. Even though user leaves a parameter blank, default value will be used.

```
self.parameters = \
  {
    'companySymbol': 'amzn',

    # Prediction Parameters
    'seqLen': 45,
    'forecastOut': 7,
    'features': ['close', 'open', 'high'],
    'featureToPredict': 'close',
    'testSize': 0.10,
    'scaler': 'StandardScaler',

    # Rnn Parameters
    'epochs': 7,
    'batchSize': 24,
    'networkUnits': [128, 128, 64, 64],
    'networkLayers': 4,
    'dropOut': 0.3,
    'batchNormalization': True,
    'metrics': ['mse', 'mae', 'mape'],
    'loss': 'mse',
    'optimizer': 'Adam',

    # Optimizer Parameters
    'learningRate': 0.001,
    'beta': 0.9,
    'beta2': 0.999,
    'epsilon': 0.099,
    'decay': 0.00000001,
    'amsgrad': False,
    'scheduleDecay': 0.002,
    'momentum': 0.,
    'nesterov': False,
    'rho': 0.9
  }
```

Snippet 61: Parameters of Recurrent Neural Network

RNN Model has seven extra class objects comparing the regression model. Those are **Data set Parameters, Scaler Object and RNN Model Object.** *Data set Parameters* are used for easily dividing the historical data, creating the sequence of historical stock prices and validating the results.

In order RNN Model to perform better, input values must be scaled. In our implementation, we used 0 to 1 scale meaning the lowest data in the dataset will be 0 and highest will be 1. All the others scaled in this interval. After training, these scaled values must be returned to their original values in order to make validation. *Scaler Object* holds the information of the scaler's structure, so values can be returned to their original values.
Model Parameter holds the information regarding the RNN Model.

Create Network
This method creates a Recurrent Neural Network Model based on the user's selection. Method will:

- Create train_x, train_y datasets to use for training.
- Create validation_x, validation_y datasets to use for testing.
- Create sequential data from the historical stock prices.
- Create the RNN Model.

At the beginning, method creates data sets to be used for training and testing by using given company stock symbol, stock price features (open, close stock prices etc.), feature to be predicted, how many days ahead user would like to make prediction, sequence length of the forecast and the test size of the test data set which will be used during the test phase.

After necessary data sets are created, recurrent neural network is created by given parameters. Sequential recurrent neural network with long-short term memory (LSTM) is used for stock prediction. LSTM allows RNN to remember the past data while making predictions on the current input.
Since the network is configurable, user can select the number of neurons and layers of the network, and parameters such as dropout, batch normalization or optimizer.

At the end, method will call **train_network** method.

Train Network
In order to train RNN, following steps must be done:

- Decide which defined optimizer user is selected and set it for the RNN Model.
- Compile the RNN Model by using given loss function, optimizer and metrics.
- Fit and validate the RNN Model by using train_x, train_x, validation_x and validation_y datasets.

```
# Initialize and configure RNN Optimizer
opt = self.configure_optimizer(self.parameters['optimizer'])

# Compiling the RNN
self.RNN_MODEL.compile(loss=self.parameters['loss'],
                       optimizer=opt,
                       metrics=self.parameters['metrics'])

# Train model
self.RNN_MODEL.fit(self.train_x, self.train_y,
                   batch_size=self.parameters['batchSize'],
                   epochs=self.parameters['epochs'],
                   validation_data=(self.validation_x, self.validation_y),
                   shuffle=True)
```

Snippet 62: Example Code Part of train_network Method

At the end, results must be converted to JSON Object.

6.3.3. Helper Classes

Retrieve Stock Prices
Retrieving stock prices is one of the important parts of the ML Module. Stock prices are retrieved from backend by using company symbol, stock features and prediction feature parameters. By using the parameters, module connects to backend, receives the stock prices and put them into a pandas [35] dataframe.

Converting Model Results to a JSON Format
After evaluating the model and creating the future predictions, results must be converted to a proper JSON format, so it can be sent to backend.

Utility Methods for Regression Models
There are three helper methods for data pre-process, model performance and predicting future stock prices prepare_data Method will shift the stock prices by given forecast_out amount. Basically, method will create predictions n days in the future by shifting the values n days backwards. Shortly, n days earlier data will be used to predict future stock prices. After this data manipulation, method will return five different datasets to be used for training, testing and forecasting:

- x_train : Set of features (For training the model)
- x_test : Set of features (For evaluating the model)
- y_train : Set of labels (For training the model)
- y_test : Set of labels (For evaluating the model)
- x_forecast: Forecast out (It will be used to predict **n days** ahead)

get_model_performance Method will use the given Linear Regression Model that has been created and fit before, to make prediction on the test dataset and report the accuracy metrics such as variance_score, max_error_value, mea_square_error.
get_feature_predictions Method will use the given Linear Regression Model that has been created and fit before, to make predictions for future.

Utility Methods for Neural Network Model
There are two helper methods for data pre-process and creating sequences that is used in RNN implantation:

prepare_data is the main method for dividing the stock prices into pieces for training and testing the prediction model. Our models use two data sets for training the prediction model, two data sets for testing the prediction model and one for forecasting.

First, we need to divide historical stock prices to two pieces. User can configure the time interval of train and test data. Depending on the interval, method divides the entire dataset into two pieces, namely, train_data and validation_data. In order RNN to accept the input, None values must be dropped from the dataset. Then, MinMaxScaler method from sklearn library is used to scale the stock prices. Scaler object for the validation data must be saved and kept for scaling back the values after training and testing the model.

At the end, there are four datasets and one scaler object returned by the prepare_data method:

- train_x and train_y: Data sets to be used for model training.
- validation_x and validation _y: Data sets to be used for model testing.
- scaler_validation_data: Scaler object for scaling the validation values back

create_sequences method is responsible of creating sequences from stock prices by given sequence length. deque method from Python's collections library is used for creating the sequences.

SEQ_LEN represents that RNN will use SEQ_LEN days of data for its input per iteration. If users select SEQ_LEN as 30, it means that method will create sequences which has length of 30 days and feed it to RNN.

6.3.4. How to Add a New Model

New prediction models can be added to STOCK Application and can be used after couple steps. The necessary steps to add new prediction models as follows:

1. Navigate to the root folder (stock_project) and simply create a .py file either in **regression_algorithms** or **rnn_algorithms** folder depending on the model you want to create.

2. For better understanding, use the following naming convention for prediction models:

   2.1. For Regression Models: **lr_<name_of_the_model>_model.py.**
   (**e.g:** *lr_lasso_model.py)*
   2.2. For Neural Network Models:
   rnn_<name_of_the_model>_modely.py.
   (**e.g:** *rnn_lstm_model.py*)

   Shortly, convention should follow the structure:
   <model_initials>_<name_of_the_model>_model.py

3. After the implementation of new model is over, open app.py file and add the new model:

   3.1. app.py file has a dictionary named algorithms. New model must be added there. Currently dictionary has following models.
   3.2. Import the new model file to the top of the code.
   3.3. Add the new model into the dictionary and save the file.

   Refer to Snippet 63 for the example code.

```
from regression_algorithms import lr_bayesian_ridge_model,
                                  lr_default_model,
                                  lr_elastic_model,
                                  lr_lasso_lars_model,
                                  lr_lasso_model,
                                  lr_ridge_model

from regression_algorithms import decision_tree_model,
                                  random_forest_model

from rnn_algorithms import rnn_lstm_model,

from example_model import source_code_file,

algorithms = \
  {
    'lr_default_model'    : lr_default_model,
    'lr_lasso_lars_model'    : lr_lasso_lars_model,
    'lr_ridge_model'        : lr_ridge_model,
    'lr_elastic_model'      : lr_elastic_model,
    'lr_lasso_model'        : lr_lasso_model,
    'lr_bayesian_ridge_model': lr_bayesian_ridge_model,

    'decision_tree_model'    : decision_tree_model,
    'random_forest_model'    : random_forest_model,

    'rnn_lstm_model'        : rnn_lstm_model,

    'example_model'        : source_code_file,
  }
```

Snippet 63: Code Example for Adding a New Model

4. As the last step, custom Nginx and Flask Services must be restarted.
   4.1. Open up a new terminal window in the Linux Machine and type the following commands in order to restart the services:
      Restart custom Stock Service: sudo systemctl restart stock
      Restart custom Nginx Service: sudo systemctl restart nginx

Now, new model can be accessed from backend module.

6.4. Environment Configuration

Machine Learning Module of the STOCK Application runs on a dedicated Linux Server as a Linux Service. Creating future predictions might be a computationally expensive task. Separating ML Module from the other modules and putting into a dedicated server, allows project to scale up easily. In order ML Module to work following steps must be done.

6.4.1. Prerequisites

- Debian 10 Linux
  - Non-Root user account with sudo privileges.
- Python 3.6
- Anaconda
- Nginx

6.4.2. Create and Configure Virtual Environment

Copy the source code under home directory of the current user.

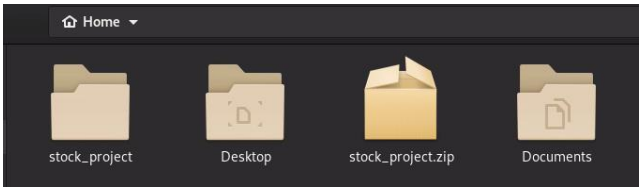

Figure 17: Linux Home Directory View

Open up a terminal window under the Home directory and check the folder permission:

```
ls -l

Output:
      drwxr-xr-x  7 kaan kaan  4096 Feb 17 07:44  stock_project
```

If your user does not have full access to project folder, use below code to gain access:

```
sudo chmod -R 755 <project_folder_name>
```

Navigate to parent folder and locate the **stock_ml_environment.yml**. Import the environment:

```
conda env create -f stock_ml_environment.yml
```

Importing the environment might take time based on the network speed. After import is done, activate the environment:

```
conda activate stock_project

Output:
      (stock_project) vm_name@vm_id:~$
```

6.4.3. Install and Configure Nginx

Open Linux Terminal and install **Nginx:**

```
sudo apt install nginx
```

Once Nginx has been installed configure the firewall setting:

```
#Allow Nginx to accept traffic on Port 80 and 443
Sudo ufw allow 'Nginx Full'
```

Verify the change by typing:

```
#sudo ufw status

Output:
       Status: active

       To                         Action      From
       --                         ------      ----
       OpenSSH                    ALLOW       Anywhere
       Nginx HTTP                 ALLOW       Anywhere
       Nginx Full                 ALLOW       Anywhere
       OpenSSH (v6)               ALLOW       Anywhere (v6)
       Nginx HTTP (v6)            ALLOW       Anywhere (v6)
       Nginx Full (v6)            ALLOW       Anywhere (v6)
```

Debian automatically starts Nginx after the initial installation. To check if the web server is up and running, use the following command:

```
syscemctl status nginx
```

When you navigate to **http://<serverIP>** browser should be showing Nginx Landing Page:

Next step is to create a server block that will use Machine Learning Code and make it available online, so backend can call prediction algorithms even from another server.

6.4.3.1. Create Server Block

Create a dedicated configuration file for STOCK Project:

```
sudo nano /etc/nginx/sites-available/stock
```

Above code will create the file and open up the nano text editor. Use the following configuration:

```
server {
  listen 80;
  server_name <server_IP>;
  uwsgi_read_timeout 600s;

  location / {
    include uwsgi_params;
    uwsgi_pass unix:/home/<user_account_name>/<project_location>/stock.sock;
    }

  location /ml {
    rewrite ^/ml/(.*) /$1 break;
    proxy_pass http://127.0.0.1:80;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP ip_address;
    proxy_set_header Connection "";
    proxy_connect_timeout 600s;
    proxy_read_timeout 600s; }
}
```

Note that there must be some changes on the configuration. Replace:

- **<server_IP>** with your server IP
- **<user_account_name>** with current user account
- **<project_location>** with the location you have copied ML Source Codes. (This path is not same as the Anaconda Virtual Environment Path)

and save the config file.

With the above steps, Nginx is configured to listen port 80 of the given server name, as well as timeout limits and re-write rule for the URL "**/ml**"

Enable the server block by creating a symbolic link to the above custom configuration inside the sites-enabled directory. This will allow Nginx the read custom configuration during startup:

```
sudo ln -s /etc/nginx/sites-available/stock /etc/nginx/sites-enabled
```

To prevent memory problems, it is required to enable server_names_hash_bucket_size configuration from Nginx configuration. Open the Nginx configuration file:

```
sudo nano /etc/nginx/nginx.conf
```

Locate the server_names_hash_bucket_size entry and remove the # symbol to uncomment the line:

```
http {
        ##
        # Basic Settings
        ##

        sendfile on;
        tcp_nopush on;
        tcp_nodelay on;
        keepalive_timeout 65;
        types_hash_max_size 2048;
        # server_tokens off;

        server_names_hash_bucket_size 64;
     }
```

Save and close the configuration file. Test the syntax of configuration files:

```
sudo nginx -t

Output:
       nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
       nginx: configuration file /etc/nginx/nginx.conf test is successful
```

Make sure the test is successful. Then, restart Nginx to enable custom configurations:

```
sudo systemctl restart nginx
```

6.4.3.2. Setup a Flask Application

Activate the virtual environment:

```
conda activate stock_project

Output:
       (stock_project) vm_name@vm_id:~$
```

Install wheel to make sure pip will install necessary packages even though they are missing wheel archives.

```
(stock_project) $ pip install uwsgi flask
```

When the installation is over, deactivate virtual environment.

```
(stock_project) $ deactivate
```

It is necessary to create a custom configuration file for uWSGI so at the end we can have Flask Application to work as Linux Service.

First necessary file is already present under the project folder. Navigate to project folder and locate the file named stock.ini
This file contains custom settings for uWSGI:

```
# Header
  [uwsgi]

  # module_name:flask_app_name
  module = wsgi:ml

  # Start uWSGI in master mode and spawn 6 worker processes
  master = true
  processes = 6

  # Create a Unix Socket named "stock.sock" and set the permissions
  socket = stock.sock
  chmod-socket = 660

  # Clean up the socket when the process stops
  vacuum = true

  # die-on-term option helps system and uWSGI to communicate and understand
process signals.
  die-on-term = truesudo nano
```

Before finalizing the configurations, we will let Debian to automatically start uWSGI and serve the Flask Application when server starts.

Open up a terminal window and create a file named **`stock.service:`**

```
sudo nano /etc/system/system/stock.service
```

Copy below configuration into stock.service file:

```
[Unit]
Description=uWSGI instance to serve Stock Project
After=network.target

[Service]
User=<user_name>
Group=www-data
WorkingDirectory=/home/<user_name>/stock_project
Environment="PATH=/home/<user_name>/anaconda3/envs/stock_project/bin"
ExecStart=/home/<user_name>/anaconda3/envs/stock_project/bin/uwsgi --ini
stock.ini

[Install]
WantedBy=multi-user.target
```

Note that there must be some changes on the configuration. Replace;

- **<user_name>** with current user account.
- **Environment=** must point to the bin folder of Virtual Environment that contains Stock Project.
- **ExecStart=**must point to the uwsgi folder of Virtual Environment that contains Stock Project.

and save the config file.

```
[Unit]
# Unit Section specify metadata nad dependencies
Description=uWSGI instance to serve Stock Project
# Start the unit after the networking target has been reached.
After=network.target

[Service]
# Service Section will specifiy the user and group that we will be using for
the process.
User=<user_name>
Group=www-data

# Set the project paths.
# WorkingDirecory must point to the source code directory
WorkingDirectory=/home/<user_name>/stock_project

# Environemnt and ExecStart must point to the anaconda virtual environment
directory
Environment="PATH=/home/<user_name>/anaconda3/envs/stock_project/bin"
ExecStart=/home/<user_name>/anaconda3/envs/stock_project/bin/uwsgi --ini
stock.ini

[Install]
# Run the service when multi-user system is up and running
WantedBy=multi-user.target
```

Start the custom uWSGI Service:

```
sudo systemctl start stock
```

Enable the service to be started automatically after the boot:

```
sudo systemctl enable stock
```

Check the status of the service:

```
sudo systemctl status stock
```

Now, Machine Learning Code can be used by the backend module of the project throughout the Flash Application which runs as a Linux Service.

Finally, run the custom services:

```
sudo systemctl start nginx
sudo systemctl start stock
```

7. Integrations

7.1. MongoDB

The database used in the STOCK application is MongoDB. Firstly, we decided to use DynamoDB, but it is a paid product and the free version only allowed up to 25 Read/Write operations per second and although it allows higher throughput in short bursts, it is a limitation that does not meet the requirements of the application. MongoDB opened new possibilities for us regarding multi-document transactions across replicas. We are able to manipulate data in a safer way ensuring ACID properties of our operations that behave in a transaction manner. MongoDB ensures optimistic concurrency on global, database and collection level and we had to only implement it at the document level using versioning. We currently have 3 replicas running.

If you want to run it locally or for testing purposes, to create replicas easily as they are required for multi-document transactions it can be easily done using the Node package run-rs [36] which is a zero configuration MongoDB runner that starts a replica set with no non-Node dependencies, not even MongoDB.

Currently, our database is on MongoDB Atlas, which is a global cloud database service for modern applications. [37] There are also 3 replicas running to ensure the optimistic concurrency.

7.2. Amazon Cognito

As an identity provider, we decided to use an external service such as Amazon Cognito, because it implements OAuth 2 and OpenID Connect specifications and furthermore it is free to use for up to 50,000 Monthly Authenticated Users (MAU) which is enough for this implementation stage of the STOCK application. Amazon Cognito provides user pools as well as federated identities with different sets of configurations and is flexible to further changes. We used user pools with required email verification during registration, password reset and password change actions. Verification is managed by using the verification code sent to the user's email in the corresponding actions.

7.3. IEX Cloud

IEX Cloud [38] is a commercial product that provides various types of financial data through its APIs. Depending on the subscription you have, you can get access to different types of APIs (historical data, current data, real-time data, data streams, etc.)

In our system we need stock, company and article information. Initially, our decision was to build scrapers and for some time we relied on them. We were scraping articles from MarketWatch , CNBC and similar sources but then a lot of these sites introduced analyzers that block the requests when they detect unusual traffic from some IP address. Also, some of them only allow you to get only a few articles to read for free and once you reach the limit, they offer you different types of subscriptions to access the rest of their data. Because of that, we were forced to switch from our scrapers to a more reliable (paid) source where we can constantly fetch new data. Now we basically have to think only how much data monthly can we consume and this is optimized in our background jobs described in Section 4.3. We are also able to get more data from various sources as well as historical stock prices as far as from 2003.

7.4. Sentry

For any kind of system, application monitoring is a necessity. For smaller applications where few events happen during the day it is sufficient to log errors in a file on the machine. However, when working on larger systems where events happen constantly and especially when you have background jobs running, monitoring and error tracking is important for building and maintaining the application.

For this we decided to use Sentry [39]. It offers a free plan with some limitations. There is a limit of number of events that can be stored and they are persisted for a month only. While building the STOCK application, this was particularly helpful during development of the front end application as some parts of the backend application were built but not yet properly tested, so we were able to discover some unusual behaviors and bugs.

When there is an error that we want to log or an unhandled exception that wasn't expected or prevented, it is immediately logged in Sentry and there is an email notification being sent. It is possible to filter issues by a prebuilt filter or even type them directly in the search bar and also filter by date, importance, environment, number of events etc. An overview of the Sentry dashboard is in Figure 24.

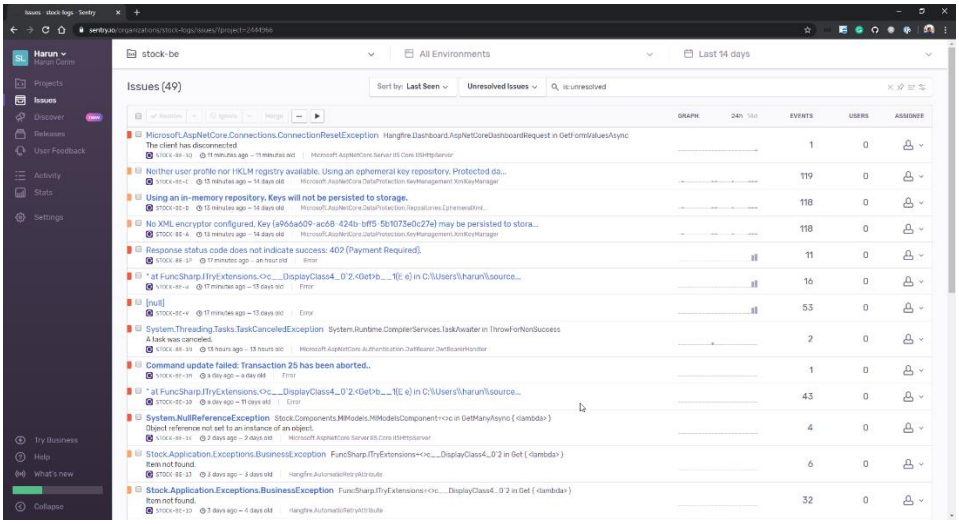

Figure 18: Sentry dashboard

8. Future Plans

While we tried to implement as many features as possible within the allowed time period, we realize that there are some limitations to this application which can be further solved by future implementations, also keeping in mind the possibility of the application going commercial.

For example, right now in the Company profile page, the shown predictions use the default RNN model. In the future we could let the user choose their own default model to be shown. Also, they are only able to create one model per company at a time, we could also allow them to create the same model for more than one company by selecting multiple companies instead of one during model creation.

Currently articles are not involved in stock predictions. One way to involve articles for Recurrent Neural Network predictions would be to encode articles in such a way that RNN can accept and use it together with current stock features. This requires strong data analysis like selecting important articles and finding important data that there exists a strong relation with published articles and stock changes.

The user has already the possibility to export a model to PDF and download it. However, for more advanced users we could offer the opportunity to download a binary file of the model they created so they can use it in their machines and run the code themselves.

On the more technical side, having background jobs supporting our application, we couldn't build a serverless infrastructure. However, in the future the application can be split into server and serverless units by keeping the background jobs in the server and migrating existing APIs to a serverless technology such as AWS Lambda.

Bibliography

[1] "Market Watch," [Online]. Available: https://www.marketwatch.com/. [Accessed January 2019].

[2] "WalletInvestor," [Online]. Available: https://walletinvestor.com/. [Accessed January 2019].

[3] "Wallmine," Wallmine, [Online]. Available: https://wallmine.com/. [Accessed 23 08 2019].

[4] "yarn," [Online]. Available: https://yarnpkg.com/. [Accessed 1 June 2019].

[5] "npm," [Online]. Available: https://www.npmjs.com/. [Accessed 1 June 2019].

[6] "Docker," [Online]. Available: https://www.docker.com/. [Accessed December 2020].

[7] "ReactJS," [Online]. Available: https://reactjs.org/. [Accessed 1 June 2019].

[8] "Material UI," [Online]. Available: https://material-ui.com/. [Accessed 1 June 2019].

[9] "Axios," [Online]. Available: https://github.com/axios/axios. [Accessed 15 July 2019].

[10] "react-localization," [Online]. Available: https://www.npmjs.com/package/react-localization. [Accessed 15 October 2019].

[11] "jest," [Online]. Available: https://jestjs.io/. [Accessed 5 January 2020].

[12] "Enzyme," [Online]. Available: https://enzymejs.github.io/enzyme/. [Accessed 5 January 2020].

[13] "Azure," Microsoft, [Online]. Available: https://azure.microsoft.com/en-us/services/app-service/.. [Accessed 14 02 2020].

[14] R. C. Martin, Clean Architecture, Prentice Hall, 2017.

[15] "REST API," [Online]. Available: https://restfulapi.net/. [Accessed 15 March 2019].

[16] "Test Driven Development," [Online]. Available: https://martinfowler.com/bliki/TestDrivenDevelopment.html. [Accessed 20 March 2019].

[17] "CRUD operations," [Online]. Available: https://stackify.com/what-are-crud-operations/. [Accessed 20 November 2019].

[18] "Knowing Monads Through The Category Theory," [Online]. Available: https://dev.to/juaneto/knowing-monads-through-the-category-theory-1mea. [Accessed 5 January 2020].

[19] "What does ACID mean in Database Systems?," [Online]. Available: https://database.guide/what-is-acid-in-databases/. [Accessed 15 March 2019].

[20] "Vader Sharp," [Online]. Available: https://github.com/codingupastorm/vadersharp. [Accessed 20 July 2019].

[21] E. G. C.J. Hutto, "VADER: A Parsimonious Rule-based Model for Sentiment Analysis of Social Media Text," in *EIGHTH INTERNATIONAL CONFERENCE ON WEBLOGS AND SOCIAL MEDIA*, 2014.

[22] "Unit of Work," [Online]. Available: https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-

4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application. [Accessed October 2019].

[23] "Swagger," [Online]. Available: https://swagger.io/. [Accessed 18 10 2019].

[24] "OpenAPI," [Online]. Available: https://www.openapis.org/. [Accessed 18 10 2019].

[25] "scikit-learn," [Online]. Available: https://scikit-learn.org/stable/. [Accessed 23 01 2019].

[26] "Decision Tree," [Online]. Available: https://scikit-learn.org/stable/modules/tree.html. [Accessed 10 08 2019].

[27] A. C. Leo Breiman, "Random Forests," [Online]. Available: https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm. [Accessed 12 08 2019].

[28] "Linear Regression," [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html. [Accessed 22 08 2019].

[29] "Bayesian Ridge," [Online]. Available: https://scikit-learn.org/stable/auto_examples/linear_model/plot_bayesian_ridge.html.

[30] "Lasso Lars," [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LassoLars.html. [Accessed 30 08 2019].

[31] "Ridge Regression," [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html. [Accessed 09 09 2019].

[32] A. Sherstinsky, "Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network," 2018.

[33] "Many to One RNN with Variable Sequence Length," [Online]. Available: http://www.easy-tensorflow.com/tf-tutorials/recurrent-neural-networks/many-to-one-with-variable-sequence-length. [Accessed 10 12 2019].

[34] "Plot All Scaling," [Online]. Available: https://scikit-learn.org/stable/auto_examples/preprocessing/plot_all_scaling.html#maxabsscaler. [Accessed 20 01 2020].

[35] "Pandas," [Online]. Available: https://pandas.pydata.org. [Accessed 29 07 2019].

[36] "run-rs," [Online]. Available: https://www.npmjs.com/package/run-rs. [Accessed 15 October 2019].

[37] "MongoDB Atlas," [Online]. Available: https://www.mongodb.com/cloud/atlas. [Accessed January 2020].

[38] "IEX Cloud," [Online]. Available: https://iexcloud.io/. [Accessed 25 October 2018].

[39] "Sentry," [Online]. Available: https://sentry.io/welcome/. [Accessed 20 January 2019].

[40] ".NET core," [Online]. Available: https://dotnet.microsoft.com/. [Accessed 15 December 2018].

[41] "Chartjs," [Online]. Available: https://www.chartjs.org/. [Accessed 1 June 2019].

[42] "Fuse Theme," [Online]. Available: http://fusetheme.com/. [Accessed 20 March 2019].

[43] "Redux," [Online]. Available: https://redux.js.org/. [Accessed 15 March 2019].

[44] "React Redux," [Online]. Available: https://react-redux.js.org/. [Accessed 15 March 2019].

[45] "Redux Thunk," [Online]. Available: https://www.npmjs.com/package/redux-thunk. [Accessed 20 November 2019].

[46] "Market watch," [Online]. Available: https://www.marketwatch.com/. [Accessed January 2019].