PROGRAMMING STUDIO 2 PROJECT DOCUMENT

Crypto Dashboard

Elmo Ahorinta
elmo.ahorinta@aalto.fi
1009628
Computer Science
1st year
25.4.2024

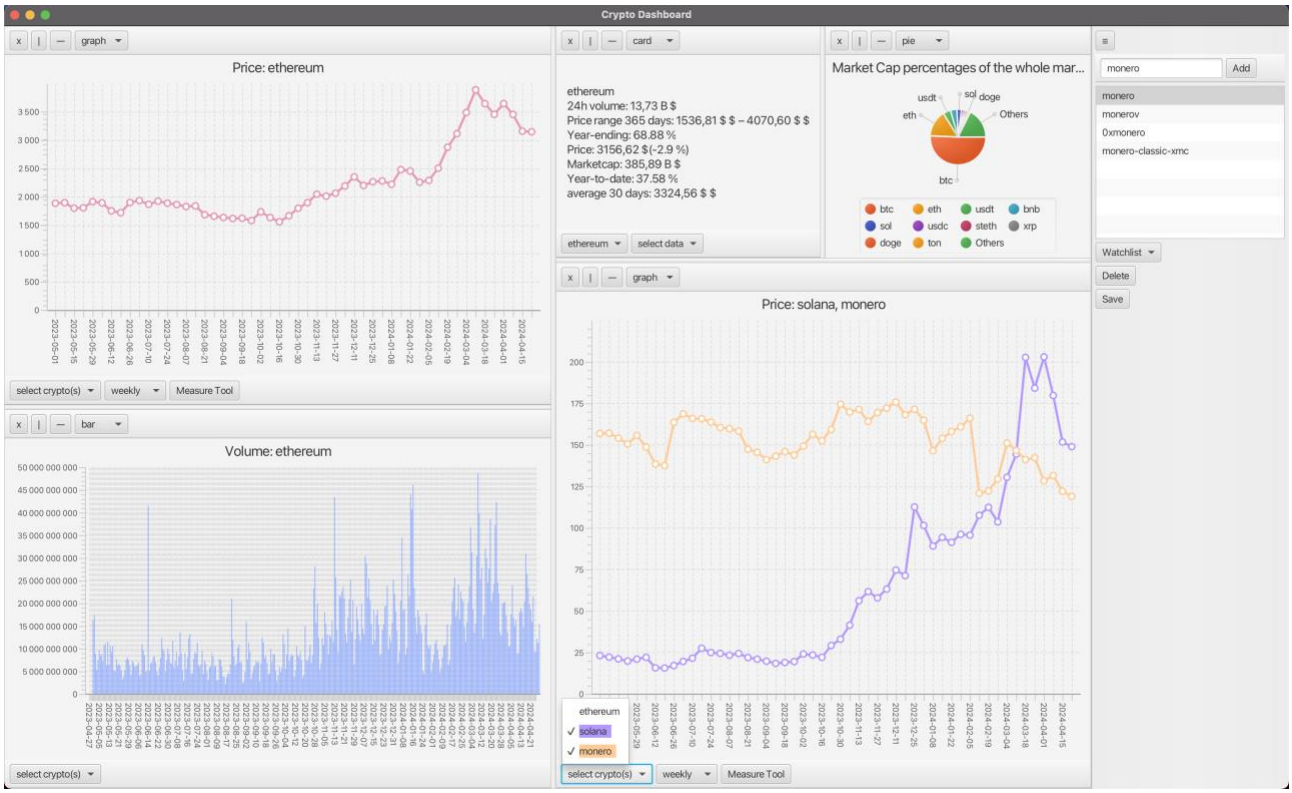
General description

Crypto Dashboard is an interactive dashboard, which consists of different types of charts and other panes that display key metrics. The user can dynamically compose the dashboard and add new panes. The content of the panes can be changed.

The user can search for new cryptos to be added to the dashboard.

The project was implemented at a demanding level.

User Interface



Launching the program

The API key should be added to the API object's api_key variable to be able to use the dashboard. The API can be found by signing up for a free plan on Coingeckos API: https://www.coingecko.com/en/developers/dashboard

The program can be launched from the LaunchDashboard object. The program will ask the user to provide a new file for the dashboard or continue with the last saved file. After that, the dashboard is created.

User interface functionality

The dashboard consists of panes and a sidebar that can be hidden or unhidden by pressing the hamburger menu in the top right corner. The sidebar has a search bar and a list of names of the cryptos that the API provides. The user can select a name and press the Add button to add crypto to the watchlist. The program will inform the user if adding the specific crypto is not possible.

The sidebar has a watchlist where the user can check which cryptos are added to the dashboard. To remove certain crypto, the user selects the crypto from the watchlist and presses the Delete button.

The sidebar has a save button for saving the whole dashboard. The dashboard is saved to the program structure and the user can download the file that creates the dashboard. When launching the program, the dashboard that was last saved or the dashboard file that the user provides will be used.

The panes of the dashboard are adjustable by dragging the edges of the panes. Each pane consists of the top toolbar, the content of the pane, and a possible bottom toolbar for controlling the content.

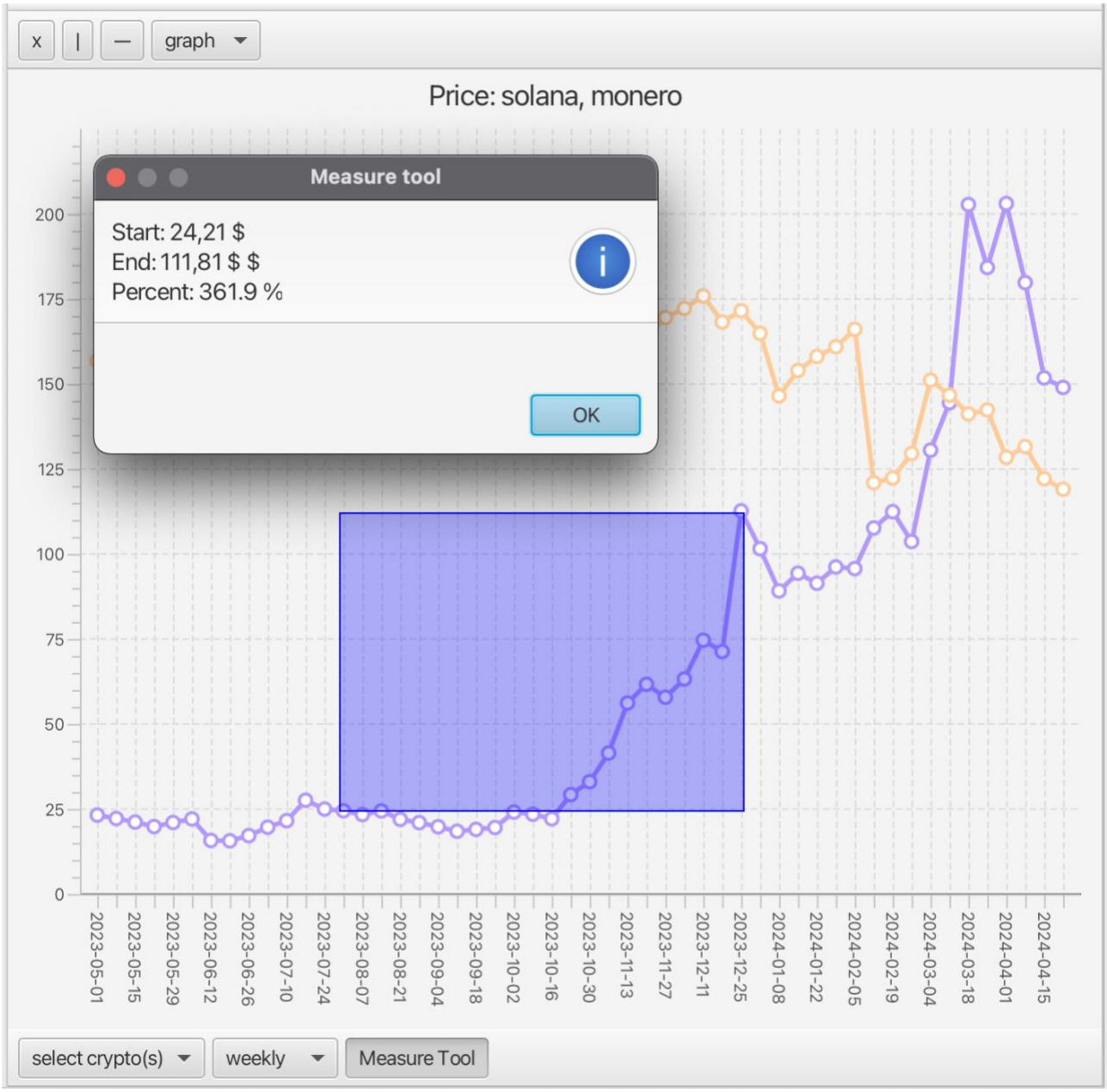
The top toolbar controls the adding, removing, and changing the content of the pane. X-button to remove pane, buttons that symbols are either vertical or horizontal line are for adding/splitting a pane. The orientation of the line defines the direction where the new pane is created. Choicebox decides what type of content the pane displays.

Possible types of content are price graphs, pie charts, volume bar charts, and cards that display key metrics.

Cards have a bottom toolbar which can be used to select crypto to be displayed. Key metrics of the crypto can be also selected and unselected from the toolbar.

The user can select cryptos to be displayed in the price graph and bar chart by selecting and unselecting them from the bottom toolbar. Right-clicking the graph's and chart's content the colors can be changed. Left-clicking provides the user with more information about the content. In the price graph, the interval can also changed from the bottom toolbar.

There is also a Measure tool in the graph view. This works as a rectangle tool that can be used to measure percent changes in the graph. By dragging and dropping anywhere on the graph, the program will inform of the percent change.


x
|
—
graph
Price: solana, monero
Measure tool
Start: 24,21 $
End: 111,81 $ $
Percent: 361.9 %
OK
200
175
150
125
100
75
50
25
0
2023-05-01
2023-05-15
2023-05-29
2023-06-12
2023-06-26
2023-07-10
2023-07-24
2023-08-07
2023-08-21
2023-09-04
2023-09-18
2023-10-02
2023-10-16
2023-10-30
2023-11-13
2023-11-27
2023-12-11
2023-12-25
2024-01-08
2024-01-22
2024-02-05
2024-02-19
2024-03-04
2024-03-18
2024-04-01
2024-04-15
select crypto(s)
weekly
Measure Tool

Program structure

```
├── build.sbt
└── src/
    ├── main/
    │   └── scala/
    │       └── cryptoDashboard/
    │           ├── app          <launching and controlling the dashbboard>
    │           ├── crypto       <crypto class>
    │           ├── services     <API, parser, file saving and building, utils>
    │           └── UI           <panes and other elements>
    ├── resources/
    │   └── data/
    └── tests/
```

The project structure above illustrates how the program is divided into packages and folders.

Since the program consists almost solely of different types of UI elements, certain design choices were made while implementing the program. The UI package includes all the content that the panes have and also Component class that controls how the panes behave. The elements such as graphs access the Dashboard object for getting the data for certain crypto. Dashboard object has also a few UI components that are used for controlling the dashboard as a whole.

UI package:

Component class
handles the deleting, splitting, and changing of the content of the pane.

Graph class
UI elements for the linecharts and logic for controlling the linechart.

Card class
UI elements for the cards and logic for controlling the cards.

Bar class
UI elements for the barchart and logic for controlling the barchart.

Pie class
UI elements for the piechat.

app package:

Dashboard object
Consists of the basic functionality of the dashboard such as adding and removing from the watchlist. The sidebar of the program is displayed by the Dashboard object.

Dashboard object has relations for other parts of the program. To add new cryptos to the dashboard, the Dashboard object asks the API object to fetch the data. API object with the help from Parser object returns this data for the Dashboard.

LaunchDashboard object
Used to launch the program. It creates the root pane, initializes the Dashboard object, and calls the BuildFromFile object to create the layout.

crypto package:

Crypto class
Needed for storing the information of the fetched data. Dashboard object keeps track of the cryptos that are added to the watchlist. The classes that display content use the Dashboard object's watchlist to get the data needed for the UI elements.

services package:

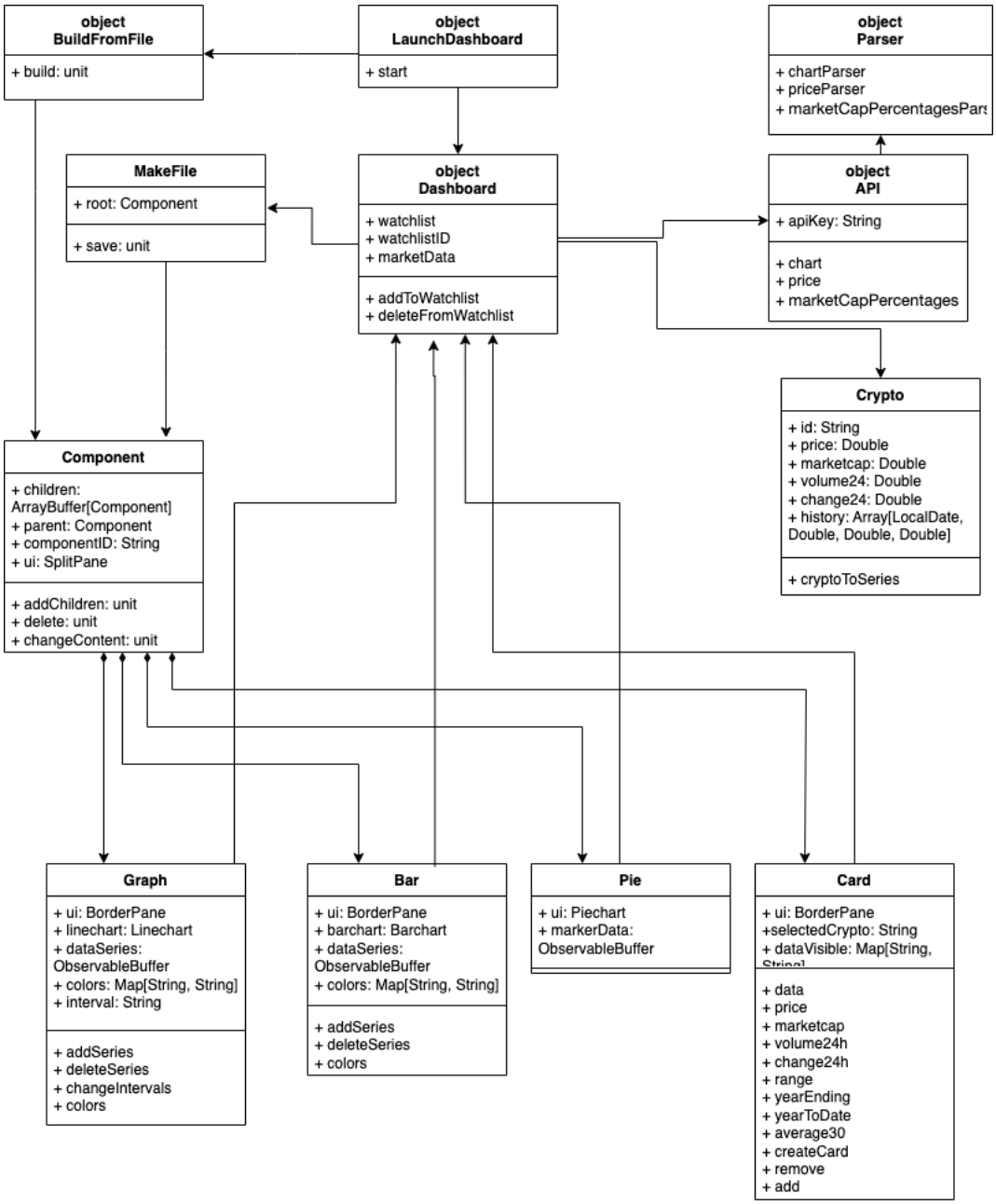
BuildFromFile object
Creates the dashboard during launch.

MakeFile object
Creates the file that is created when saving the dashboard.

Under services, there is also a package Utils that has a formatter that is used to format numbers for a better format to be displayed.

UML chart

UML chart illustrates the key classes, object, variables and methods of the program

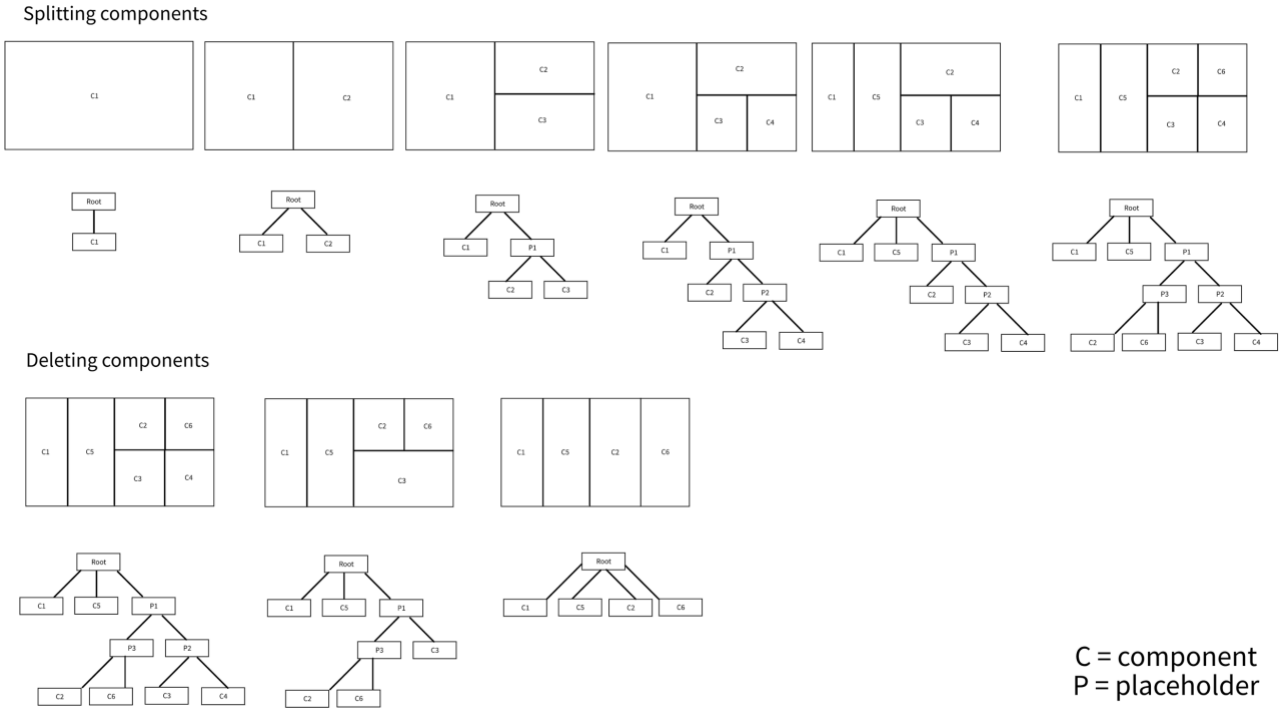


Algorithms

The main requirements of the program didn't require any complex algorithms. Calculating key metrics in Cards class utilizes quite simple algorithms, such as averages and percent changes from certain data points. However, Component class has an algorithm that handles the adding and removing of panes.

Dynamic layout splitting algorithm

Components class uses splitpanes to achieve adjustable panes. Splitpane is a ScalaFx container that can be split by adding new children to the container. But splitting splitpane can be done only to the orientation that the splitpane already has. To dynamically decide the orientation of the new split, an algorithm for the system was developed. Making the file and building the layout from the file also utilize the structure that is created by this splitting. The graph structure, the algorithm creates is illustrated below.



Algorithm for splitting components as pseudocode:

If the parent of the component is root and splitting is vertical ->
add a new component to the root to the correct position.

else if the component's parent's orientation is the same as the splitting orientation ->
add a new component to the parent to the correct position.

else if the component's parent's orientation is different from the splitting orientation ->
add a placeholder to the parent. This placeholder has the orientation of the splitting orientation. Remove the component that the splitting was initialized and add it to the new placeholder. Add a new component to the new placeholder.

Algorithm for deleting components as pseudocode:

if the component is only children of the parent -> delete the component and its parent placeholder.

if the component has one sibling -> delete the component and replace the parent with the sibling.

if the component has multiple children -> delete the component

Data structures

The program uses different data structures to achieve different functionality. When planning which data structures to use, the performance was not a factor, because the project doesn't utilize large data sets. To add functionality for large data collections, refactoring the program's data structures might be necessary. The most important data structures in the program are introduced below.

Dashboard's watchlist that stores the instances of the Crypto class is a **Mutable Map**. This map's keys are the IDs of the cryptos. The IDs are used widely in the program and are used as synonyms for the name of the crypto. This choice was made because it was the most consistent way of recognizing the cryptos. The names and tickers that the API provides were not a suitable way to differentiate the cryptos. By using the IDs all across the program made it easier to do API calls and for the contents of the components to display menus, etc. The contents could display the IDs as options in menus and get the data straight from the Dashboard's watchlist map by the ID.

ObservableBuffers are also utilized heavily in the program because this allows an easy way of updating UI elements of the program automatically. For example, the watchlist Map has a companion ObervableBuffer watchlistID that is updated when deleting or adding occurs to the watchlist. Once watchlistID is updated, all the menus, that listen to its updates, are also updated.

Components utilize **ArrayBuffers** to keep track of the children components each has. Arraybuffers enable an easy way of deleting and adding components. This ArrayBuffer and the variable that stores each component's parent are used in the dynamic layout algorithm explained in the previous section. This algorithm creates a tree graph that can be traversed up and down by accessing either the parent or the children. MakeFile creates the JSON from this tree and stores each component by creating an ID for the component and storing its content and parent's and children's IDs.

Files and Internet access

The program utilizes two files. The first one is for search functionality and the other one is meant to save the dashboard. There are also sample dashboards in the project structure that can downloaded and used to load a dashboard during launch.

cryptoList.JSON

The program contains a file of cryptos that the API provides. This JSON file is stored under resources/data and is used to provide the user with the search functionality of cryptos. The file contains also information on the ticker and name of cryptos but only the IDs are utilized.

output.JSON

The file is stored in resources/data/dashboardData. During launch, the program either uses this file or the file provided by the user to build the dashboard layout. MakeFile object overwrites this file whenever a user saves the dashboard. The user is also provided with the option to download this file during saving. Section of the JSON as an example:


```
"HjU" : {
  "parent" : "!a$",
  "children" : [ ],
  "cType" : "graph",
  "orientation" : "VERTICAL",
  "contentData" : [ "ethereum" ],
  "contentInterval" : "weekly",
  "dataVisible" : {
    "volume24h" : "",
    "range365" : "",
    "yearEnding" : "",
    "price" : "",
    "marketCap" : "",
    "range30" : "",
    "yearToDate" : "",
    "average30" : ""
  },
  "dataColor" : {
    "ethereum" : "e699b3"
  }
}
```


The file contains a map of the components. The content etc. other metrics are stored as a map. The watchlist or the IDs that need to be fetched from the API are also stored in this JSON.

There are sample dashboard files in resources/data/dashboardData/sampleDashboards. These can be manually downloaded and loaded to the program during launch.

Internet access is needed for the program. The program fetches the information from Coingecko's API and therefore will not fetch any information if there is no internet connection and therefore will not launch if there is no internet connection. If the connection is lost during runtime, the user can use the dashboard normally except there will be no possibility to add new cryptos.

Testing

The program was mainly tested using the actual graphical userface because many of the main features of the program concerned the UI elements. Especially testing and implementing the dynamic layout was done by trial and error. This method of development helped to find the proper algorithm for the layout system.

Other parts of the program were also tested using a graphical userface. For example, cases of the axis not scaling properly were found using the graphical userface, and the cause was fixed.

Exceptions were used to ensure that the data fetched from the API was not faulty. Many new exception cases were added to the program when new issues arose. For example, the data that the API provides unexpectedly changed during production; when requesting data for 365 days, the API sometimes returns an array of length 366 with an extra duplicate item at the end. The data the API provides is frequently faulty with lesser-known cryptos, and the faulty data caused many issues in the graphs. To prevent these issues, many conditions for the integrity of the data were made to ensure that these issues would not arise. This meant that many cryptos are not possible to add many to the dashboard from the search list.

Known bugs and missing features

Bugs

Complex dynamic layout issue:

When splitting components in the dynamic layout multiple times, the new splits cause components to go over the program window. This causes some components to be inaccessible. Also, some components, depending on the layout, will stretch and therefore be quite unusable.

Many fixes for this bug were considered. However, a dashboard build with a bigger screen would not work with smaller screens at all if the splitting is restricted. With bigger screens, more complex layouts could be built and the choice not to restrict this was made.

API issues:

Due to the limitations and slowness of the API, adding multiple Cryptos to the dashboard may be slow, or adding them could not work. The program handles these situations well, but if this issue arises the wanted cryptos are not added to the dashboard. A more user-friendly way of doing this would be adding them in the background if it was not successful. Also, the API calls during launch could be reduced due to the API changing during production, but this is not yet implemented.

Missing features

Though the requirements of the projects were met, the way some features are implemented does not at least satisfy me.

For example, legends were not implemented with the built-in methods in ScalaFx, because with multiple-series scatters changing a specific series's color was not an easy task. The colors of the series could be changed but the legends were not changing properly. Due to this built-in legends were removed and the menu, where the user can select cryptos to be displayed, is used as legends. Better implementation using the actual legends is missing, but though not even necessarily possible in ScalaFx.

More options for analyzing the graph's data are much needed but missing. Different types of indicators, such as moving averages, are not yet implemented. These were not a requirement in the project, but they were part of my initial plan. Also, more control of changing time series would be useful.

3 best sides and 3 weaknesses

3 best sides:

Dynamic layout using the Component class

The finished functionality to split components to either of the directions is novel and works well. Coming up with a solution for the working implementation was hard and time-consuming, but it paid off in the end. Though the code probably can be improved.

Ability to save and load complex layouts
The user can arrange the dashboard and specify colors, intervals, etc. and the layout will look the same after saving and loading the dashboard.

Clear and intuitive user interface
The program is quite user-friendly.

3 weaknesses:

Redundancy with implementing classes that have the same type of sections.

The project as a whole needs refactoring to accommodate future development. Classes that extend other classes could be used to remove unnecessary redundancy.

Unnecessarily complex methods were used a few times in the project.

Breaking down the more complex method could help with the readability of the code.

Test classes are missing
The project doesn't currently have a written test. One reason for this is a lack of suitable sections to test. Error handling deals already with faulty data. For future development, test classes would help implement for example more complex calculations. Currently, calculations are quite simple and it would be unnecessary to add tests to ensure the results.

Deviations from the plan, realized process and schedule

Order, in which the project was implemented only roughly followed the initial plan. The following sprints don't necessarily describe the exact two-week periods, but give a timeline of how the program was implemented:

Sprint 1:

Initial API features were developed. This data was also used to test how to implement the graphs. Time was also spent getting familiar with the Scalafx library. The initial plan didn't specify how the dynamically adjustable components would be implemented, and for this reason, many viable solutions were considered. SceneBuilder, a program used for designing JavaFX applications, was used for testing what kind of containers and controls could be used. SceneBuilder was used multiple times during production to decide what kind of elements would be suitable.

Sprint 2:

There was not much actual progress made during the second sprint mainly because of the problems with implementing the dynamic layout. The plan on how the layout should look in the end was decided in sprint 1, but implementing it ended up way harder than I thought. The first draft of the layout functionality was added, but it was full of shortcomings.

Sprint 3:

Implementing the dynamic layout was continued. Other UI elements that are used inside the components were developed. These classes were mostly developed to be used in the implementation of the layout, and they were not finished.

Sprint 4:

The dynamic layout was finished. A save file, that contains the layout information, was developed. Also, a builder that builds the layout from that save file JSON was implemented. During implementation, the dynamic layout was also rewritten and improved. The different types of content were developed further.

Sprint 5:

The main goals of the project were mostly done at this point, but many of the features had many bugs. Fixing these bugs was not always trivial. For example, changing the colors of the legends in Scalafx was not easy or completely impossible. Time was wasted on trying to do something and. eventually just implementing it differently, because either it was not possible in ScalaFx or I couldn't do it. The rest of the production was focused on fixing the known bugs and testing possible corner cases. Refactoring of classes was also done to simplify and improve the project as a whole.

Main takeaways realized during implementation

The plan should have been more detailed and designed. This would have helped with tracking the progress of the whole project. The actual implementation of features was not focused enough; time was not used to develop one feature at a time. Instead features that were being developed changed too frequently, and that is why many features were left halfway implemented during production. In addition, the time used for the project was not evenly distributed. Usually, the progress of the two-week sprints was made only in a day or two. More evenly distributing the workload would have made the production smoother.

Final evaluation

As a whole the project was succesfull and meets the requirements and in other fields exceeds the goals by a lot. Functionality for these level of project is great, but some parts of implementations need a bit of refactoring.

The program has a lot of room for improvement and many features could be added, but refactoring is much needed. The projects main structure though, at least in my opinion, supports already future features.

If I started the project again from the beginning, I would have made a better plan. This way many of the struggles regarding how to implement the project as a whole would have been easier. Also divinding the workload more evenly would have ensured better pace for the progress. Also I should have prepared for the interium meetings. That way I could have asked for advice for some key strugles with my design choices.

References

- CoinGecko API (2024). [online] Available at: https://docs.coingecko.com/reference/introduction

- Gluon. JavaFx Scene Builder. [online] Available at: https://gluonhq.com/products/scene-builder/

- Haoyi, Li (2024) uPickle 3.1.2. [online] Available at: https://com-lihaoyi.github.io/upickle/

- SoftwareMill (2024). sttp: the Scala HTTP client. [online] Available at: https://sttp.softwaremill.com/en/stable/

- Playframework (2024). The Play JSON library. [online] Available at: https://www.playframework.com/documentation/3.0.x/ScalaJson

- ScalaDex (2024). toolkit. [online] Available at: https://index.scala-lang.org/scala/toolkit/artifacts/toolkit/0.1.7