Graphelier

Team members

Name and Student id	GitHub id	Number of story points that member was an **author** on.
Lucas Turpin 40029907	Lercerss	14+19
Patrick Vacca 40028886	pvacca97	10+16
Duy-Khoi Le 40026393	alvyn279	11+18
Panorea Kontis 40032422	panoreak	10+14
Chirac Manoukian 40028500	chimano	18+16
Christoffer Baur 40028050	chris-baur	10+10
Riya Dutta 40028085	riyaGit	12+11

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: 19 stories, 68 points over 19 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. Users can now see the order book as it is being modified one message at a time.

Iteration 5, (2 stories, 4 points)
The instrument whose orderbook is analyzed can now be selected. Furthermore, the application is now adaptive to custom screen sizes (i.e.: the computer screens at Squarepoint).

Iteration 6, (1 story, 2 points)
Timezone-related bugs were fixed. The action of clicking on a row of the message table will lead to the update of the orderbook being shown.

Iteration 7, (2 stories, 7 points)
Migration of front-end source code to a typed version of JavaScript, namely TypeScript. The application can now be accessed through a public URL. The continuous integration and continuous delivery pipeline is now set up.

Iteration 8, (5 stories, 24 points)
Graph displaying the best bid and best ask over time for an entire business day. Added an icon while loading a new orderbook state. Prompted the most recent message in the message list. Convert native DatePicker to be compatible with material-ui DatePicker. Updated milestone document.

Release 1 Total: 9 stories, 31 points over 8 weeks
Release 2 Total: 10 stories, 37 points over 11 weeks

Overall Arch and Class diagram

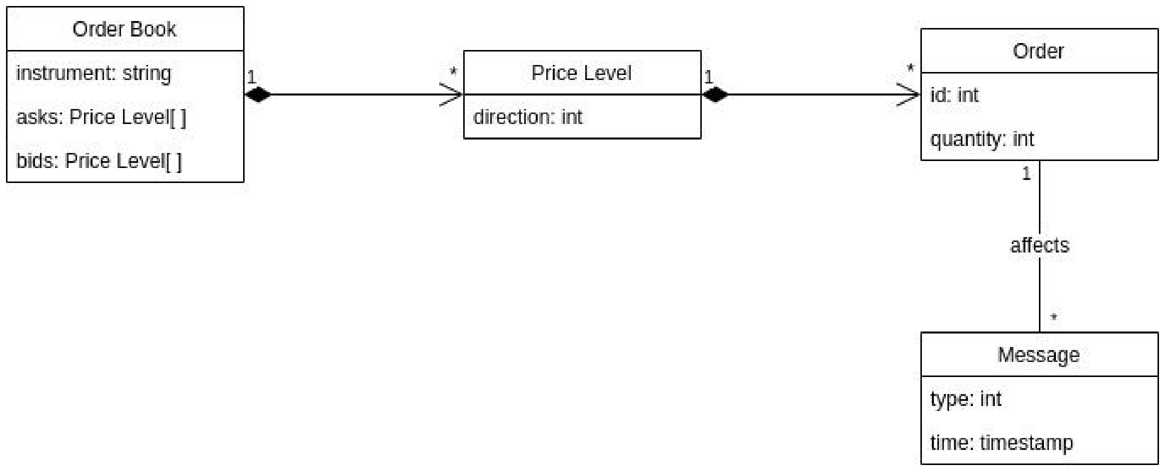

Figure 1: Domain Model

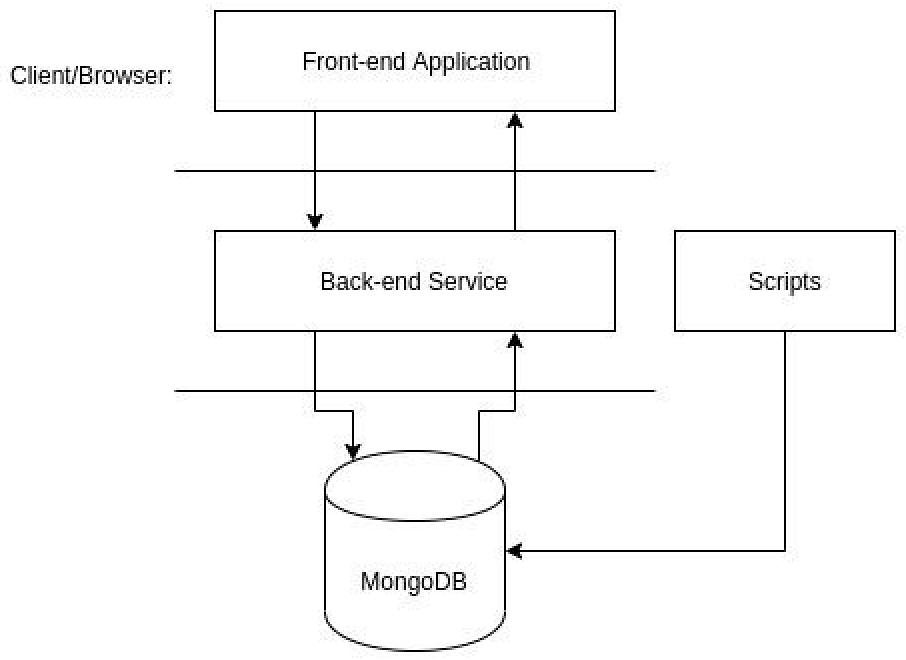

Figure 2: High-Level Architecture

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.

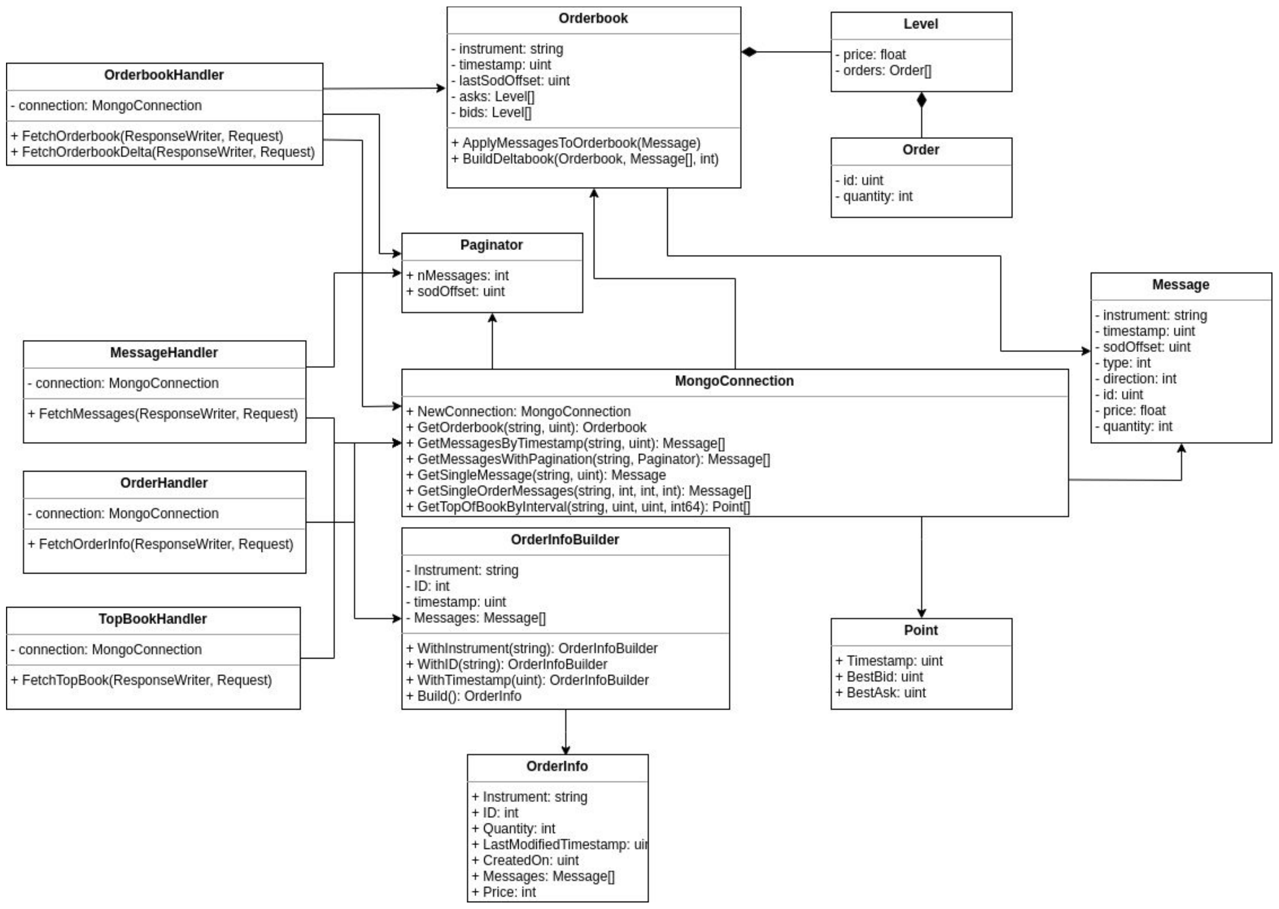
Orderbook
- instrument: string
- timestamp: uint
- lastSodOffset: uint
- asks: Level[]
- bids: Level[]
+ ApplyMessagesToOrderbook(Message)
+ BuildDeltabook(Orderbook, Message[], int)
Level
- price: float
- orders: Order[]
Order
- id: uint
- quantity: int
OrderbookHandler
- connection: MongoConnection
+ FetchOrderbook(ResponseWriter, Request)
+ FetchOrderbookDelta(ResponseWriter, Request)
Paginator
+ nMessages: int
+ sodOffset: uint
Message
- instrument: string
- timestamp: uint
- sodOffset: uint
- type: int
- direction: int
- id: uint
- price: float
- quantity: int
MessageHandler
- connection: MongoConnection
+ FetchMessages(ResponseWriter, Request)
MongoConnection
+ NewConnection: MongoConnection
+ GetOrderbook(string, uint): Orderbook
+ GetMessagesByTimestamp(string, uint): Message[]
+ GetMessagesWithPagination(string, Paginator): Message[]
+ GetSingleMessage(string, uint): Message
+ GetSingleOrderMessages(string, int, int, int): Message[]
+ GetTopOfBookByInterval(string, uint, uint, int64): Point[]
OrderHandler
- connection: MongoConnection
+ FetchOrderInfo(ResponseWriter, Request)
OrderInfoBuilder
- Instrument: string
- ID: int
- timestamp: uint
- Messages: Message[]
+ WithInstrument(string): OrderInfoBuilder
+ WithID(string): OrderInfoBuilder
+ WithTimestamp(uint): OrderInfoBuilder
+ Build(): OrderInfo
Point
+ Timestamp: uint
+ BestBid: uint
+ BestAsk: uint
TopBookHandler
- connection: MongoConnection
+ FetchTopBook(ResponseWriter, Request)
OrderInfo
+ Instrument: string
+ ID: int
+ Quantity: int
+ LastModifiedTimestamp: ui
+ CreatedOn: uint
+ Messages: Message[]
+ Price: int

Figure 3: Class Diagram - Service Implementation

The OrderBook contains an array of PriceLevels which contains an array of Orders. The OrderBook collection represents snapshots of the orderbook at every interval specified in the Meta collection. The Message collection is used to store messages broadcasted by the market. The data view can be seen in figure 4.

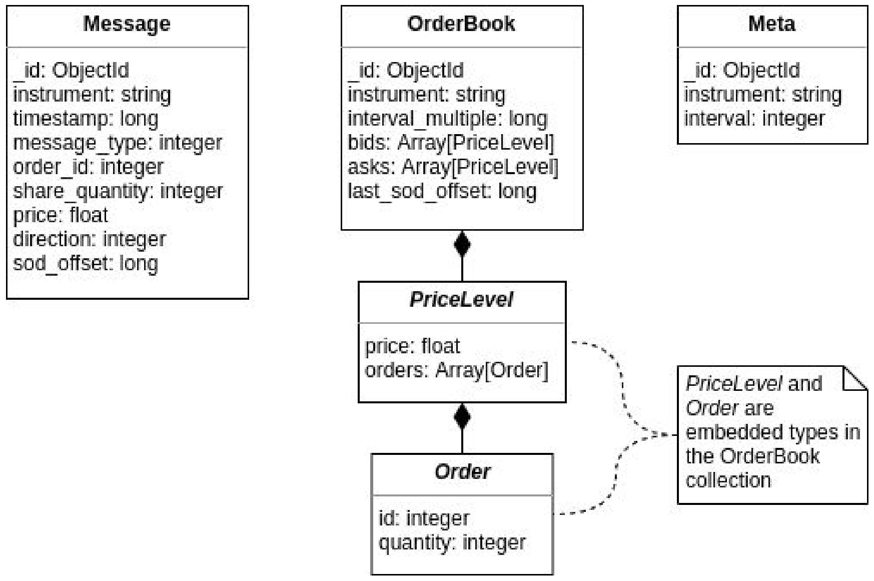

Figure 4: Data View

Infrastructure

On the front-end side, React bootstrapped with Create React App is used to implement a dynamic application on the browser. As per the language used, TypeScript is included on top of the React JSX code to enforce typing in the source code. This allows for design and implementation by contract. With models defined, the interaction between the front-end and back-end is made seamless through verifiable interfaces with respect to each other. Not only does this improve the development process for the developer, good code quality and standards are encouraged.

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.

The production environment is very similar to the development environment in terms of backend and mongo. However, the frontend application gets served using an Nginx container. The production environment is hosted on AWS, making use of their EC2, ECS and ECR services. The docker containers are run on EC2 using ECS for deployments, and the docker images are pushed to ECR. The setup can be viewed in figure 5.

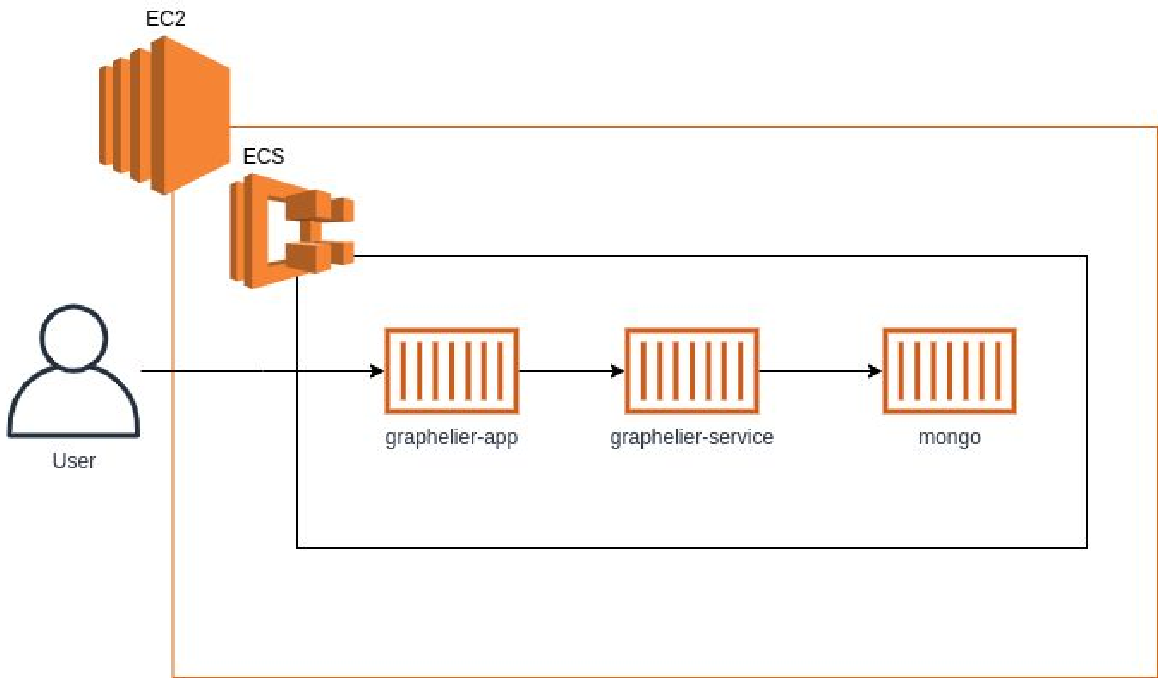

Figure 6: Production Infrastructure Diagram

Name Conventions

JavaScript code should be formatted according to a custom set of rules extended from the Airbnb JavaScript Style Guide and ES6 standards. TypeScript standards are also applied at compile time in order to detect coding style errors.
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.tsx	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/TopOfBookGraph.tsx	This file is responsible for displaying the best bid and best ask over the course of a full business day in the format of a graph in which the users can select various points.

Testing and Continuous Integration

Test File path with clickable GitHub link	What is it testing (1 line description)
graphelier/core/scripts/extender.spec.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/obhandler_test.go	This test file is responsible for ensuring the different orderbook and deltabook 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.tsx	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.tsx	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.

Continuous deployment is also configured on CircleCI. Upon merging a branch on master, four jobs start running. Three of these jobs are used to build the mongo, frontend and backend images and push them to AWS' Elastic Container Registry. The fourth job then updates the Elastic Container Service's task definition of our application to use these newly pushed docker images. The service that is running our application then tears down the old containers and redeploys containers built off the new images.