Service-Centric and Cloud Computing

Coursework Report

Daniel Holland

N0697611

Table of Contents

Evaluation of Application....3
- Functional Description of Application....3
- Major Design Decisions....4

Analysis of Quality of Service....5
Application of Semantic Web and Linked Data Technologies....8
References....11
Bibliography....11
Appendices....12
- Appendix A....12
- Appendix B....13
- Appendix C....13

Evaluation of Application

Functional Description of Application

The system is built around a web server which provides a REST API for consumption by the client. This web server itself makes use of two other REST services, one of which is external (WorldTradingData.com), the other of which has been developed alongside the main web service. This other web service is written in Go to cache the daily exchange rates from the European Central Bank (via exchangeratesapi.io), and to provide these to the main server via a REST API.

The REST API on the main web service provides functions to create a user account, to login and generate an API key, to get the shares and quantities for the user (and apply search and order-by criteria to this), to make share transactions, to list all shares (and to also apply search and order-by criteria to this), and to remove shares from a user's account.

The Currency Converter Service provides methods to convert currencies, to list all currency codes, and to get exchange rates.

On opening the web-based client, the user is required to log-in or register, then presented with a list of the shares that they are tracking, and in what quantities. They may search this list of shares, change the display currency, sort them by any of the columns on the table. The user may buy and sell shares which they are tracking. The User can open a list of all shares to view and search all shares. From this they may select additional shares to track. See Appendix A for UI Screenshots.

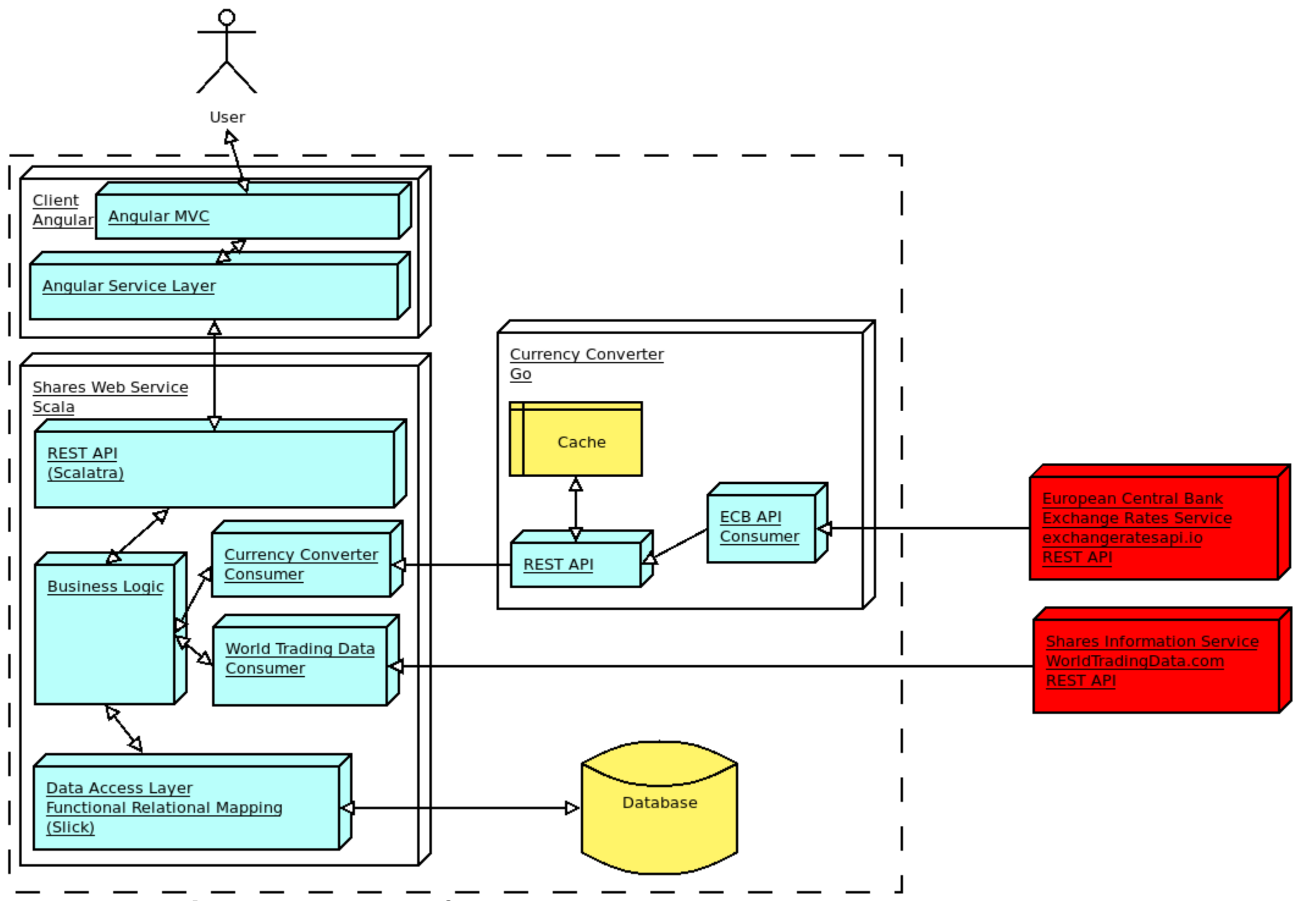


Figure 1: Deployment Diagram of System

Major Design Decisions

REST was chosen over SOAP for the main web service provided. An advantage of REST is that it is very easy to test using a variety of tools. Whilst SOAP can also be tested, no setup or orientation of any kind was necessary to use curl and Firefox for initial testing. Later in the project, more powerful tools such as Postman, Gatling, and JMeter were used to supplement these. Another strong advantage of REST is the ease with which it can be consumed by Angular applications and static websites.

The Shares Web Service REST API was written in Scala, using the Scalatra framework. The Scalatra Framework was chosen because of its simplicity and ease of use, particularly the concise way in which REST methods are declared using it. The Business Logic, Data Access Layer, REST API Servlets, and External API consumers are all kept carefully separate. The Data Access Layer uses the Slick framework used for Functional Relational Mapping. Slick is a powerful framework, but has a complex Domain Specific Language which gave it a very steep learning curve. Choosing an easier alternative would have likely resulted in the product being finished in a shorter time. However, the framework now having been learnt, if redoing the project, the same framework would be used. The Database used is SQLite, which is well suited to small deployments, but a weak choice for high volumes and data and large numbers of requests. As such, with growth in number of users and amount data stored, a more powerful database solution may become more appropriate. The cost of such future modification has been pre-emptively minimised by isolating the data access layer and using Slick, so that the database used can be changed without requiring any changes to elements of the program outside the data access layer.

The Client is a statically hosted site written in Angular, which allows it to be used from Desktop, Laptop, and Mobile devices alike. When the Angular App loads, the whole app loads before the first page is displayed. The advantage of this is that the user can click seamlessly between different pages in the client, a disadvantage is that the app takes a couple of seconds to load the first page. The client separates out the REST API calls into a service layer, and the frontend uses the Model View Controller design pattern to further separate the different elements. This separation allows for changes to be made to parts of the system without directly effecting others. Because the client is statically hosted, it can be deployed either to a single web server or to a Content Delivery Network (CDN). Being able to be deployed to a CDN provides a powerful ability to scale the deployment up and down easily. This will allow the capacity to adjust to the number of users making requests, both as the service grows over the long term, and also during peak and off-peak times of day. Outsourcing hosting to a CDN would also improve the reliability of the service, as well as providing a resilience against Distributed Denial of Service attacks.

Go was used to write the Currency Conversion Web Service, which retrieves the European Central Bank exchange rates on a daily basis, caches them, and provides them for usage by the main Shares Web Service. A major benefit of Go in this context is that it is that software written in it is very fast, as it is a compiled language. Go has been designed with cloud development in mind, and has efficient concurrency made easy to implement through Goroutines. This strong support for concurrency and cloud could be taken advantage of to increase capacity by parallelising processing of user requests across different threads and servers. As with the API Consumers in the Shares Web Service, the API Consumer in the Currency Conversion Web Service has been isolated from the rest of the application. This allows it to be modified if the external service it consumes changes or is replaced, without effecting the rest of the application.

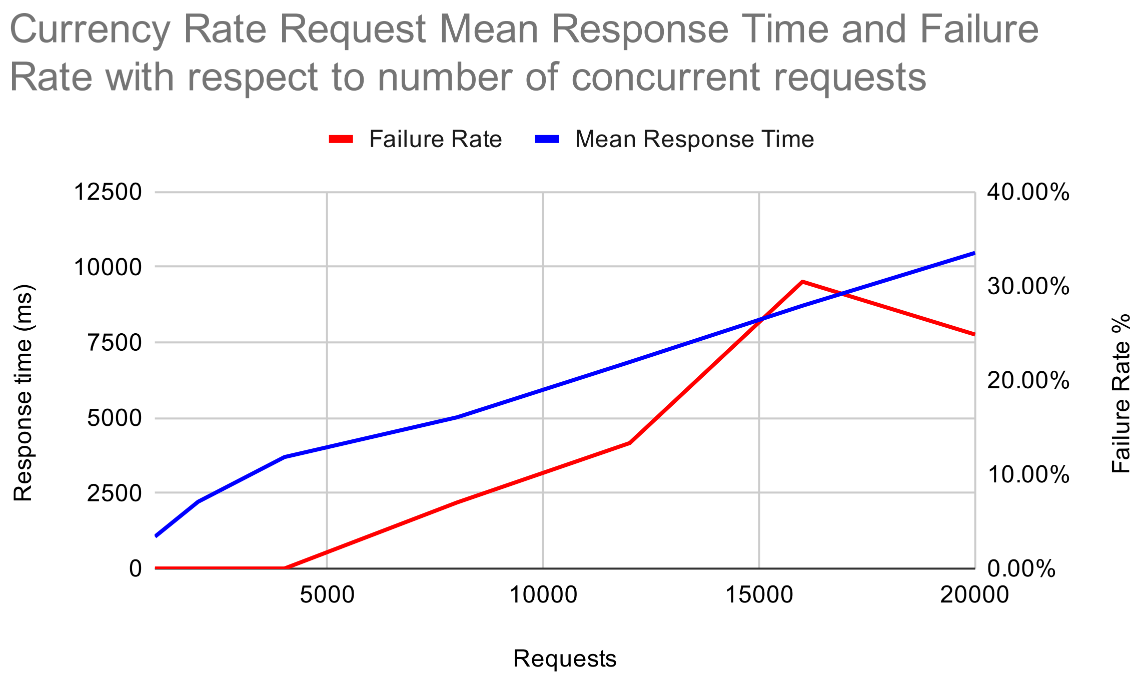
Analysis of Quality of Service

As service demand increases, a service's software and infrastructure must scale with it, or else risk a deterioration in Quality of Service.

All testing was carried out from the university network, with the two Web Services and the Angular app all hosted on the same DigitalOcean droplet.

Effect of Concurrent Requests

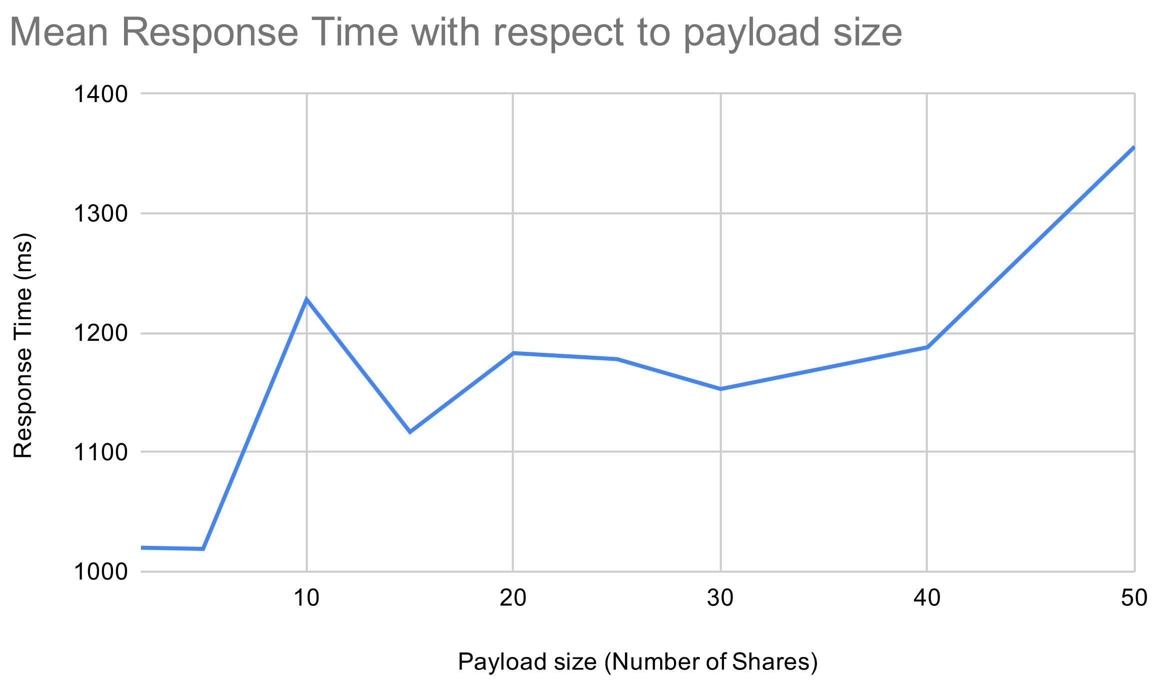
The graph below shows how the number of concurrent Client requests for Currency Rates from the Shares Web Service affect its response time and failure rate.



Note the general upwards trend for both failure rate and response time. The failure rate for 20000 Requests is higher in absolute terms than for 16000 requests, but not in proportion to the number of requests, hence the drop-off at the end. See Appendix B1 for raw data.

Effect of Payload Size

The graph below shows how the payload size effects the service response time.



Mean Response Time increases with payload size. This test made 1000 concurrent requests to the system to list x shares. Shares are approximately 160 bytes in size each. Due to the size of the dataset in use, the testing could not be extended to a larger number of shares. See Appendix B2 for raw data.

Keeping Users Updated

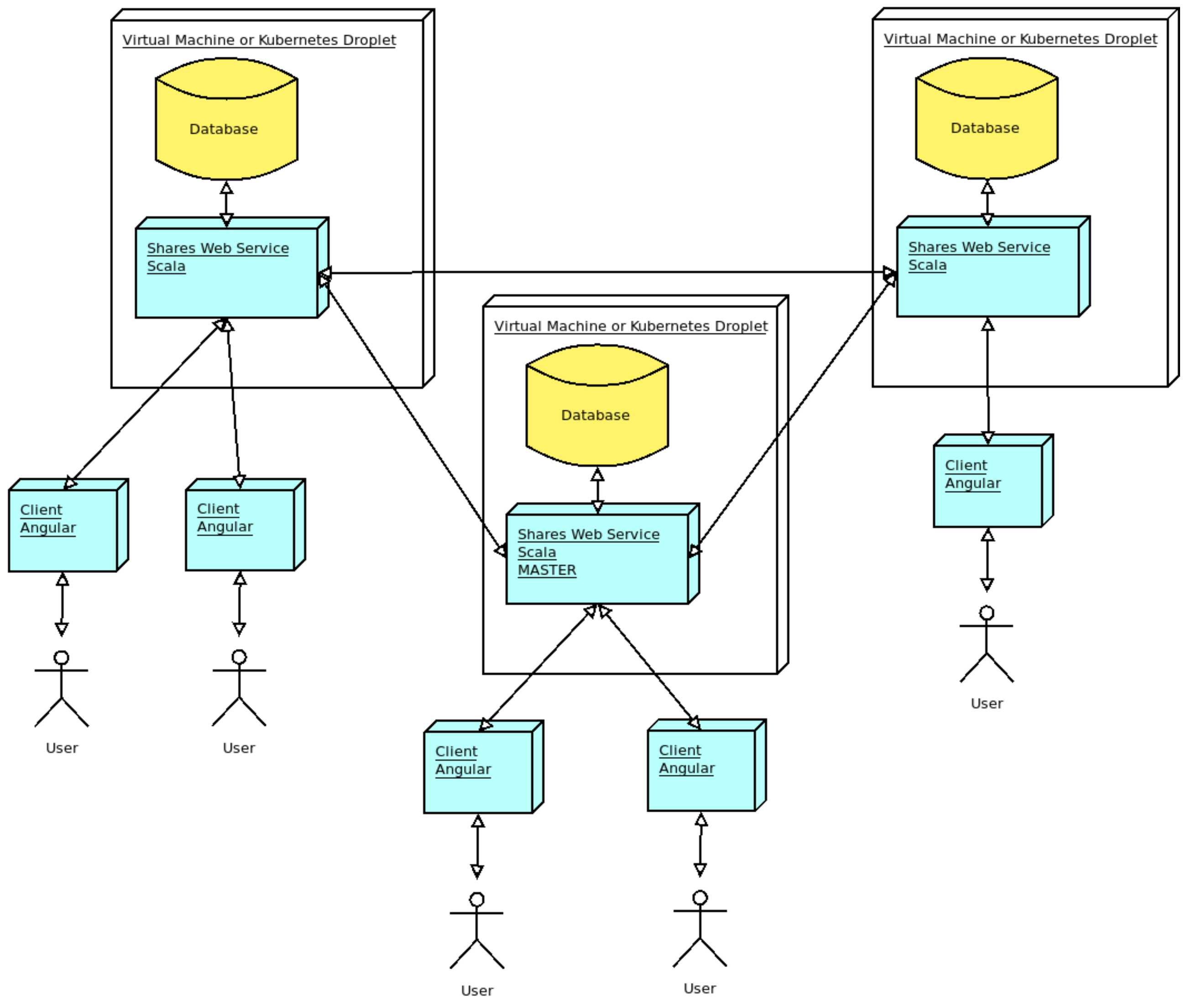
A problem with the current implementation is that user clients can become out of sync with the service, as their view does not update unless they interact with the system. This would give them an inconsistent view of the system. One possible solution to this problem is polling. The client would periodically check for updates. A problem with this is that either the client only updates occasionally, or the client will be making many fresh requests. An advantage of this method would be that, if deployed on a small enough scale that the servers could handle the requests, no change to the server side would be necessary. A disadvantage is that large scale usage would require the server scaled up too. Using long polling, keeping the polling requests open longer so that fewer need to be sent, would reduce the number of requests, but is considered bad practice because it wastes CPU, memory, and bandwidth (baasil.io). A better solution is to open a socket. A TCP Socket could be used, but these have the risk of being blocked by firewalls. Instead, a WebSocket could be used, which would allow for the client to be updated in real time. In addition, this is straightforward to implement in Angular (Aviles, L., 2018).

Converting the Shares Web Service to a distributed solution

Regardless of whether polling or WebSockets is used, if the user views are constantly being updated, far more GET requests (or WebSockets) will be made. As demonstrated above, more requests will likely worsen response times. These GET requests would modify information in the system, but only retrieve information from it. By having multiple deployments with synchronised databases, these requests can be processed in parallel. In addition, if one of these deployments goes down, the request can be processed by a different one. As the Shares Web Service is written in Scala, a JVM Language, it can be run on any device which can run Java, provided it has sufficient resources.

Below, a possible setup is illustrated. In this setup, there would always be a master server. The other servers would keep their database synchronised with this at all times. If it went offline, a different server would take over as master. Clients would manage load balancing. GET requests would be handled as normal, but POST, PUT, and DELETE requests, which all modify the database, would be relayed to the current master server. The other servers would then update from the master.

A particular challenge with implementing this solution would be ensuring that the Users can buy and sell shares extremely quickly. The solution as diagrammed could run into the problem that users connected to the master are able to buy and sell shares more quickly, and gain an advantage in this way. A possible solution to this problem would be the usage of Blockchain technologies to create a decentralised ledger.



Possible Multiple Deployment Setup (Currency Converter and External APIs not displayed in order to prevent cluttering of the diagram)

Application of Semantic Web and Linked Data Technologies

By making use of Semantic Web and Linked Data technologies, smarter, meaning driven features could be added to the application. One application of this would be in search, where at the moment the program only checks the literal data in the database for the search term given. Currently, "Bank" used as a search-term will return "The Royal Bank of Scotland", but not "Barclays". Furthermore, a search for "Financial Institution" will return neither. By building a domain ontology, and classifying companies within it, it becomes possible to store large amounts of data about the real-world nature of the company.

Domain Exploration and Analysis

In the process of developing this ontology, the first step is to define its domain and scope. Noy and McGuinness suggest sketching out competency questions that the ontology should be able to answer, in order to set its scope. These can then be used to test the ontology against later. Because this report is exploring the application of ontologies to search, instead of setting normal form questions, typical queries will be listed instead.

Domain: Companies on the stock market

Example Queries:

#	Term
1	SVT
2	Utility
3	Insurance Provider
4	Finance
5	Financial Institution
6	Bank
7	Investment
8	London Based
9	Invests in other companies
10	Car Insurance Provider

Noy and McGuinness then suggest considering reusing existing ontologies, in the case of this domain, there already exists an ontology for company data, the euBusinessgraph ontology (euBusinessGraph, 2018). By using an existing ontology such as this, not only can its existing data be made use of, but also the data in the ontologies which it references. The euBusinessGraph refers to twelve other ontologies (euBusinessGraph, 2018), which in turn also refer to further ontologies. However, if it is necessary to describe relationships which do not exist, a new ontology must be developed. euBusinessGraph does not just define the ontology, but provides a large set of linked data making use of it.

For this application, an new ontology could be developed, but still making use of relationship information from existing ontologies by referencing them. By categorising and sub-categorising the companies within this ontology, it becomes possible to know that RBS is not only a bank, but also a financial institution, and a company. By using the dbo:Company class from dbpedia, instead of a locally defined Company class, the information in dbpedia can also be made use of, for example that the dbo:Company is a dbo:Organisation, and may have a dbo:netIncome.

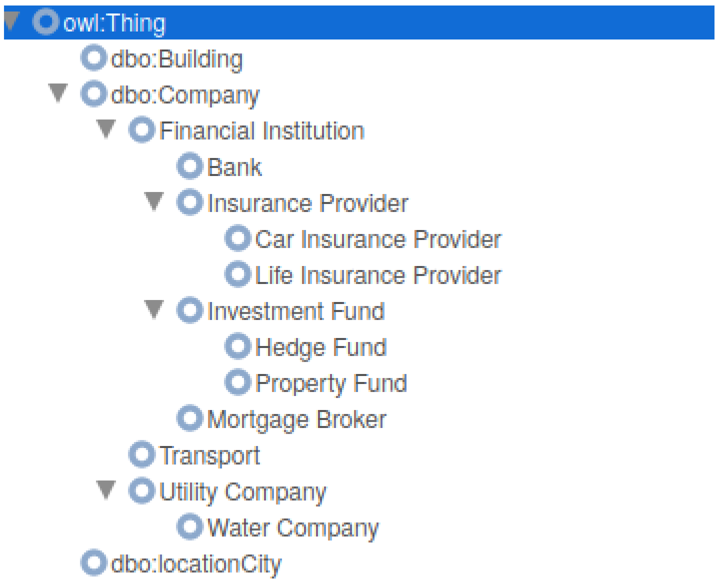


Figure 1: Ontology developed in WebProtégé

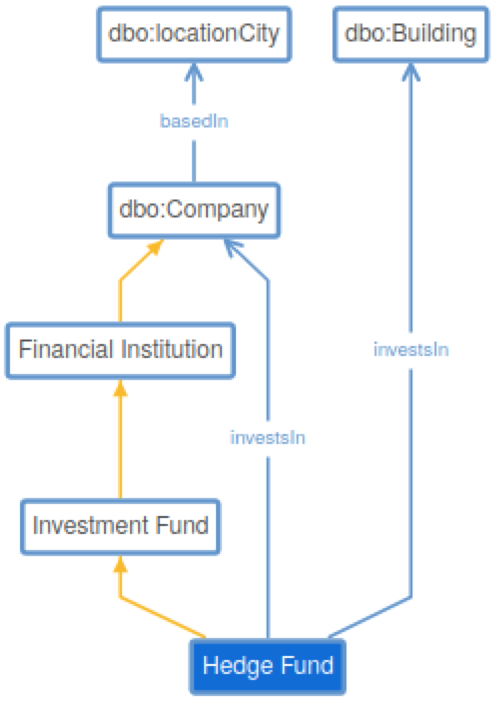


Figure 2: Graph of the relationships of a Hedge Fund

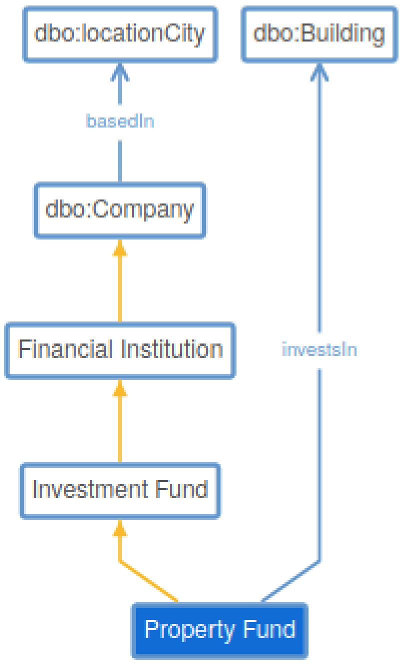


Figure 3: Graph of a the relationships of a Property Fund

To explore this further, a prototype Ontology has been developed. See figures 1-3. See Appendix C for a link to the full ontology.

If companies in the system were grouped in this ontology, it would be possible to determine that when a user searches for “Financial Institution”, they may mean “Bank”, or they may mean “Hedge Fund”.

The existing linked data set, WordNet, could be used to check for synonyms to search terms. For example, if the user enters “financial organisation”, a search of WordNet then reveals that is a synonym of financial institution (Princeton.edu, 2020). By taking advantage of existing online data, the burdens of gathering, processing, and storing data can be reduced.

Semantic Tagging

Once the ontology has been built, the existing shares data would have to be tagged according to the ontological classes which it is a member of. This could be done automatically by making use of existing linked data sets, or this could be done manually. By making use of the linked data provided by euBusinessGraph or similar, large amounts of existing data could be made use of without having to gather or format it.

Reasoning

To use the ontology, a reasoning engine must be developed. The reasoning engine should produce appropriate results for the example queries previously defined. To identify the best fits for the query, statistical query expansion could be used (Osman, T et al. 2007). A similar statistical model could be used here, using the following tests to score the shares according to degree of similarity to search.

- In the case of keywords which are the names of classes in the ontology, the engine should award a high score. If there are subclasses of this class, the members of these should be returned as well, albeit with lower scores the more levels of abstraction between the classes.
- If no exact match is found, the synonyms of the words, retrieved from WordNet, should be used to retry the query. Synonyms similarity scores can be retrieved, and the more remote the synonym the lower the score given to results found using it.
- Queries containing verbs (or their synonyms) expressing relationships which are within the ontology should be turned into logical tests. e.g. a query of "London Based", should cause a check to be run to find shares of companies with a basedIn relationship with the dbo:locationCity of London, and returned shares should have their scores improved.
    - In the prototype ontology, a Hedge Fund is a dbo:Company can invest in other dbo:Companies. As such it becomes necessary for the engine to be able to differentiate between subject and object. e.g. between "invests in companies" (a search for the subject) and "invested in by companies" (a search for the object)

Once the tests have been carried out, the highest scoring results should be returned, ranked in order of degree of match.

References

Aviles, L. (2018). *Real Time Apps with TypeScript: Integrating Web Sockets, Node & Angular*. [online] Medium. Available at: https://medium.com/dailyjs/real-time-apps-with-typescript-integrating-web-sockets-node-angular-e2b57cbd1ec1 [Accessed 29 Jan. 2020].

Bizer, C. and Heath, T. (n.d.). Special Issue on Linked Data. [online] International Journal on Semantic Web and Information Systems. Available at: https://eprints.soton.ac.uk/271285/1/bizer-heath-berners-lee-ijswis-linked-data.pdf.

euBusinessGraph. (2018). *Ontology for Company Data - euBusinessGraph*. [online] Available at: https://www.eubusinessgraph.eu/eubusinessgraph-ontology-for-company-data/.

Exchangeratesapi.io. (2018). *Foreign exchange rates API with currency conversion*. [online] Available at: https://exchangeratesapi.io/ [Accessed 29 Jan. 2020].

Gros-Dubois, J. (2016). *The Myth of Long Polling*. [online] Medium. Available at: https://blog.baasil.io/why-you-shouldnt-use-long-polling-fallbacks-for-websockets-c1fff32a064a

Noy, N. and Mcguinness, D. (n.d.). *Ontology Development 101: A Guide to Creating Your First Ontology*. [online] Available at: https://protege.stanford.edu/publications/ontology_development/ontology101.pdf.

Osman, T and Thakker, D et al. (2007) An Integrative Semantic Framework for Image Annotation and Retrieval

World Trading Data (2020). *Free Real Time, Intraday, Historical Stock and Forex Data | World Trading Data*. [online] Worldtradingdata.com. Available at: https://www.worldtradingdata.com/ [Accessed 29 Jan. 2020].

Princeton.edu. (2020). WordNet Search - 3.1. [online] Available at: http://wordnetweb.princeton.edu/perl/webwn?o2=&o0=1&o8=1&o1=1&o7=&o5=&o9=&o6=&o3=&o4=&s=financial+organisation [Accessed 29 Jan. 2020].

Bibliography

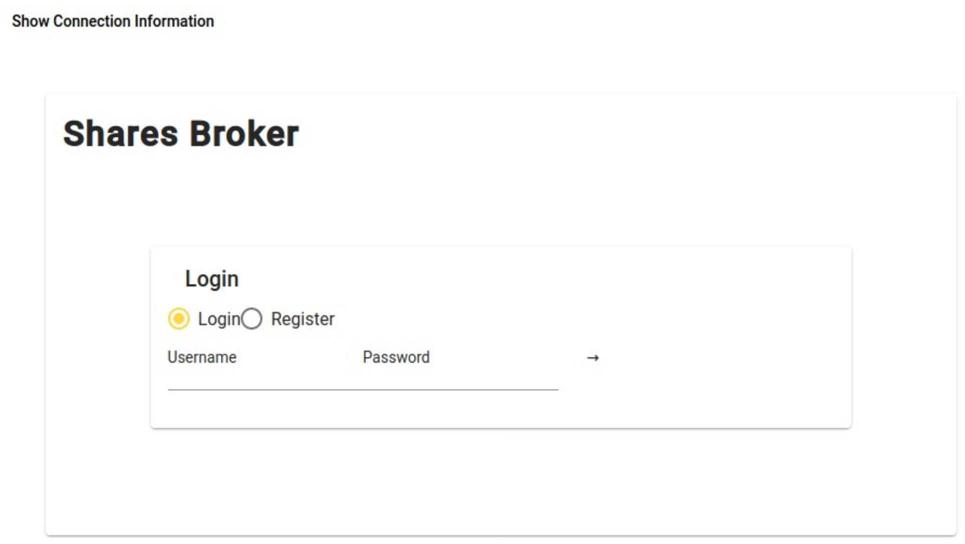
Alain Chautard (2019). *How to use WebSockets with RxJS and Angular?* [online] Medium. Available at: https://blog.angulartraining.com/how-to-use-websockets-with-rxjs-and-angular-b98e7fd8be82 [Accessed 29 Jan. 2020].

Brin, S. and Page, L. (1998). The anatomy of a large-scale hypertextual Web search engine. Computer Networks and ISDN Systems, 30(1–7), pp.107–117.

Hanson, J. (2014). *What is HTTP Long Polling?* [online] PubNub. Available at: https://www.pubnub.com/blog/http-long-polling/ [Accessed 29 Jan. 2020].

Appendices

Appendix A



Show Connection Information

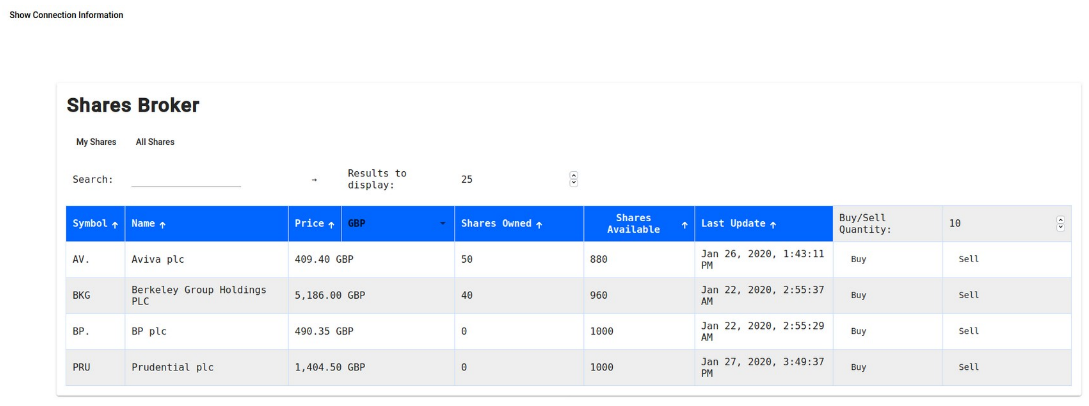
Shares Broker

Login

Login Register

Username Password →

A1: Login Page



Show Connection Information

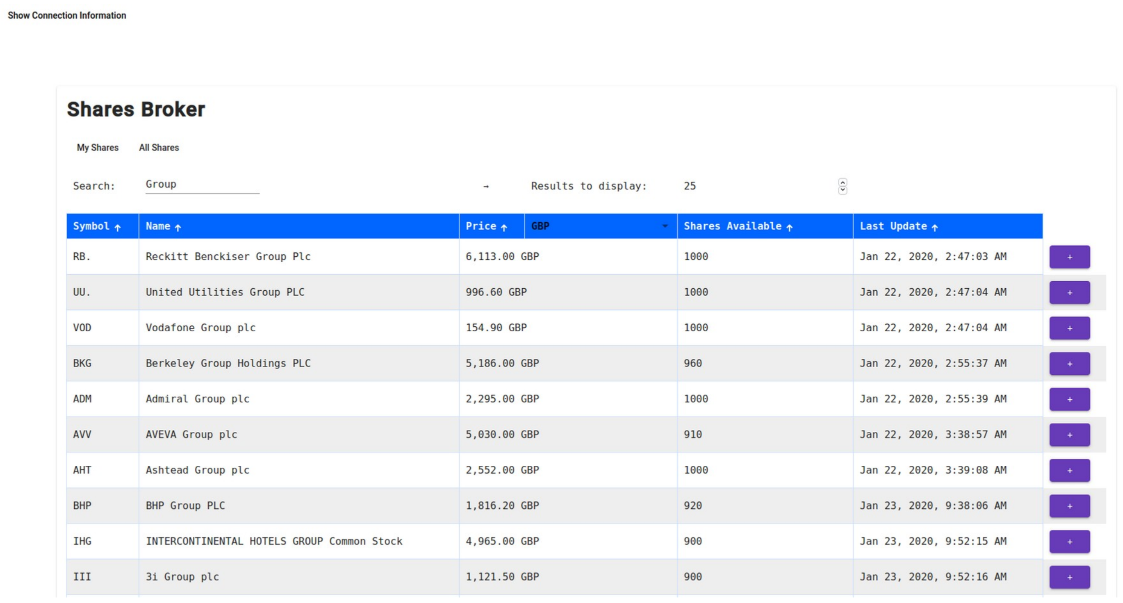
Shares Broker

My Shares All Shares

Search: → Results to display: 25

Symbol ↑	Name ↑	Price ↑	GBP	Shares Owned ↑	Shares Available ↑	Last Update ↑	Buy/Sell Quantity:	10
AV.	Aviva plc	409.40 GBP		50	880	Jan 26, 2020, 1:43:11 PM	Buy	Sell
BKG	Berkeley Group Holdings PLC	5,186.00 GBP		40	960	Jan 22, 2020, 2:55:37 AM	Buy	Sell
BP.	BP plc	490.35 GBP		0	1000	Jan 22, 2020, 2:55:29 AM	Buy	Sell
PRU	Prudential plc	1,404.50 GBP		0	1000	Jan 27, 2020, 3:49:37 PM	Buy	Sell

A2: User Shares Page



Show Connection Information

Shares Broker

My Shares All Shares

Search: Group → Results to display: 25

Symbol ↑	Name ↑	Price ↑	GBP	Shares Available ↑	Last Update ↑	
RB.	Reckitt Benckiser Group Plc	6,113.00 GBP		1000	Jan 22, 2020, 2:47:03 AM	+
UU.	United Utilities Group PLC	996.60 GBP		1000	Jan 22, 2020, 2:47:04 AM	+
VOD	Vodafone Group plc	154.90 GBP		1000	Jan 22, 2020, 2:47:04 AM	+
BKG	Berkeley Group Holdings PLC	5,186.00 GBP		960	Jan 22, 2020, 2:55:37 AM	+
ADM	Admiral Group plc	2,295.00 GBP		1000	Jan 22, 2020, 2:55:39 AM	+
AVV	AVEVA Group plc	5,030.00 GBP		910	Jan 22, 2020, 3:38:57 AM	+
AHT	Ashtead Group plc	2,552.00 GBP		1000	Jan 22, 2020, 3:39:08 AM	+
BHP	BHP Group PLC	1,816.20 GBP		920	Jan 23, 2020, 9:38:06 AM	+
IHG	INTERCONTINENTAL HOTELS GROUP Common Stock	4,965.00 GBP		900	Jan 23, 2020, 9:52:15 AM	+
III	3i Group plc	1,121.50 GBP		900	Jan 23, 2020, 9:52:16 AM	+

A3: All Shares page

Appendix B

B1: Currency Rate Request Mean Response Time and Failure Rate with respect to number of concurrent requests raw data

Requests	Absolute Failure Count	Mean Response Time (ms)	Failure Rate (%)
1000	0	1058	0.00%
2000	0	2216	0.00%
4000	0	3701	0.00%
8000	561	5017	7.01%
12000	1599	6851	13.33%
16000	4873	8720	30.46%
20000	4969	10474	24.85%

B2: Mean Response Rate with respect to payload size

	Number of Shares Requested	**Total**	**OK**	**Min**	**50th pct**	**75th pct**	**95th pct**	**99th pct**	**Max**	**Mean**	**Std Dev**
Get 2 Shares	2	1000	1000	11	1007	1483	1911	1964	1977	1020	568
Get 5 Shares	5	1000	1000	13	1007	1519	1878	1921	1950	1019	582
Get 10 Shares	10	1000	1000	12	1297	1794	1974	1983	1988	1228	595
Get 15 Shares	15	1000	1000	15	1140	1643	2064	2106	2144	1117	627
Get 20 Shares	20	1000	1000	16	1277	1686	1968	2049	2078	1183	579
Get 25 Shares	25	1000	1000	15	1156	1731	2182	2228	2246	1178	663
Get 30 Shares	30	1000	1000	54	1145	1736	2136	2166	2186	1153	634
Get 40 Shares	40	1000	1000	23	1160	1776	2266	2287	2320	1188	670
Get 50 Shares	50	1000	1000	26	1420	1955	2449	2505	2525	1356	717

Appendix C

Link to WebProtégé Ontology:

https://webprotege.stanford.edu/#projects/3e65a232-6382-46ee-8eae-223a7241db5b