# Graphelier

## Team members

| Name and Student id | GitHub id | Number of story points that member was an **author** on. |
|---|---|---|
| **Lucas Turpin 40029907** | Lercerss | 14 |
| Patrick Vacca 40028886 | pvacca97 | 10 |
| Duy-Khoi Le | alvyn279 | 11 |
| Panorea Kontis 40032422 | panoreak | 10 |
| Chirac Manoukian | chimano | 18 |
| Christoffer Baur | chris-baur | 10 |
| Riya Dutta 40028085 | riyaGit | 12 |

## Project summary

Graphelier is a web application that displays historical financial market data through an intuitive graphical representation. The application allows the user to explore the state of the market's order book across time, viewing events that modify its contained orders. On each price level, not only is the total volume displayed but also how it is divided into individual orders. Users can explore the order book at varying time granularities from hours or minutes to individual events separated by only a few nanoseconds. The best bid and ask, called the top of book, can also be viewed as a graph over time allowing the user to cross-reference it with complementary data sets such as time series or proprietary order flow.

## Risk

The project's biggest risks are related to the data required to achieve its features. The first one comes from the fact that data is not easily accessible as it is distributed on a subscription basis or sold from various vendors. The second is the size of the data set which can reach multiple gigabytes per day. To allow for an enjoyable user experience, the application must be able to efficiently handle this large data set.

The first risk was tackled by using a data sample from LOBSTER then using that sample as a basis to build a larger extended data set.
The second was tackled by building a large data set early. A key frame algorithm is used for building an orderbook efficiently for any given time. Due to the inherently

partitioned nature of the data, the application is scalable to achieve higher throughput if necessary.

## Legal and Ethical issues

The market data required to accomplish this project cannot be distributed openly and therefore poses a challenge in both acquiring it and displaying it publicly. As such, our project uses fake data based on a sample from June 2012 to circumvent those issues.

## Velocity

*Project Total*: 9 stories, 31 points over 8 weeks

Iteration 1 (2 stories, 6 points)
Containerized the project using docker and docker-compose. Wrote a parser that parses the market data and saves it in MongoDB. Backend successfully generated an order book for a fixed point in time and the react application displayed it to the user.
Iteration 2 (1 story, 3 points)
Set up the continuous integration environment for the project using Circle-CI. Set up the linters and tests for all the components in the project.

Iteration 3, (2 stories, 5 points)
Implemented the keyframes algorithm to build the order books for any given time. The react application now allows users to pick the time they would like to generate an order book for. Modified the parser so that it can extend our sample to be longer than an hour. Styled the frontend.

Iteration 4, (4 stories, 17 points)
The application now displays the list of messages. User can now see the order book as it is being modified one message at a time.

*Release 1 Total*: 9 stories, 31 points over 8 weeks
Release 1, Iteration 4, (4 stories, 17 points)
Application displays the order book at any given point in time at a nanosecond precision. Users can view messages relevant to the current state of the order book and apply them one by one and see how it affects it.

## Overall Arch and Class diagram

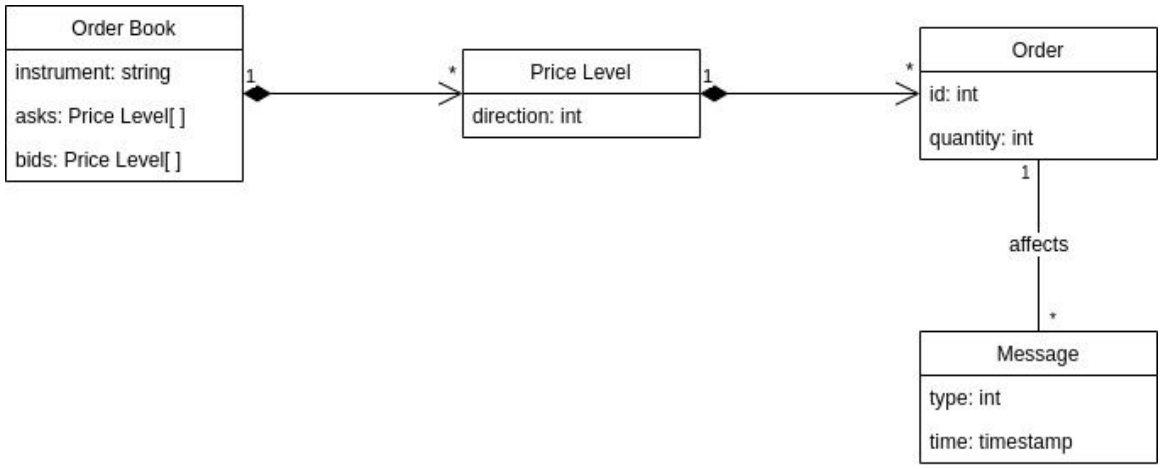


Figure 1: Domain Model

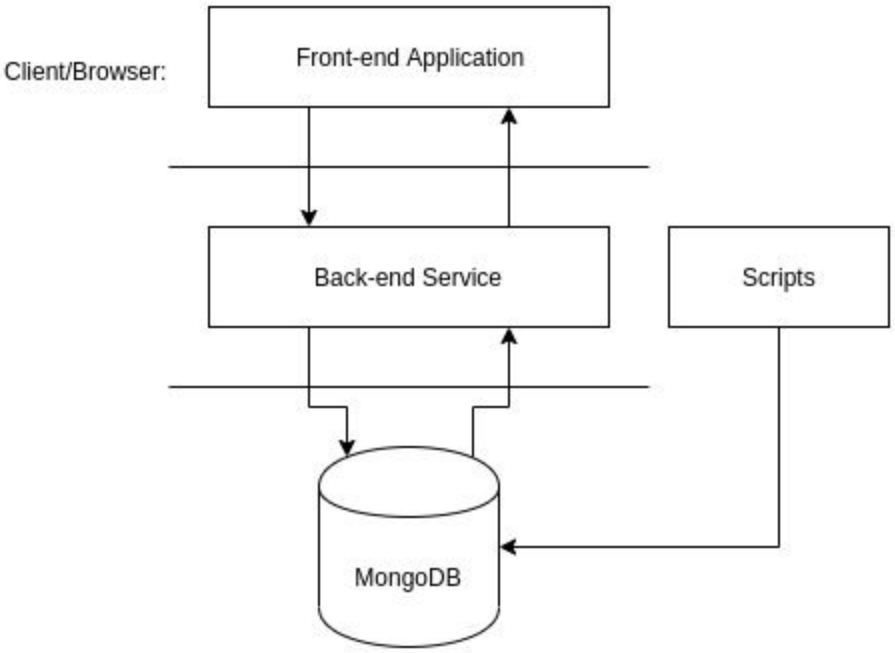


Figure 2: High-Level Architecture

From a high-level architecture perspective, the project is separated into three separate layers composed of the front-end application, the back-end service with its associated scripts, and the mongoDB database.

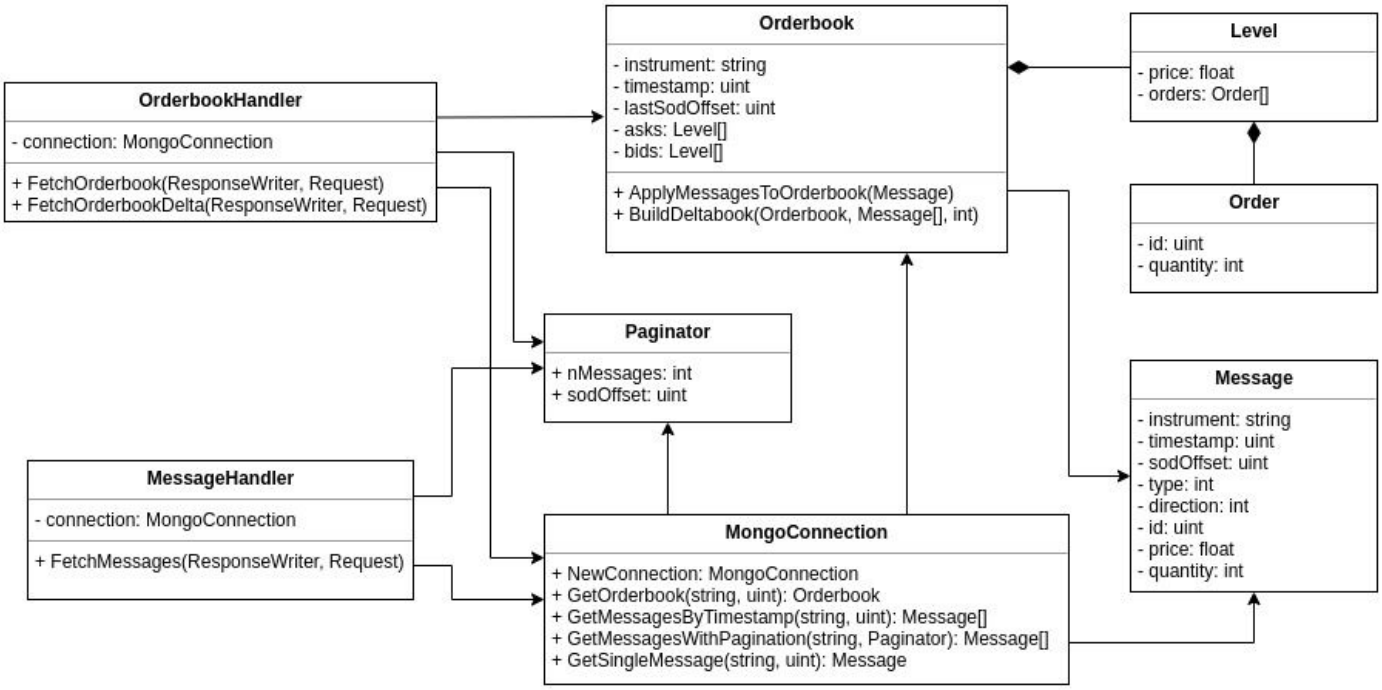


Figure 3: Class Diagram - Service Implementation

## Infrastructure

On the front-end side, React bootstrapped with Create React App is used to implement a dynamic application on the browser.
On the back-end side, the HTTP server is built using the golang built-in http library along with Gorilla Mux.
MongoDB is used as the database for the server.
Each component is run as its own Docker container to allow for environment encapsulation and orchestration of processes.

## Name Conventions

JavaScript code should be formatted according to a custom set of rules extended from the Airbnb JavaScript Style Guide and ES6 standards.
Go code should be formatted according to Effective Go.
Python code should be formatted according to the PEP 8 Style Guide.

## Code

Key files:

| File path with clickable GitHub link | Purpose (1 line description) |
|---|---|
| graphelier/core/scripts/importer.py | This file is responsible for populating the database with fake orderbook and message data based on a given sample set. |

| | |
|---|---|
| graphelier/core/graphelier-service/api/hndlrs/msghandler.go | This file is responsible for supporting pagination to the frontend of the application by returning messages based on parameters given. |
| graphelier/core/graphelier-service/models/orderbooks.go | This file is responsible for holding the orderbook structure as well as various manipulation functions on an orderbook itself. |
| graphelier/app/src/components/OrderBookSnapshot.js | This is the parent component that holds the orderbook, the messages list, and the controls to view different states of the orderbook. |
| graphelier/app/src/components/TimestampOrderBookScroller.js | This is the component that renders every row of the orderbook, and that provides controls to change the orderbook state at message-level granularity. |

### Testing and Continuous Integration

| Test File path with clickable GitHub link | What is it testing (1 line description) |
|---|---|
| graphelier/core/scripts/tests/test_extender.py | This test file is responsible for ensuring that the sample set given is properly extended over multiple days in order to have a larger set of data. |
| graphelier/core/graphelier-service/api/hndlrs/msghandler_test.go | This test file is responsible for ensuring the different pagination calls to the api return the correct information to the frontend application. |
| graphelier/core/graphelier-service/models/orderbooks_test.go | This test file is responsible for ensuring that the manipulation of the orderbook content is correct. |
| graphelier/app/src/tests/component-tests/OrderBookSnapshot.test.js | It tests the date and timestamp picking functionality and the handling of the initial orderbook raw data received from the back-end. |
| graphelier/app/src/tests/component-tests/TimestampOrderBookScroller.test.js | It tests the correctness in rendered orderbook rows and function calls related to message stepping and scrolling within the orderbook table. |

Continuous integration is configured on CircleCI with builds for every pull request or new commits to *master*. Builds execute tests, static code analysis and packaging

for each of the three components composing the application: the front-end application, the back-end service and the data-related scripts. An additional tool called GolangCI provides a static analysis report directly on GitHub pull requests specifically for the service code implemented in Go. These tools together enforce that commits are always functional and adhere to the code style rules for each programming language.