



Engenharia e Dados e Conhecimento

Bolsa Client - TP1

Final Report

November 12, 2018

By:

Davide Cruz - 71776,
João Aniceto 72255 ,
Javier Borrallo Fernández 92092



Table of contents

Table of contents	2
Introduction	3
Data and Sources	3
Data Schemas (XML Schema)	4
Data Transformations (XSLT)	7
Operations over the data (XQuery and XQuery Update)	8
UI Functionality	17
Additional Considerations	24
Compromises	24
Layout System	24
Custom template tags	25
Django Authentication System	25
Division of views and urls	26
Coin Selector	26
Frameworks used	26
Conclusions	26
Configuration to execute the application	27



Introduction

The objective of this project was to simulate a stock market analysis and transactions *API*.

Given a *XML* database containing several companies, their stock price and other relevant information, a visitor can see the variations of their stocks in different intervals of time, and also real rss feeds of news related to the companies (if these happen to be real of course).

The visitor is given the option to become a user by registering.

By becoming a user a person can buy and sell stock of any company they choose given that they have the necessary funds, building their own personal portfolio.

If the user happens to be a superuser they can both export and alter the database using the proper xml code.

Data and Sources

The project contains 2 databases:

1. *SQLite* database which is used only for registering, authentication and user management purposes (deleting and granting/removing permissions). This is the Django default database. This way we don't have a major concern regarding the security of the authentication system (storing passwords properly) and we can use all the features it provides.
2. *XML* database used for almost everything else in the website, companies stock price, logos, information, history, and also all the user portfolio informations (wallet, money, history of buys/sells).

The only feature that doesn't use any database is the news rss feed. However it does need the company symbol stored in the xml database. The *RSS* feed is dynamic (different for every company) and real (if the company symbol exists). Its source is the yahoo finance *RSS* feed.

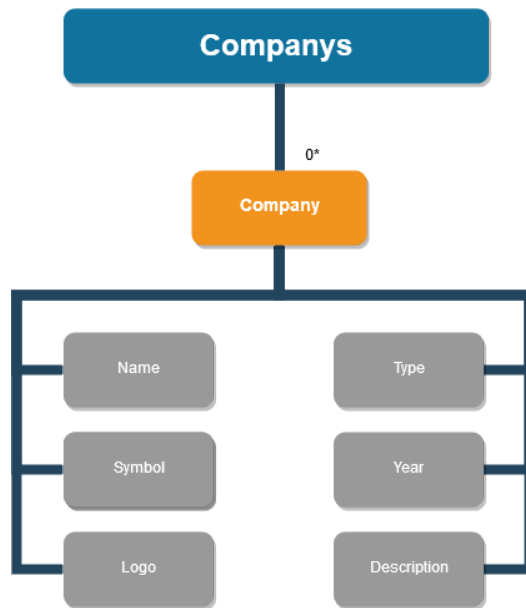
The data that is currently stored by default is dummy data created by us. Although it is based on real values it is not accurate as the market is always changing. Given that it's hard to get stock market data we made the compromise that the system includes values on a daily basis. This means that each stock (company) has a value for each day.



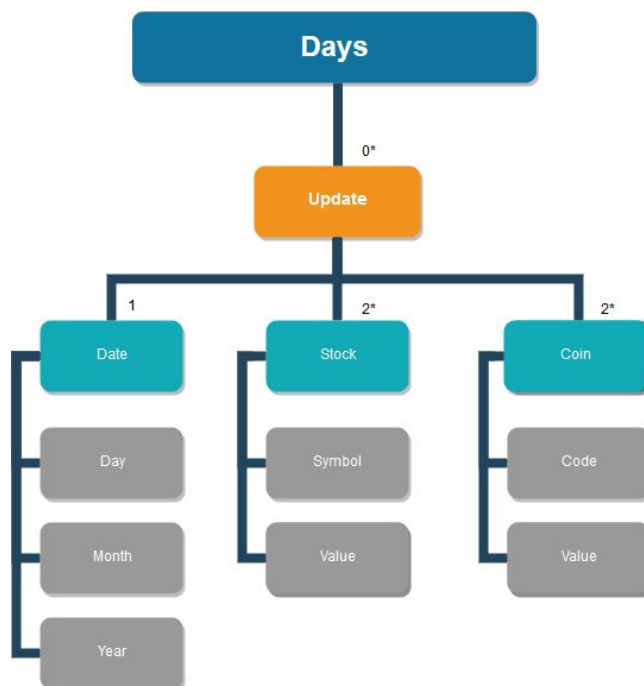
Data Schemas (XML Schema)

The project contains 4 XML:

1. Companies.xml: Is used to store Company data, as the name, symbol, icon...

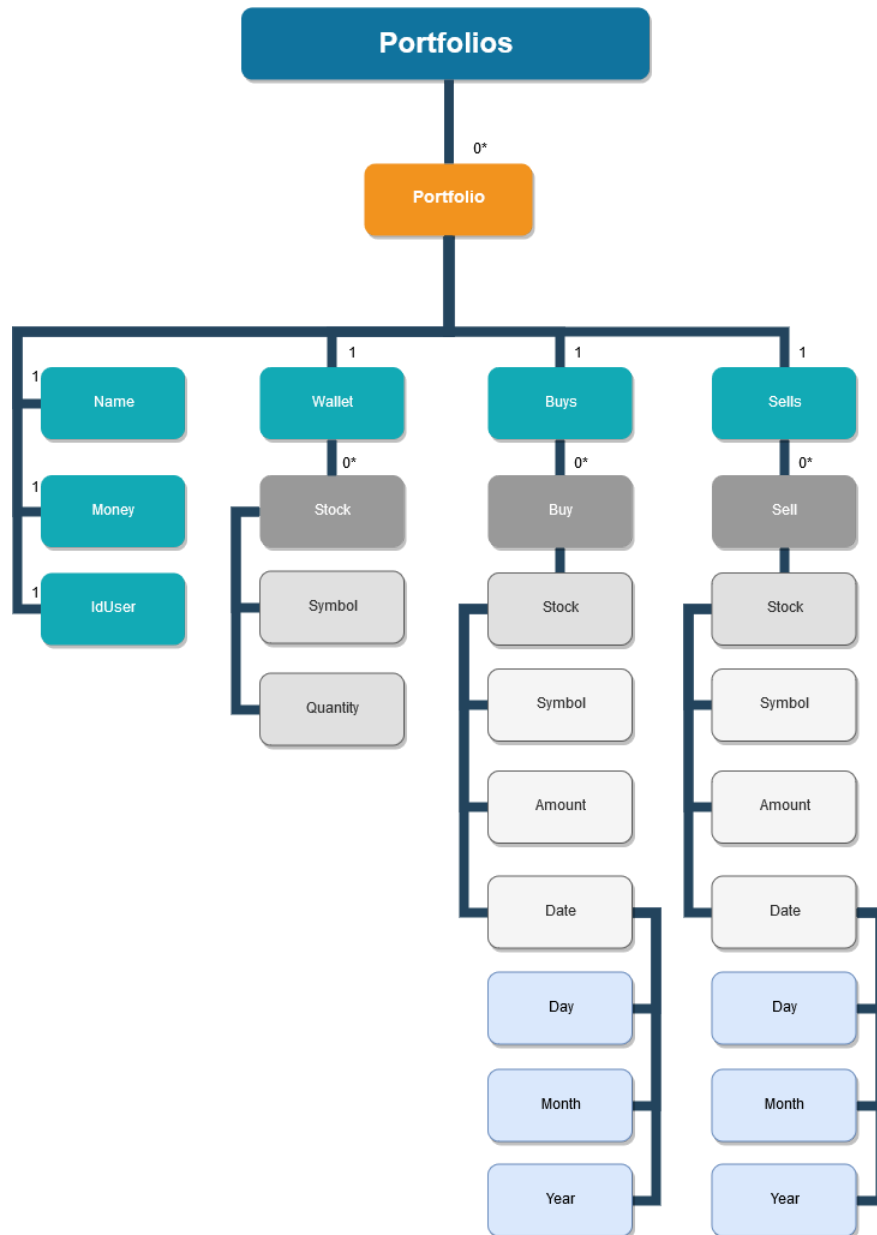


2. Days.xml: This xml is used to store a date and the value of each company that day.



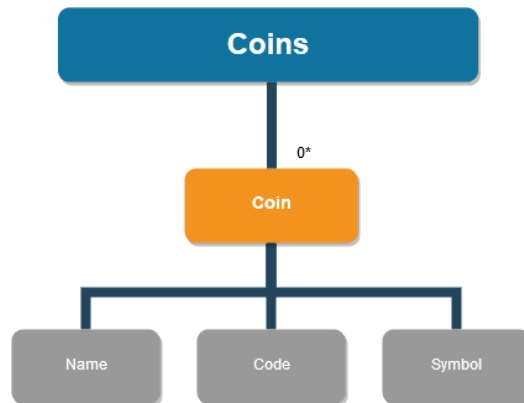


3. Portefolios.xml: In this *XML* we find the basic information related to the user, the company stocks it has, the quantity and the value of the stocks.





4. Coins.xml: Coin's value according to dollar value is stored in this xml.



All these files have schemas that validate them accordingly that have the same name but end with .xsd extension.

We also have additional schemas to validate data introduced in the Admin menu:

1. Update.xsd: Validates an insertion of an update (day) in the database
2. Company.xsd: Validates an insertion of a company in the database

These last schemas are validated upon submission using lxml.

Data Transformations (XSLT)

We use XSLT transformations on the following pages:

- Companies page
- Coins page

The files *coins.xslt* and *companies.xslt* both transform the result of the query to the BaseX database into a normal html table that is the passed as context to the respective django template. The content is rendered with safe option so that Django template render system doesn't escape the html symbols/elements. Given our layout system that tries to make full use of Django template inheritance (extends, includes and blocks) it wouldn't be feasible to create the whole page using XSLT. This provides extended flexibility to create or modify the base structure of the pages.

For the rest of the pages we parse the data returned by the query to the BaseX database with xmldict. We then pass it as context to the template system with minimal or some pre-processing. This pre-processing is to facilitate the task of passing the data to the graph mostly.



Operations over the data (XQuery and XQuery Update)

Throughout the project we used 19 queries, which of whom 6 are XQuery Update.

Since we use them in views lets enumerate them in same.

The '{}' string will appear in most queries, that string is replaced by necessary attributes or clauses before the query is made and if not obvious the context will be explained.

❖ Query used in *home view* found in *views/bolsaViews.py* :

This query is quite simple, we need the last two updates so we read them from the database and coupled them together in a tuple.

```
let $x := db:open("Bolsa")//days/update[last()]
let $y := db:open("Bolsa")//days/update[last()-1]
return ($x,$y)
```

❖ Query used in *valuesToday view* found in *views/bolsaViews.py* :

We wanted to create a table that showed the name, symbol and last value of a company stock.

For this purpose we needed to extract those same elements from the database and concatenate them in a meaningful way.

The name and value are in two different tables/xml files in the database, however the symbol is common in both.

The first 'for' iterates the necessary values and the second the names. Using the symbol as the common attribute we then return a xml element <stock> with all the information.

In our project we offer a search feature where you can look up companies by name or symbol, the back end of its implementation its made in this same query as seen in the second part of the where clause.

```
for $x in db:open("Bolsa")//days/update[last()]/stock
for $y in db:open("Bolsa")//companys/company
where $x/symbol = $y/symbol
and
(contains(lower-case($y/symbol/text()),lower-case('{}')) or
contains(lower-case($y/name/text()),lower-case('{}')))
return <stock>{$y/name,$x/symbol,$x/value}</stock>
```



❖ **Queries used in *valuesFilter* view found in *views/bolsaViews.py* :**

all the companies values from all the updates, we extracted this data from the following query: In the search and filter web page, accessed from the values subsection in the site navbar we show the user a time/value graph of all the companies from every update that we have.

We also give the user the option to choose the time interval and companies that he wishes to see in the graph.

In order to implement these features we required several accesses to the database data, we divided said accesses into 2 queries.

For the initial graph we needed the first and last dates present in the database, and also

```
let $dateFirst := db:open("Bolsa")//days/update[1]/date
return <firstDate>{$dateFirst}</firstDate>,
let $dateLast := db:open("Bolsa")//days/update[last()]/date
return <lastDate>{$dateLast}</lastDate>,
let $companies := db:open("Bolsa")/companys//company//symbol
return <companies>{$companies}</companies>,
for $x in db:open("Bolsa")//days/update
let $y := $x/stock
return <update>{$x/date,$y}</update>
```

Next we needed a query that allowed a more specific access to the data, in relation to the requirements of the user.

```
let $companies := db:open("Bolsa")/companys//company//symbol
return <companies>{{ $companies }}</companies>,
let $dateFirst := db:open("Bolsa")//days/update[1]/date
return <firstDate>{{ $dateFirst }}</firstDate>,
let $dateLast := db:open("Bolsa")//days/update[last()]/date
return <lastDate>{{ $dateLast }}</lastDate>,
for $x1 in db:open("Bolsa")//days/update
let $y1 := db:open("Bolsa")//days/update
where $x1/date/day/text() = "{}" and $x1/date/month/text() = "{}" and
$x1/date/year/text() = "{}"
return
  let $pos1 := index-of($y1,$x1)
  return
    for $x2 in db:open("Bolsa")//days/update
    where $x2/date/day/text() = "{}" and $x2/date/month/text() = "{}"
    and $x2/date/year/text() = "{}"
    return
      let $pos2 := index-of($y1,$x2)
      for $a in db:open("Bolsa")//days/update[position() >= $pos1 and
not(position() > $pos2)]
```




```

return <update>
  {{$a/date,
  for $stock in $a/stock
  {}
  return $stock
  }}
</update>

```

The last '{}' string after the for loop represents the following where clause that depends on the number of companies that the user selects:

```

while i < len(companyList):
    if i == 0:
        companyCommand += 'where $stock/symbol/text() =
    "{}{}\".format(companyList[i])
    else:
        companyCommand += 'or $stock/symbol/text() =
    "{}{}\".format(companyList[i])
    i += 1

```

❖ Query used in *valuesCoins* view found in *views/bolsaViews.py*:

This query is made to get all the coins and their most recent values. It returns in a stock format with the name, symbol and value.

We need to access two different XML documents and then join them together.

The query also features a where statement that is used when we search coins. The query tries to match by name or symbol and is not case-sensitive as we lowercase both the item to compare with and the search query.

If the search option is blank the query matches all the coins. That is the default behaviour when a user reaches the page.

```

for $x in db:open("Bolsa")//days/update[last()]/coin
for $y in db:open("Bolsa")//coins/coin
where $x/code = $y/code
and
(contains(lower-case($y/name/text()),lower-case('{}'))
or contains(lower-case($y/code/text()),lower-case('{}'))
or contains($y/symbol/text(),'{}'))
return <stock>{{$y/name,$y/symbol,$x/value}}</stock>

```



◆ **Query used in *companies* view found in *views/bolsaViews.py* :**

```
for $x in db:open("Bolsa")//companys/company
{}
return <company>{{ $x/name, $x/symbol }}</company>
```

This query is very simple. It returns all the companies name and respective symbol that we then show in a table.

Similar to the one above if the user searches the companies this query is modified to have a where statement (with contains) which filters by name or symbol.

◆ **Queries used in *companiesInfo* view found in *views/bolsaViews.py* :**

On the company info page we have the option to buy stock. This query handles all the logic associated with it. One drawback of this approach is that we have no way to check if the operation went through as XQuery Update expression can't be mixed with normal ones (they have to return nothing or insert/delete/update)¹. The only way to give feedback to the user is to get his portfolio before and after the operation and see if it changed. That is obviously not an ok approach as we would have tripled our queries to the database.

Update: At the time of writing this report we found out that in BaseX, it is in fact possible to returning something even when using XQuery Update². This was previously unknown to us, and if had this information we would have added feedback to both the buy and sell, given that this in fact works.

The advantage is that given the objective of the project we learn a lot about XQuery and be ready to do both server and database checks when needed.

First, the query get the user money and checks if he has enough money to buy the stock at the current price. If he does, it checks if he already has the stock in his wallet. If he does it increments the number, else it inserts the stock in his wallet. Finally it adds the transaction to his buy history.

```
for $y in db:open("Bolsa")//portfolios/portfolio
where $y/idUser = "{id}"
return
let $a := db:open("Bolsa")//days/update[last()]
for $stock in $a/stock
where $stock/symbol/text() = "{c}"
return if($y/money/text() > $stock/value/text()*{q}) then
  (replace node $y/money/text() with
  $y/money/text()-$stock/value/text()*{q},
```

¹<http://www.xmlmind.com/tutorials/XQueryUpdate/index.html#d0e696>

²http://docs.basex.org/wiki/XQuery_Update#Returning_Results



```
let $userStock := $y/wallet/stock
return(
  if($userStock/symbol/text() = "{c}") then
    for $c in $userStock
    where $c/symbol/text() = "{c}"
    return replace node $c/quantity/text() with $c/quantity/text()+{q}
  else
    insert node
    <stock><symbol>{c}</symbol><quantity>{q}</quantity></stock> into $y/wallet
),
let $b := $y/buys
return insert node
<buy><stock>{{ $stock/symbol, $a/date }}<amount>{q}</amount></stock></buy>
into $b
)
else
  ()
```

This query is the default behaviour of the view and is done everytime the page is loaded. It gets all the company info from the database and then it get all the updates it has on it.

```
let $symbol := '{}'
for $y in db:open("Bolsa")//companys/company
where $y/symbol/text() = $symbol
return $y,
let $symbol := '{}'
let $x := db:open("Bolsa")//days/update
where $x/stock/symbol/text() = $symbol
return $x
```

❖ Queries used in *portfolio* view found in *views/bolsaViews.py*:

For the portfolio we needed to extract all of the current user portfolio information and also since the functionalities of the website depend of the user's money we needed to we give the option do change the current money in the wallet, which immediately inserts that quantity in the database.

We do these actions with the following queries:

```
for $y in db:open("Bolsa")//portfolios/portfolio
where $y/idUser = "{}"
return replace node $y/money/text() with $y/money/text()+{}

for $y in db:open("Bolsa")//portfolios/portfolio
where $y/idUser = "{}"
return $y
```



❖ **Query used in *register* view found in *views/bolsaViews.py* :**

Query used when registering a user. It inserts a new portfolio (at the end by default) for a new user with no money. The id is given by Django and is unique for each user created. The name is given by the user in the form to register.

```
for $a in db:open("Bolsa")//portfolios
return insert node
  <portfolio>
    <idUser>{</idUser>
    <name>{</name>
    <money>0</money>
    <wallet></wallet>
    <buys></buys>
    <sells></sells>
  </portfolio>
into $a
```

❖ **Queries used in *sell* view found in *views/bolsaViews.py* :**

This is this most complicated query in our work.

Similarly to the buy query, it first gets the user portfolio. It then check if the user actually has the quantity of stock he wanted to sell in his wallet.

If the user has more stock than he wants to sell, the query simply update the number. Else it deletes the node as it's pointless to have a entry with value 0.

Now we need to update the money. The query dynamically gets the latest value and calculates the value to be added to his money.

Finally it adds the transaction to his sell history.

```
for $y in db:open("Bolsa")//portfolios/portfolio
where $y/idUser = "{id}"
return
for $stock in $y/wallet/stock
where $stock/symbol/text() = "{c}"
return
if ($stock/quantity/text()-{q} > 0) then
(
  replace node $stock/quantity/text() with $stock/quantity/text()-{q},
  let $a := db:open("Bolsa")//days/update[last()]
  for $stock in $a/stock
  where $stock/symbol/text() = "{c}"
  return
```



```
    let $price := $stock/value
    return replace node $y/money/text() with $y/money/text()+$price*{q},
    let $b := $y/sells
    let $a := db:open("Bolsa")//days/update[last()]
    return insert node
<sell><stock>{{{$stock/symbol,$a/date}}<amount>{q}</amount></stock></sell>
into $b
)
else if ($stock/quantity/text()-{q} = 0) then
(
    delete node $stock,
    let $a := db:open("Bolsa")//days/update[last()]
    for $stock in $a/stock
    where $stock/symbol/text() = "{c}"
    return
        let $price := $stock/value
        return replace node $y/money/text() with $y/money/text()+$price*{q},
        let $b := $y/sells
        let $a := db:open("Bolsa")//days/update[last()]
        return insert node
        <sell><stock>{{{$stock/symbol,$a/date}}<amount>{q}</amount></stock></sell>
        into $b
    )
else()
```

This query is the default behaviour of the view and is done everytime the page is loaded. It gets the user portfolio so we can show his wallet and money on the page.

```
let $idUser := '""' + str(current_user.id) + '""'
for $y in db:open("Bolsa")//portfolios/portfolio
where $y/idUser = $idUser
return $y
```

❖ Query used in *rssFeed* view found in *views/bolsaViews.py*:

When we implemented the rss feed we found a good idea to show that same company info in the same page, for that we used a simple search query.

```
let $symbol := '""' + symbol.upper() + '""'
for $x in db:open("Bolsa")//companys/company
where $x/symbol/text() = $symbol
return $x
```



❖ **Query used in *get_stock_current_value* view found in *views/bolsaViews.py* :**

In several views we needed the last value of a certain company so we created a query that by giving their symbol it returns that necessary information.

```
let $symbol := '""' + symbol.upper() + '""'
for $x in db:open("Bolsa")//days/update
where $x/stock/symbol/text() = $symbol
return $x
```

❖ **Query used in *seeFull* view found in *views/bolsaViews.py* :**

The initial wallet/buys/sells tables in the portfolio html page are limited to 3 items to keep the UI user friendly and organized.

```
let $idUser := '""' + str(current_user.id) + '""'
for $y in db:open("Bolsa")//portfolios/portfolio
where $y/idUser = $idUser
return $y
```

❖ **Query used in *selectCoin* view found in *views/bolsaViews.py* :**

In this view we needed only a simple search query to verify that a certain coin exists, hence why the return value seems a bit strange.

```
for $a in db:open("Bolsa")/coins/coin
where $a/code = "{}"
return $a/code/text()
```

❖ **Query used in *adminXmlSubmitUpdate* view found in *views/adminXML.py* :**

In order to insert a new update in the database we created a insert query that modifies the *updates* table and inserts the new node at the end of it. Since the user submits xml code and its validate previously this query inserts the text directly.

```
let $a := db:open("Bolsa")/days
return insert node {} as last into $a
```



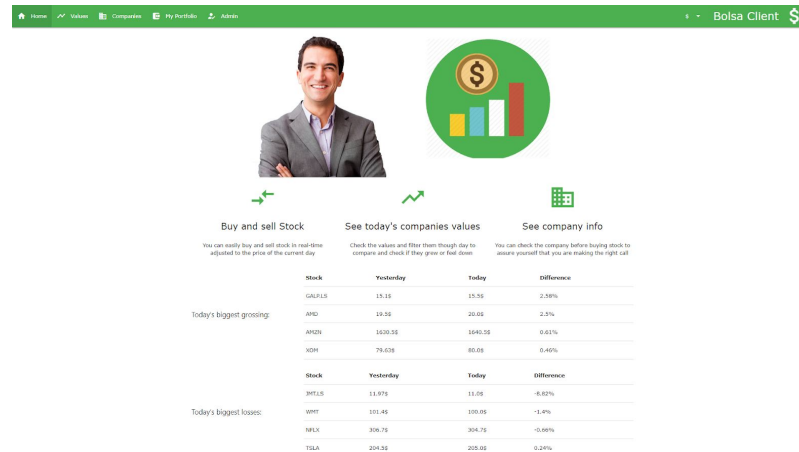
❖ **Query used in *adminXmlSubmitCompany* view found in *views/bolsaViews.py* :**

In order to insert new companies in the database we created a simple insert query that modifies the *companys* table and inserts the new node at the end of it. The '{}' represent the relevant values given by the user.

```
let $a := db:open("Bolsa")/companys
return insert node <company>
<name>{}</name><symbol>{}</symbol><logo>{}</logo><type>{}</type><year>{}</y
ear><description>{}</description>
</company> as last into $a
```

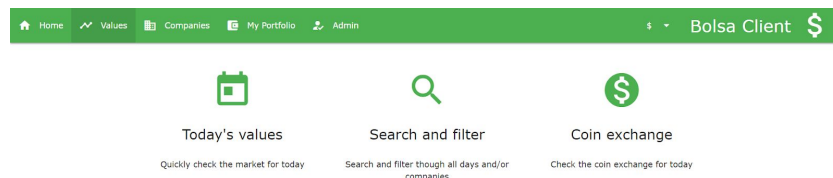


UI Functionality



Main Page.

In the main page, we can see in the upper part all the different pages that can be accessed, we also have two tables where we can find information about the companies that have gotten a bigger grow and the one which has had a lower one, or losses. This tables can be ordered alphabetically by symbol, by value or by difference between values



Values Page.

Values page links us two three other pages, “Today’s values, “Search and filter” and “Coin exchange”

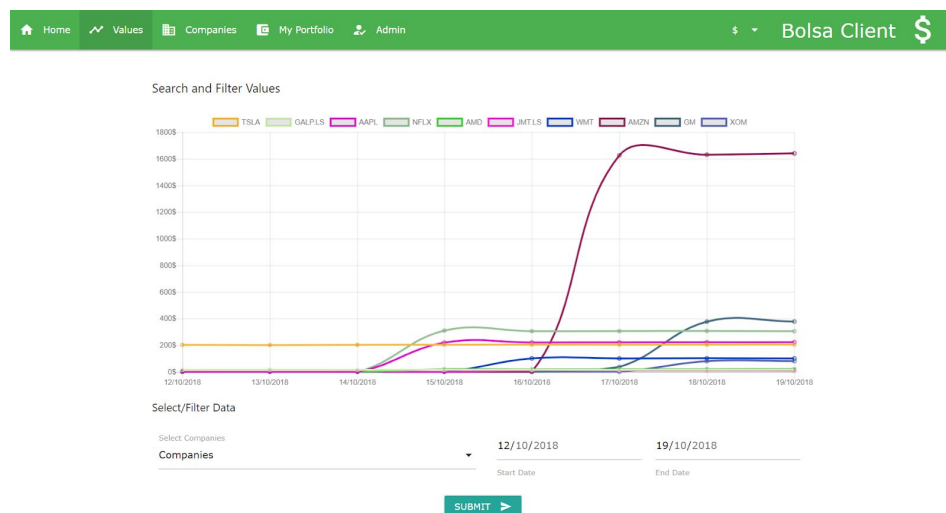


Company	Symbol	Price	
Tesla	TSLA	205.0\$	BUY
Galp Energia	GALPLS	15.5\$	BUY
Apple Inc.	AAPL	222.8\$	BUY
Netflix Inc.	NFLX	304.7\$	BUY
Advanced Micro Devices Inc.	AMD	20.0\$	BUY
Jerónimo Martins	JMTLS	11.0\$	BUY
Walmart Inc.	WMT	100.0\$	BUY
Amazon.com, Inc.	AMZN	1640.5\$	BUY

Today's values.

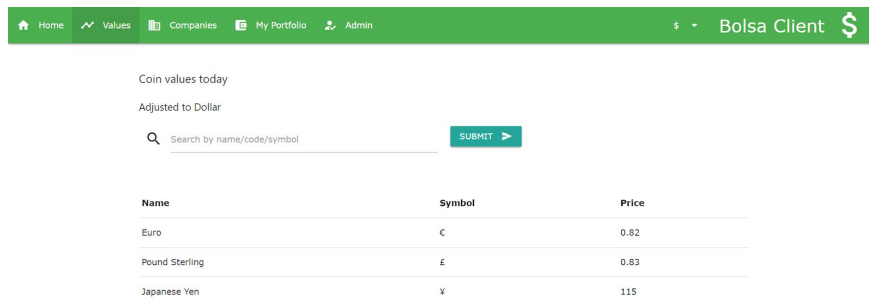
In this page we see a chart with all the companies, their symbol and the last value they had, this table can also be sorted by Company name, Symbol or its value. If we click in the company name, we will be redirected to the company page.

We can also use the search bar in order to look for a certain company, it searches by company name or symbol.



Search and filter

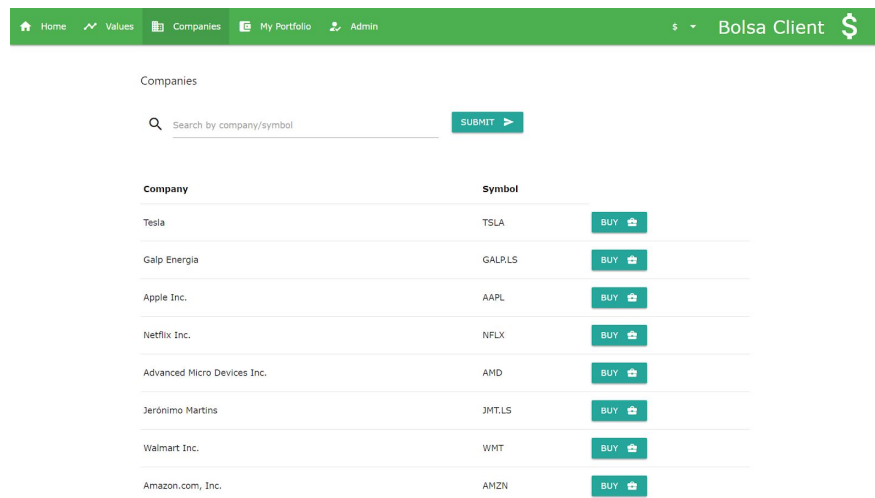
This page shows a graph of all the companies' evolution. Using the filter, we can choose which companies we would like to be shown and between which dates.



Name	Symbol	Price
Euro	C	0.82
Pound Sterling	£	0.83
Japanese Yen	¥	115

Coin values page.

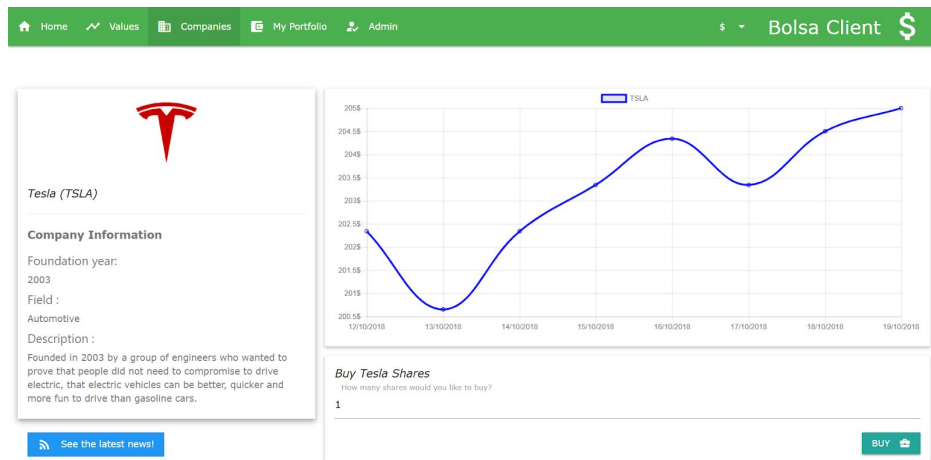
In this page we can see which is the value for Yens, Euro and Pound Sterling, adjusted to Dollar, the table can be sorted and using the search function we can search for just one coin.



Company	Symbol	
Tesla	TSLA	BUY
Galp Energia	GALPLS	BUY
Apple Inc.	AAPL	BUY
Netflix Inc.	NFLX	BUY
Advanced Micro Devices Inc.	AMD	BUY
Jerónimo Martins	JMT.LS	BUY
Walmart Inc.	WMT	BUY
Amazon.com, Inc.	AMZN	BUY

Companies Page.

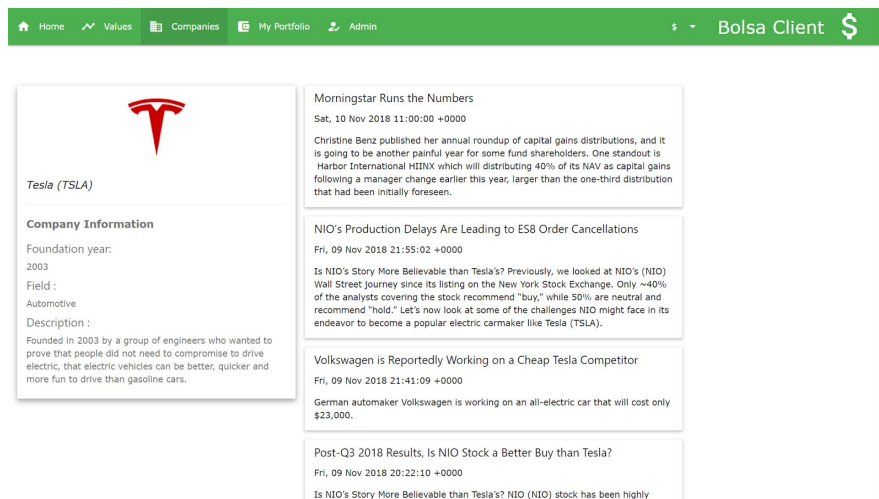
In this page we find a table with all the companies and their symbol, it can be ordered by company name or symbol and we can use the search function in order to look for a certain company, if we click on the company name or in “buy” button, we will be redirected to the company page.



Certain Company page.

Here we get the company info obtained from the XML using the company symbol. We also see a graph where we can see the company evolution and a function that allows us, if we are registered and have money, to buy shares of the company.

In this page we find the RSS button, that will show us an Rss feed of the company we have acceded to, as we can see in the next image:



Rss Feed.

Now we go to the portfolio tab, this tab will show a login page if we have not logged-in or an user page.



[Home](#) [Values](#) [Companies](#) [My Portfolio](#) \$ ▾ Bolsa Client \$

Login

Username:

Password:

SUBMIT

Don't have an account?

Click here to register

Login page.

The login page will allow us to enter in our account or to create a new one.

[Home](#) [Values](#) [Companies](#) [My Portfolio](#) \$ ▾ Bolsa Client \$

Register

Name:

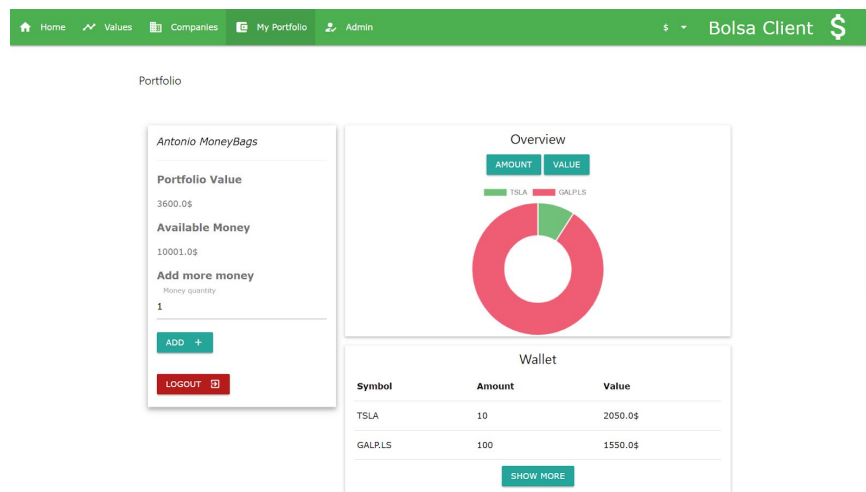
Username:

Password:

Password confirmation:

SUBMIT

Register Page



User Page.

In case the user is logged in, the user page will be shown in the portfolio tab, this page shows user information, the money it has and the information relative to the wallet, number of share, historic of purchases and sells, value of shares, from this page we are able to go to the shell page, that will allow us to sell shares.

Sell Shares

Wallet

Symbol	Amount
TSLA	10
GALPLS	100

Sell

Select Company to sell shares

Companies

How many shares would you like to sell?

1

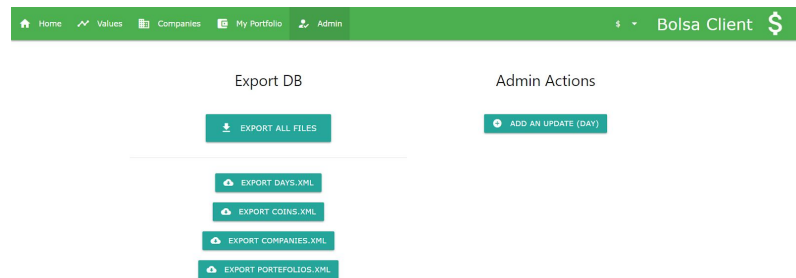
SUBMIT

Current Balance

Money:10001.0\$

Sell page

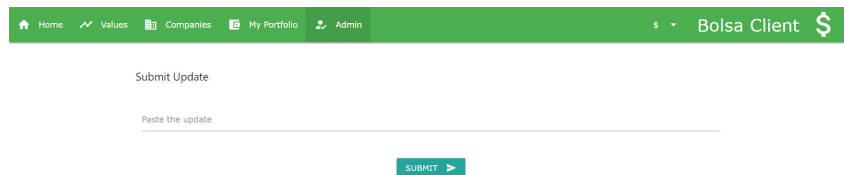
This page shows all the companies and the number of shares the user has, and allow to sell shares.



Admin page.

If the user is super-user, an admin tab will appear, this tab allows to administrate the page:

Download all database files, download just one of the xml stored on the xml and it also allows to add new updates, in order to refresh the values of each company.



Submit Update.

This page allows the administrator to modify the days xml, in order to add new values for the company each day. It implements error controls in order that only a correct update could be introduced inside the XML.



Coin selector.



In the upper right part of the page we find a coin selector, that will allow us to choose the coin we would like to see in the values, adjusted always to US Dollar.

Symbol	Price	Symbol	Price	Symbol	Price	Symbol	Price
TSLA	205.0\$	TSLA	168.1€	TSLA	170.15£	TSLA	23575.0¥
GALP.LS	15.5\$	GALP.LS	12.71€	GALP.LS	12.87£	GALP.LS	1782.5¥
AAPL	222.8\$	AAPL	182.7€	AAPL	184.92£	AAPL	25622.0¥
NFLX	304.7\$	NFLX	249.85€	NFLX	252.9£	NFLX	35040.5¥
AMD	20.0\$	AMD	16.4€	AMD	16.6£	AMD	2300.0¥
JMT.LS	11.0\$	JMT.LS	9.02€	JMT.LS	9.13£	JMT.LS	1265.0¥
WMT	100.0\$	WMT	82.0€	WMT	83.0£	WMT	11500.0¥
AMZN	1640.5\$	AMZN	1345.21€	AMZN	1361.62£	AMZN	188657.5¥

Prices in \$

Prices in €

Prices in £

Prices in ¥

Additional Considerations

Compromises

Even though our app simulates a full stock market with buy/sell operation we had to make some compromises:

- Values adjusted for each day
 - Stock markets are usually updated as fast as possible so that the users have the advantage compared to competitors. Our solution, given that we had no real access to data feeds, was to save values by day. Basically our values resemble a closing/typical price.
- Updates inside days.xml are sorted from old to latest
 - Our queries work with the *last()* operation in Xquery to get the most recent stock value. It is unfortunately hard to validate this at this schema level. Our system is designed so that the admin introduces updates on each new day, and they are inserted last to the list.

Layout System

When we started our work, frontend was our main focus. Upon counting the number of pages that were going to build we immediately saw a need to use Django template inheritance. This makes creating structured pages much easier.



We first built components, such as the one used to load our frameworks in *JS*. Then we included (by using the `include` function) them in a base page layout. That layout is our main structure that all the pages share. It includes the nav-bar, etc. By using blocks inside we can extend it to modify and/or add content.

For example: in our navbar we need to assign an active class to an item so that the frontend framework can style the button to make it appear selected (this makes the user know where he is inside the platform). By using a block that has default blank on each item, we can modify, on the pages that extend the layout, the respective block to active.

This system makes changes after the initial design easy, making the whole frontend flexible. If we want to add a item in the navbar, or simply load another script/CSS style we simply need to add it to the base layout and all the pages are updated/changed.

Custom template tags³

Even though it was our first time using Django we wanted to explore as much functionalities as we could doing this project. We found custom template tags very useful and decided to use them.

Template tags are now something we can customize to our needs effectively running Python code in the template system.

We use them to get all the coins to fill the navbar coin selector, to round values directly on the template, to dynamically calculate the converted values for the selected coin and to calculate the value of the user wallet or a specific stock in the portfolio page.

Django Authentication System⁴

While approaching this project we saw the need for a authentication system. Given that Django already has one implemented and secure proven, we didn't saw the need to reinvent the wheel. The only drawback was the fact that Django uses its configured database (default is SQLite).

The solution was to use the user id from Django, that is unique, inside the xml and effectively associate the two database that way.

With this we have the best of both worlds: our information in xml as requested and the security already implemented in Django.

Division of views and urls

³ <https://docs.djangoproject.com/en/2.1/howto/custom-template-tags/>

⁴ <https://docs.djangoproject.com/en/2.1/topics/auth/>



Near the end of project, our views file was getting big and hard to search so we decided to separate the views in two files upon starting the admin dashboard. One views file for the Admin pages and another for the rest of views. The same was applied to urls except no additional file was created but separated between to variables with the use of *include*.

Coin Selector

This functionality was added when the app was functional. Not wanting to completely re-engineer the work completely, we made use of template tags to get the coins and current coin value.

This made the coins completely dynamic. If we add another coin to the db it will appear in the dropdown on the navbar.

To save the coin selected across pages we made use of session variables⁵. This way the selected coin is saved by session and the transitions between pages do not break the selected coin.

RSS Feed

The RSS Feed implemented is dynamic. If the company symbol exists it will show news related. The news are sorted by date and each has a link to the source. The feeds are courtesy of [Yahoo Finance](#).

Frameworks used

We used a couple of frameworks that we need to reference/thank:

- [Materialize](#) - front-end styling and components
- [W3.CSS](#) - front-end styling
- [Jquery](#) - JS scripting
- [Chart.js](#) - included graphs
- [Material Icons](#) - icons on the app
- [Sortable.js](#) - sortable tables

Conclusions

Thanks to this project, we have acquired a better knowledge of the theoretical area of the course and applied in a deeper way, what we had learned in the practical one. It has also allowed us to form a multicultural group in which we have seen the strengths and weaknesses of each component of the group, in order to take advantage of those strengths and try to improve each

⁵ <https://docs.djangoproject.com/en/2.1/topics/http/sessions/>



other weaknesses, in order to get to the final result which is a functional page where you can do everything that was planned at the starting point.

Configuration to execute the application

We tried to make the app run almost automatically without much user configuration. BaseX database is created dynamically given that the BaseX server is running, and its default data imported. There is also a file which lists all the requirements used in the project so that the user can easily install python dependencies by calling *pip requirements.txt*.

Base requirements:

- Python 3.6/7
- BaseX
- Pycharm (optional)

Python packages requirements:

- BaseXClient==8.4.4
- Django==2.1
- lxml==4.2.5
- xmltodict==0.11.0

To run the project:

1. Check if you have all the requirements:
 - a. Run "pip install requirements.txt"

OR

 - b. Install manually each package
2. Start BaseX Server
3. Run Django Project
 - a. Import to a project in Pycharm proj1 folder which contains all django project

OR

 - b. Run "python manage.py runserver"

Considerations:

BaseX database is created dynamically on each Django start.

bolsa/baseX_db.py contains all this code create the db and validates the xml documents against their schema before importing them into BaseX.



User Accounts (admin):

There is one user account already create which is the admin. This account was created to test and see the functionalities of the system fully without the need to tamper around.

References

<https://developer.yahoo.com/finance/>

<https://www.w3schools.com/>

<https://docs.djangoproject.com/en/2.1/>

<https://tutorial.djangogirls.org/en/>

<https://simpleisbetterthancomplex.com/>

http://docs.baseex.org/wiki/Main_Page