



# Content Management System for a Real-Time e-Commerce Marketplace

Final Report

CM3203 - One Semester Individual Project, 40 Credits

Author: Tom Wynne-Owen  
Student Number: C1530734

Supervisor: Natasha Edwards  
Moderator: Xianfang Sun

## Abstract

The aim of this project is to create an application that provides consumers with the ability to buy and sell products between one another in a real-time marketplace similar in style to the stock market. The system allows users to place bids and asking prices on products to buy or sell products. The solution has been implemented as a web application.

This document details the entire developmental life-cycle of the system, from background research to implementation and evaluation. I have carried out background research and identified key aims and limitations for the project. Throughout the report I compare and contrast the options available and justify the final decision based on the aims and limitations.

## Acknowledgements

Firstly, I would like to express my gratitude to my supervisor Dr. Natasha Edwards for her invaluable guidance and support throughout the completion of this project.

I would also like to thank my family for their continued support and allowing me to be where I am now, particularly my father for finding the time in his schedule to help me at such short notice.

# Contents

<b>Abstract</b>	<b>1</b>
<b>Acknowledgements</b>	<b>2</b>
<b>1. Introduction</b>	<b>6</b>
<b>2. Background</b>	<b>8</b>
2.1. The Problem	8
2.2. Aims & Objectives	8
2.3. Constraints	10
2.3.1. Resource limitations	10
2.3.2. Time limitations	10
2.4. Potential Stakeholders	11
2.5. Current Solutions	11
2.5.1. osCommerce	12
2.5.2. PrestaShop	13
2.5.3. eBay	14
2.5.4. StockX	14
2.6. Choosing Tools and Methods	14
2.6.1. NodeJS	14
2.6.2. Real-Time Functionality	16
Polling and Piggybacking	16
Comet	17
Socket.io	18
Real-time Databases	18
Data Stream Networks	19
Conclusion	20
2.6.3. Choosing the Database	20
2.7. Trust in e-Commerce	21
<b>3. Specification &amp; Design</b>	<b>22</b>
3.1. Requirements Specification	22
3.1.1. Must	22
3.1.2. Should	27
3.1.3. Could	29
3.1.4. Will Not	31
3.1.5. Non-functional Requirements	32
Usability	32

Performance & Scalability	32
Security	32
Portability	32
3.2. Design	33
3.2.1. Use Case Diagrams	33
Buy from an Asker	34
Sell to a Bidder	35
Place a New Bid	36
Place a New Ask	37
3.2.2. User Interface Design	38
Homepage	39
Product page	40
Login	41
Register	42
Profile area	43
3.2.3. Application structure	45
The Final Application Structure	46
3.2.4. Database Design	49
3.3. Risk Analysis	52
1. Data Loss	53
2. Overly Optimistic Schedule	53
3. Absence or Illness	53
4. Unclear Specification	53
5. Changeability	54
6. Insufficient Skill set	54
<b>4. Implementation</b>	<b>55</b>
4.1 Authentication	55
4.2. Placing a Bid or an Ask	57
4.3. PubNub	59
4.4. Security & Validation	60
4.5. Placing Orders	61
4.6. Problems Encountered	62
<b>5. Results &amp; Evaluation</b>	<b>63</b>
5.1. Changes to the Initial Plan	63
5.1.1. Application Structure	63
5.1.2. PubNub Configuration	63

5.2. Testing Functional Requirements	64
5.2.1. Exploratory Testing	64
5.2.2. Test Cases	65
5.3. Testing Non-Functional Requirements	66
5.3.1. Usability	66
5.3.2. Performance & Scalability	66
Performance	66
Scalability	67
5.3.3. Security	67
5.3.4. Portability	68
<b>6. Future work</b>	<b>69</b>
6.1. Payment processing	69
6.2. Product Categories and Variants	69
6.3. Using the Data	70
6.3. Portfolios	70
6.4. Monetisation Strategies	71
6.5. Usability Testing	71
<b>7. Conclusion</b>	<b>72</b>
<b>8. Reflection on Learning</b>	<b>73</b>
<b>Appendix</b>	<b>75</b>
A: Instructions	75
Prerequisites	75
Install & Run	75
B: Test Cases:	76
<b>References</b>	<b>87</b>

# 1. Introduction

The purpose of this project is to design and build a secure, scalable, and extensible e-Commerce content management system that enables users to buy from and sell to one-another using a real-time bidding marketplace, similar in style to the stock market, where supply and demand determine the price of the item. To achieve this, the purpose of the project has been separated into five key aims, detailed in section 2.2. These aims are defined by the motivation for the system: to simplify and increase the participation in consumer-to-consumer e-commerce.

The vast majority of consumer-to-consumer e-commerce platforms, a popular example being eBay, rely on the user to browse through potentially hundreds of listings for the same product to find a suitable one to purchase, and even then it can be hard to judge whether the asking price is fair. The motivation for this system is to simplify and speed up the process of buying and selling items in the marketplace, to increase participation in consumer-to-consumer commerce. According to research conducted by Talkmobile, *15 million Britons have an old tech item left in their home, worth an estimated combined £300 million* [1]. Reducing the time investment and complexity of buying and selling products on the second hand marketplace could potentially massively increase participation, and the proposed system would eliminate this need to browse through hundreds of listings, in favour of having a single page for a single product, where you purchase from any seller - significantly reducing the time it takes to purchase items.

This approach to commerce is not a novel one, it has been used by the stock market since its inception, however until recently it has not been successfully adopted into an online marketplace for selling physical items. At present there is no available open-source e-commerce content management system that addresses this style of commerce. Popular e-commerce content management systems available today, such as Magento and PrestaShop, focus on the traditional business-to-consumer or business-to-business style of commerce. While there are no available solutions for a business to deploy on their own, there are proprietary platforms that do facilitate this style of commerce. One such example is StockX, a proprietary platform that specialises in the market for luxury clothing, shoes and watches (Section 2.5.4).

Deploying a system of this nature presents various opportunities to create enormous value. One option is to directly monetise the system through methods such as commission fees or membership fees (section 2.4), another is the ability to leverage the magnitude of structured, consistent, temporal market data that the system produces.

This data would enable rich analysis of the marketplace in a way that existing consumer-to-consumer solutions such as eBay cannot offer, as their listings system is by nature incongruent with this application.

This report details the successful specification, design, implementation and testing of such a system. The approach I have taken is to implement it as a web application, with design considerations to make it a robust, extensible and scalable application.

## 2. Background

### 2.1. The Problem

As stated in the introduction, buying and selling items on the second-hand market requires users to search through potentially hundreds of listings, all with different prices and descriptions for the same product. Take, for example, a new product release such as an iPhone; during the weeks after a product launch, it is often reported that the product is out of stock and is being sold on third-party platforms such as eBay or Gumtree. If you wished to purchase the product on one of these platforms you would likely have to browse and decide between hundreds of product listings for the same product, all with varying prices, titles, descriptions, images, etcetera, the wide variation of listings and prices can make it difficult to judge the actual value of the product and if you are paying a reasonable price.

A solution to this problem would be to have just one place where you could go to get all of the information you needed before making a purchase or sale. This page would contain data on asking prices, bids and completed transactions as they are made, allowing you to create a real-time, data-driven summary of the market for any product, and performing real-time analysis of the marketplace, such as deriving the value of items using the bid-ask spread.

### 2.2. Aims & Objectives

The overall purpose of this project is to create a system that provides the ability for users to buy and sell products in real time in a bidding marketplace. To achieve this goal the system will be implemented as a web-application, as this allows the system to be platform independent, enabling a larger user base to use the system.

**Aim 1:** Create a complete application that enables a user to purchase, sell, bid on or submit an asking price for multiple products.

This is the aim that defines core functionality and the shape of the application and guides the functional requirements specified in section 3.2. This aim outlines the minimum requirements for the project to be complete.

**Aim 2:** Create a secure, robust and scalable system, without compromising usability or performance.

The system should be designed and implemented in a way that theoretically it could be expanded upon and deployed into production. This means that the final product should provide core functionality that is robust and scalable. The specification and design must take this aim into account these considerations at every stage. This aim is important to ensure that the work completed within the project has a purpose and can be further developed into a system that can be deployed and used.

**Aim 3:** Design and implement the system in such a way that it is easily maintainable, extensible, and compatible with a variety of popular platforms, including desktop and mobile.

As detailed in the previous aim, the system is designed to be able to be deployed in the future, so in addition to being robust and scalable, it needs to be maintainable and extensible. This means that best practices should be used and the code should be well commented, consistently styled and concise. Additionally it needs to be compatible with as many devices as possible so that the user-base is not limited in size. The reason for this aim is to ensure that continuation of the development of the project in the future is as easy and efficient as possible.

**Aim 4:** Enable the communication of new bids and asks in real-time (less than one second)

If the system does not facilitate real-time updates without page refreshes then this will limit the business case for the application. If bids, asks and orders are coming in quickly, requiring constant page refreshes from the user will severely affect the system's effectiveness and usability.

**Aim 5:** Design and implement a simple and usable frontend user interface for the system.

The initial plan for the project stated that a project aim was to create and test a user interface that follows good HCI principles, however as the project developed, the scope has shifted to focus on the backend. It is, however, crucial to provide a usable front-end, as if the system is confusing or difficult to use then it will fundamentally fail to address the problem defined in section 2.1. If the time it takes to learn how to use the system is too long then the platform will have no real value, as users would rather spend the time to use competing platforms such as searching through listings on eBay.

## 2.3. Constraints

Due to the time and resource limitations of the project, there are a number of constraints to final product. Efforts have been made in the specification and design of the project to reduce these limitations as much as possible and to make them easy to address and correct where applicable in the future.

### 2.3.1. Resource limitations

Testing the scalability and robustness of the system is infeasible in the project as I do not have the resources to deploy the system at scale. Even though this constraint is in place, considerations have been made in the specification and design of the system to try and make sure that when the system is extended and then deployed in the future, it is scalable, secure and robust.

Additionally, some security features, such as obtaining an SSL certificate for HTTPS connections require resources that are not available to me for this project. While these limitations exist, the specification and design of the system has taken these factors into account when making decisions, for example the key reason NodeJS was used was due to its non-blocking I/O design.

### 2.3.2. Time limitations

While creating a simple user interface is an aim for the project, the scope of the project focuses on creating an extensible backend to facilitate this type of marketplace instead of the user interface. This alteration to the scope was made due to the time constraints on the project, this means that designing, testing and implementing a user interface for the system falls outside of the scope. Additionally, the timeframe limited my ability to complete full compatibility testing for the range of client devices that could use the system.

The completed project also does not have sufficient security and validation mechanisms to be used in production, particularly for handling user data and payment processing. As a result there is no method to make payments with the system, as this falls outside of the scope of the project. This constraint is a combination of resource and time limitations on the project, as NodeJS provides the ability natively and with NPM modules to implement secure systems for the most part, however as mentioned in section 2.3.2 some features such as SSL require more resources to implement.

Additionally, the time must be taken to penetration test the system to verify that the system is secure.

## 2.4. Potential Stakeholders

There are 15 million Britons with unused electronic devices in their homes[1] that could potentially be traded on the platform. The platform has two key stakeholders: consumers who are looking to purchase products second-hand, and consumers who have unused products that they want to sell. The platform can both enable buyers to make a more informed purchase in less time than current solutions and enable sellers to make a faster and easier sale.

Deploying the system would present a number of different models for monetisation, these could include models such as charging commission for each transaction, charging for bids and asks, or charging a membership fee. Implementing these features falls out of the scope of the project however it would be possible to add them in the future. These features have been discussed in more detail in section 6.4.

## 2.5. Current Solutions

At present there are few content-management systems that address this specific problem, the majority of available to download or purchase e-commerce CMS packages focus on a business-to-x model, whereas the focus of this project is a consumer-to-consumer based e-commerce system. Whilst the available open-source solutions are not a perfect match, evaluating them can still provide some valuable guidance on the approach to the design and implementation of my solution as well as highlight some novel solutions to sub-problems and appropriate best practices.

There are however, closed source platforms such as eBay, Gumtree and StockX that offer consumer-to-consumer commerce. The most relevant example is that of the latter - StockX, a platform that offers the same style of bidding marketplace for luxury Handbags, Shoes and Watches.

### 2.5.1. osCommerce

Link: <https://www.oscommerce.com/>

Licence: GNU GPL

osCommerce is a free, open source e-commerce solution that anyone can download themselves and install on their own server. It is written entirely in PHP, and requires a MySQL database to function. It is one of the oldest and longest running e-commerce solutions, boasting an age of 18 years. This maturity gives it a strong selection of over 8000 free themes, extensions and modifications, as well as a strong community of over 300,000 users that are active in the product forum. Whilst the system is completely open source and free for anyone to download, osCommerce monetises the product with a partnership with their hosting partner: '1&1'.

While osCommerce is not the same style of real-time marketplace content management system that this project aims to create, it does offer some valuable insight into the implementation of features such as creating reports and promotions, as well as best practices for storing product, user and order data.

osCommerce is a fully featured e-commerce tool that you can run a business on. Additionally the functionality can be extended dramatically with free, open source extensions available on their website. The standard osCommerce package has the following key features:

- Product catalogue management. Here you can have as many products as you want, and you can define as many categories and product attributes as needed.
- Simple inventory control that consists of a stock counter.
- Customer reviews on products.
- Creating special offers and coupons.
- Customer accounts and customer management.
- Currency, language and local tax settings.
- Powerful report builder.
- Order processing features.
- Database backups, security checks.

Unfortunately the standard osCommerce theme that comes with the package is extremely dated, necessitating a different theme to be used, however there is a large selection of themes available for free from the add-ons repository. In addition, the

standard package also misses some important features, such as an inventory management system, or a way of providing different shipping options.

## 2.5.2 PrestaShop

Link: <https://www.prestashop.com/en>

Licence: OSL v3

PrestaShop is another free, open source business-to-x style e-commerce system written in PHP with a MySQL database. The base software package is completely free to download and install on your own server, however PrestaShop monetises its product with its Addons Marketplace, where users can purchase themes, extensions and services for the base package. PrestaShop's product catalogue management and inventory control system is more powerful than that of the osCommerce system. Additionally it has a search engine optimisation tool for products in your catalogue and powerful real-time analytics tools that provide useful data on traffic and sales figures.

PrestaShop has a large and active community to support the product, their website boasts a community of one million members. The software facilitates the following key features:

- Versatile product catalogue management, products can be divided into categories - which you can define however you want. PrestaShop also provides some SEO tools for products. The software also has a bulk import tool to import product listings from CSV files or Excel spreadsheets.
- Powerful order management allows you to view, edit, cancel and delete orders after they have been placed, for example discounts can be applied to orders after they have been placed.
- Inventory control features that can track inventory and give restocking alerts based on sales data.
- Powerful real-time analytics and extra analytic features such as abandoned cart information, customized promotions and an automatic email system.

Whilst PrestaShop is a powerful e-commerce tool, it does require some expertise in web technologies to setup the software and database, however once the software is installed on the server, the backroom panel is fairly intuitive for any user to continue setting up the online store.

### 2.5.3. eBay

Link: <https://ebay.co.uk/>

Ebay provides a platform for anyone to list almost any item for sale. Ebay allows users to post a new listing for each item they want to sell, this means that a potential buyer needs to search through tens or hundreds of listings of the same product, introducing friction in the buying process. Additionally, sellers have two different ways to list a product, either as an auction or as a fixed price 'buy it now' listing.

### 2.5.4. StockX

Link: <https://stockx.com/>

StockX is the most similar solution to the proposed problem, it is a real-time bidding marketplace that specialises in the luxury clothing, shoe and watch marketplace. The system works by allowing users to place a bid or ask on a product. If a newly placed bid is higher than the lowest ask on the product, then a transaction is automatically made where the bidder purchases the item for the lowest asking price, conversely if a newly placed ask is lower than the highest bid on the product, then a transaction is automatically made where the asker sells the item for the highest bid price.

StockX does not disclose how their system pushes new bids, asks and orders to the clients in real time as they are made, however from studying the client source code with Firefox developer tools, it appears as though they use the WebSocket protocol for this functionality, likely with a framework such as socket.io. Their service also provides historical pricing data for all of the products in its database, and they claim to gather this data from its own transactions as well as from third party sources, however it does not name these third parties.

## 2.6. Choosing Tools and Methods

### 2.6.1. NodeJS

As there are no prevalent open-source solutions addressing this specific area in e-commerce to investigate I have chosen to build the web-application from scratch. There are a vast number of options for languages and web frameworks to build the system with. I will be using a web framework for this project to make development as

fast as possible and to take advantage of the security features, maintainability and extensibility that many frameworks provide.

I have chosen to use NodeJS and the Express framework to implement the system. I chose to use NodeJS for a number of reasons, primarily because its non-blocking design increases speed and scalability. This non-blocking design means that when I/O operations occur, such as querying a database, they happen asynchronously, allowing the rest of the program to continue running whilst the query runs in the background, and when the query is complete a callback function is called[2]. This design makes NodeJS ideal to create scalable network apps that can handle receiving thousands of concurrent requests and still perform well.

Another reason I have chosen to use NodeJS is because it is a JavaScript framework, which allows me to use JavaScript across across both sides - client and server - this ubiquity means that code can be written once and used anywhere across the stack, it also increases compatibility and decreases complexity as it means everything is written in the same language, as opposed to using a PHP or Java backend paired with a JavaScript front-end, as using different multiple languages typically decreases compatibility and can make development more difficult.

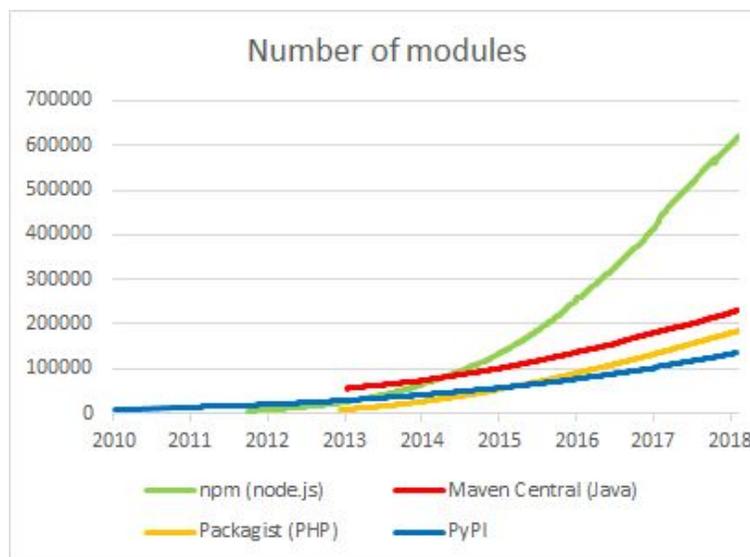


Figure 2.1: Number of unique modules in each repository over time. Data provided by Module Counts (<http://www.modulecounts.com/>).

Additionally, a key advantage of using NodeJS is because of the prevalence of it. According to the 2018 Stack Overflow Developer Survey[3], JavaScript was the most popular technology among respondents and NodeJS was the most popular framework.

Due to this popularity NPM - Node Package Manager, as you can see from figure 2.1, is one of the world's largest and fastest growing repository of modules (data provided by Module Counts [4]). This means that there is a huge amount of support from a large community and there is a massive selection of modules - notably Passport and PubNub - to use to aid development.

### 2.6.2. Real-Time Functionality

The ability to publish the new bids, asks and orders as they are made in real-time is a key requirement for the system. The first decision to be made when designing this functionality is what approach to take. There are a number of ways to achieve this functionality including, but not limited to the following:

- Polling or Piggybacking
- Comet
- Socket.io (WebSockets)
- Real-time database
- Data Stream Networks

#### Polling and Piggybacking

The first and most crude option, this would send a request to the server from the client for the latest data at a set interval. The advantage of this approach would be twofold: it would be much simpler to implement and it would be supported across a much larger selection of clients, however it cannot be classed as real-time unless the polling interval is extremely short, making it very inelegant and wasteful on server bandwidth and resources, with a large amount of clients polling the server, this can accidentally simulate the effects of a distributed denial of service attack.

Piggybacking is a very similar technique to the polling technique, however it reduces the load on the server. The way this technique differs from traditional polling techniques is that if there is no change in the server's state since the last poll, the server will instead return a message saying that there is no change to the data. The advantage of this method is that it can reduce the amount of requests that the server has to respond to, reducing load. The disadvantages of this method are that it does not scale well and there is a need to create a robust way of determining and comparing the state of the client and server. This could be achieved using timestamps or hash functions, however if the messages are small and frequent this could be less efficient than simple polling [5].

## Comet

Comet is an umbrella term for a number of techniques designed to enable the server to push messages to the client when the server's state changes. These techniques can be separated into two main classes: long polling and streaming.

Long polling is very similar to the polling technique described above, however when the client polls the server for information, the server will hold this request, keeping the request alive, and only responding once there is new data available. Once the client receives the response, it immediately sends a new long poll request and repeats this process indefinitely. This approach facilitates push data from the server, and it is much more efficient than traditional polling, however dealing with the requests from the backend can be complex, and the complexity can be exacerbated with scale.[6]

Comet streaming techniques work in a similar way to long polling, where a HTTP request is sent from the client and kept alive by the server, however with streaming techniques there will only ever be one indefinite request that is never killed. Through this request, the server can keep pushing data to the client indefinitely. Comet streaming is typically achieved in one of two ways, either using 'forever iframes' or through 'multipart XML HTTP requests'.[7]

The forever iframes technique uses hidden iframes in the HTML page that points the source to a path that the server returns events to. When an event occurs, the server clears the iframe and then writes a new script into it, which will then be executed by the client. This method has the advantage of being compatible with all browsers that support iframes (all major browsers) and is relatively simple to implement. The disadvantages of this method are that it can be difficult to determine if the connection has been broken and it can be difficult to handle errors.

Multipart XML HTTP requests use the multipart functionality for AJAX supported by some browsers, this allows an AJAX request to be returned over time in multiple parts, as and when each of the parts are sent by the server. An AJAX request is kept open and each time an event occurs on the server, the server adds a new section to the multipart response.[8]

## Socket.io

Socket.io is a JavaScript library that abstracts the WebSocket protocol to provide real-time two-way communication between client and server. Socket.io is supported on almost any modern browser as it is built with JavaScript, additionally, iOS supports socket.io natively for push notifications [26].

Socket.io is protocol agnostic, by default it will always try to use the WebSocket protocol where available however if WebSocket is not supported by either the client or server it is able to fallback on other protocols, such as traditional polling. While socket.io does do a lot to abstract the complexity of streaming data, it still requires you to host and maintain your own server infrastructure.

## Real-time Databases

*A data stream is a real-time, continuous, ordered (implicitly by arrival time or explicitly by timestamp) sequence of items. It is impossible to control the order in which items arrive, nor is it feasible to locally store a stream in its entirety. [9]*

Real-time databases, such as RethinkDB, are designed to facilitate the communication of large-scale data streams in real time. Instead of traditional database systems that require you to poll for data changes, they push data to the client as it is updated in real-time. The advantages of the RethinkDB system are that it facilitates rich real-time querying of the database using its own query language - ReQL, it natively supports pushing data to the client and it can dramatically reduce the time it takes to build real-time applications [10].

While it is a powerful query language, the plan for the integration of real-time communication for this project does not have any need for the features that ReQL offers. Another key disadvantage of RethinkDB is that, whilst it can be used as a general purpose database, it is not suitable for storing user data and authenticating users. This would make the addition of a permanent database a necessity, and running two different database systems simultaneously would introduce unnecessary complexity and present difficulties ensuring data security and integrity.

An additional concern is the unstable history of the platform. The company behind RethinkDB is a startup with an unstable history [11] and the RethinkDB software is a fairly immature, leading-edge product. As a result the product does not have the same support and compatibility that competing options offer which could lead to problems

maintaining the system in the future, making the system unable to fulfil aim 2 (section 2.2).

### Data Stream Networks

Data stream networks facilitate the communication of bidirectional data streams in real-time between an arbitrary number of users. Data stream networks are similar to real-time databases however they offer some additional features on top of real-time databases, including storing channel history, detecting presence of clients, and unicast or multicast publishing patterns.

PubNub is a data stream network service that allows you to purchase usage of their existing worldwide data stream network infrastructure and 70 APIs for different platforms instead of having to build your own. The advantage of this is that you no longer have to maintain the performance, scalability and security of the real-time data communication system and PubNub guarantees delivery of messages in 250 milliseconds or less. It is also protocol independent, it will use the WebSocket protocol wherever possible however it can fallback onto polling for clients that do not support WebSockets, maximising compatibility. All of these features directly contribute toward the fulfilment of aims 1 and 2 (section 2.2).

*PubNub utilizes a Publish/Subscribe model for real-time data streaming and device signaling which lets you establish and maintain persistent socket connections to any device and push data to global audiences in less than  $\frac{1}{4}$  of a second. [12]* Figure 2.2 shows the two patterns that PubNub facilitates for communication between users: unicast mode (left) and multicast mode(right). In unicast mode there is a one-to-many relationship with a single publisher to whom many clients can subscribe. Conversely, in multicast mode there is a many-to-many relationship in which there can be an arbitrary number of both publishers and subscribers [12].

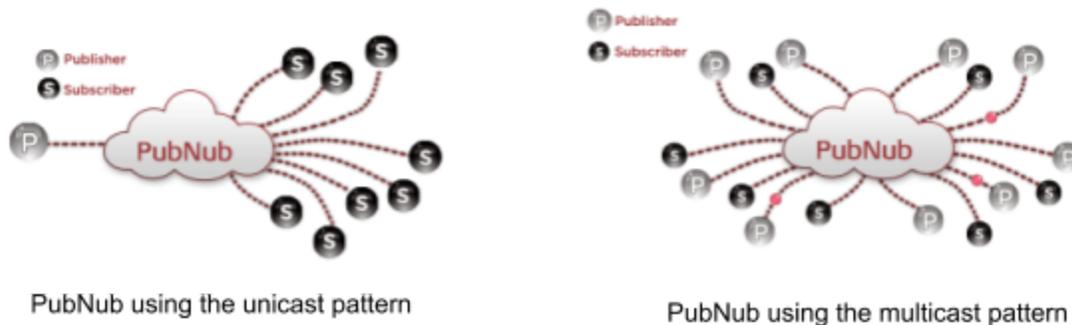


Figure 2.2: PubNub's channel design patterns for publishing and subscribing. Unicast (left) and Multicast (right). Images courtesy of PubNub [12]

PubNub also has charting features built in using its Eon chart engine, which enables rendering charts from the messages sent through a channel. Eon creates multiple different types of charts and graphs that updates in real-time as messages are published. This charting engine could be used to provide historical pricing graphs and other useful visualisations on the product page. Additionally, unlike real-time databases such as FireBase, PubNub supports the storage of channel history. This means that the historical data can be included in the real-time charts on the product page.

## Conclusion

The final decision I made was to use PubNub's infrastructure to push messages to all of the clients viewing products. PubNub has a free account option for development and testing that allows up to one hundred devices to be connected simultaneously and for 1 million messages monthly. This limit is more than enough for the development and testing of the system, however if the system is deployed the paid service will be required.

The plan calls for PubNub to be used in unicast mode, where the server publishes a message to all of the subscribers - the clients viewing the product - would allow real-time communication of new bids, asks and orders to all of the currently viewing the product.

Using PubNub for this project is advantageous because of the time constraints in place for the development of the system. To design, build and test a similar system from scratch, even if it were feasible, would take up a significant portion of the project's development time reinventing the wheel, and PubNub will be able to abstract the entire process and provide a more performant, scalable and secure service than I could deploy. Additionally, the JSON document model that JavaScript and MongoDB both use is the same model that PubNub uses for communication, which will maximise compatibility across the whole system.

### 2.6.3. Choosing the Database

As detailed in section 2.6.2, whilst PubNub does enable the storage and transmission of data that is passed through the channel, it does not, by default, store the messages permanently and it does not provide many querying options for the data. This means that PubNub is not an appropriate method for storing permanent data such as product data.

The decision I made when choosing the database was to use a document-oriented NoSQL database system: MongoDB. MongoDB is a flexible and performant database management system that has been designed to scale extremely well through the use of its sharding technology. While relational database systems such as MySQL or Oracle do have the advantage of ACID transaction management (Atomicity, Consistency, Isolation, Durability), they do not offer the flexibility in schema design and horizontal scalability that MongoDB does. These features make the database schema much easier to modify and extend after-the-fact, as MongoDB leaves much of the schema design and validation to be enforced by the program instead of the database system. To do this, NPM has a module called Mongoose, an object relational middleware for MongoDB. This allows me to define a number of schemas and relationships for my data models, and abstracts the MongoDB connection and querying mechanisms. It also offers powerful validation options for these schemas, allowing me to do any data validation within the application logic instead of the database [13].

Another benefit of using MongoDB with this project is it's JSON (JavaScript Object Notation) structure for documents. This structure makes it ideal for programs written in JavaScript - such as this system - as it uses the native notation which makes it fully compatible. Additionally and more specifically to this project, PubNub shares the same JSON notation for sending messages, which will allow me to make JSON the ubiquitous data format across the entire system.

## 2.7. Trust in e-Commerce

In general, trust in the context of consumer-to-consumer e-commerce as defined by Pavlou, Liang & Fygenson is "*a buyer's intentions to accept vulnerability based on her beliefs that transactions with a seller will meet her confident transaction expectations due to the seller's competence, integrity, and benevolence*" [14]. A crucial consideration when creating a consumer-to-consumer e-commerce system is how to create trust between users transacting with one-another and between users and the platform. One approach to this issue is for the organisation to act as a broker, having the seller ship the item to the organisation for verification and then having the organisation ship to its final destination.

Another trust consideration when using an ecommerce platform is the security and privacy of user data. If the system cannot be trusted to provide security and privacy for users' data then this will negate any consumer confidence and any benefits to the system over its competitors. This is a significant factor for the justification for the second key aim for the project, to create a secure system (section 2.2).

## 3. Specification & Design

### 3.1. Requirements Specification

To aid the implementation process, in this section I have detailed all of the functional and quality requirements for the completed system and all of its features. This list enabled me to refer to specific requirements to make decisions during the design and implementation phases.

The requirements specification has been completed with an awareness of the time and resource limitations affecting the project, and the commitment to create a scalable, robust and extensible system. Additionally, the requirements have been written in an order that I plan to implement them - allowing me to work through each one in sequence and almost use it as a todo list.

#### 3.1.1. Must

##### Requirement 1.1: Storing Product Data

###### Requirement:

The system must allow the storage and retrieval of product data from a client. It must allow the retrieval of data for an individual product or multiple.

###### Acceptance Criteria:

- The data is delivered formatted in a way that is accurate and concise.
- The system provides an appropriate error message in response if no product could be found.

##### Requirement 1.2: Adding Products

###### Requirement:

The system must allow an administrator user to add a product to the database, given required product data

###### Acceptance Criteria:

- The system ensures the user is an authenticated administrator before allowing them to add a new product.

- The system lets the administrator user to add a product title, slogan and a description.
- The system supports the uploading of one or multiple product images.

### Requirement 1.3: User Creation

#### Requirement:

The system must enable a user to create an account.

#### Acceptance Criteria:

- The system requires the user to provide a new username and password.
- The system does not allow the creation of an account with the same name as an existing account.
- The system stores the password in a secure way, by salting and hashing it.

### Requirement 1.4: User Authentication

#### Requirement:

The system must allow a user who has registered an account to log in to the account and access the account's profile.

#### Acceptance Criteria:

- The system confirms the username and password provided match a record in the database. If there is no match the system displays a message saying that the credentials were not recognised.
- The system allows a maximum of 5 failed attempts to log into an account.
- The system clearly shows the user when they are logged in and gives them an option to log out.

### Requirement 1.5: Administrator Authentication

#### Requirement:

The system must provide a method to differentiate between normal user accounts and administrator user accounts.

#### Acceptance Criteria:

- Authenticated administrator accounts are given a visible link to the administrator area.
- Non-administrator accounts must not be able to access the administrator area in any way.

#### Requirement 1.6: Place Bid

##### Requirement:

The system must allow an authenticated user who wishes to buy a product to place a bid on said product. The bid must be viewable to the public.

##### Acceptance Criteria:

- The system ensures the input is a valid number.
- When the bid has been placed, other users viewing the same product page should be notified within 1 second, with no need to refresh the page.
- The system informs the user placing the bid if their new bid will become the new highest bid for the product.

#### Requirement 1.7: Place Ask

##### Requirement:

The system must allow an authenticated user who wishes to sell a product to place an asking price to sell said product. The asking price must be viewable to the public.

##### Acceptance Criteria:

- The system ensures the input is a valid number.
- When the ask has been placed, other users viewing the same product page should be notified within 1 second, with no need to refresh the page.
- The system informs the user placing the ask if their new ask will become the new lowest ask for the product.

#### Requirement 1.8: View bids and asks on a product

##### Requirement:

The system must allow anyone to view all of the current bid prices and asking prices on a product at any time.

Acceptance Criteria:

- The system displays bids and asks in a table format on the product page.
- The data is concise and anonymised.
- The data can be sorted by either price or date, ascending or descending.

Requirement 1.9: Purchase from an askerRequirement:

The system must allow an authenticated user to place an order to purchase a product for the the lowest asking price.

Acceptance Criteria:

- The product page displays clearly what the current lowest open asking price is for a product.
- The system provides a button that allows a user to purchase from the lowest asker.
- The system closes the ask once the order has been placed.
- If there are no open asks on a product, the system provides the option to place a bid instead.

Requirement 1.10: Sell to a bidderRequirement:

The system must allow an authenticated user to accept a bid and place an order to sell the product for the agreed bid price.

Acceptance Criteria:

- The product page displays clearly what the highest open bid is for a product.
- The system provides a button that allows a user to immediately sell their item to the highest bidder.
- The system closes the bid once the order has been placed.
- If there are no open bids on a product, the system should provide the option to place an asking price instead.

Requirement 1.11: View all of your bids

Requirement:

The system must allow a user to view a list of all of the open bids they have placed on any item(s).

Acceptance Criteria:

- Each element of the list contains the product name, the bid price, the time it was placed and the status of the bid (open or closed).
- The list should be sortable by date.

Requirement 1.12: View all of your asksRequirement:

The system must allow a user to view a list of all of the open asking prices they have placed on any item(s).

Acceptance Criteria:

- Each element of the list contains the product name, the ask price, the time it was placed and the status of the ask (open or closed).
- The is should be sortable by date.

Requirement 1.13: View all of your ordersRequirement:

The system must allow a user to view a list of all of the orders they have placed in the past.

Acceptance Criteria:

- Each element of the list contains the product name, the sale price, the time the order was placed and the status of the order.
- The list is sortable by date.

### 3.1.2. Should

#### Requirement 2.1: Search the product database

##### Requirement:

The system should have the ability for a user to search for products with keywords.

##### Acceptance Criteria:

- The system searches for keywords in the product title and description.
- The system returns a list of products descending in similarity to the search term.
- If there is no match, the system informs the user that no product matched the search term.

#### Requirement 2.2: Update the product page automatically

##### Requirement:

The system should update the product page as bids, asks and orders are made.

##### Acceptance Criteria:

- The product page updates the lowest ask and highest bid values shown on the page when they are updated.

#### Requirement 2.3: Notify the user of events

##### Requirement:

The system should notify a user of the following events:

- Their open bid has been outbid by another user.
- Their open ask has been undercut by another user.
- Their bid or ask has been accepted by another user.

##### Acceptance Criteria:

- The notifications display the new bid or ask price, or order price as well as the timestamp.
- The notifications are displayed ordered by the time they occurred, with the most recent first.

Requirement 2.4: Display previous transactions on the product pageRequirement:

The system should provide a list of anonymised completed orders on the product page.

Acceptance Criteria:

- The list only contains the timestamp and the price of the order.
- The list can be re-ordered by time or price, ascending or descending.

Requirement 2.5: Allow users to search for products with keywordsRequirement:

The system should allow a user to search the database for products for keywords contained in the product title and description.

Acceptance Criteria:

- A list of most similar products to the search term is returned.
- The list contains links to the product page for each element it contains.

Requirement 2.6: Provide useful and timely data on the state of the marketplaceRequirement:

The system should display the following information on a product page:

- All of the open bids and asking prices.
- The highest open bid.
- The lowest open asking price
- The number of completed transactions.

Acceptance Criteria:

- The open bids and asking prices are displayed in a table or list that can be sorted by either time or price.
- The highest open bid on the product is clearly displayed and the user can immediately sell to the highest bidder with the click of a button.
- The lowest open ask for the product is clearly displayed and the user can immediately purchase the product from the asker with the click of a button.

Requirement 2.7: Provide useful visualisations of the marketplaceRequirement:

The system should provide a line graph displaying the prices of bids and asks over time and a line graph containing the order prices over time.

Acceptance Criteria:

- The graphs are clearly titled and labeled.
- The graphs update in real-time.

## 3.1.3. Could

Requirement 3.1: Allow third party authentication for accountsRequirement:

The system could allow users to create an account and login using third party accounts such as a Google account or Facebook account.

Acceptance Criteria:

- The system gives an option on the register page to create an account with a third party account.
- The system allows a user to login with a selected third party account.

Requirement 3.2: Send relevant push notifications to the userRequirement:

The system could use supported browsers push notification features to automatically notify users of the following events:

- A new highest bid on the product they are looking at.
- A new lowest ask on the product they are looking at.
- Their bid has been accepted.
- Their ask has been accepted.

Acceptance Criteria:

- A push notification with appropriate information is sent to the client within 5 seconds of the event occurring.

### Requirement 3.3: Automatically complete purchase

#### Requirement:

The system could automatically complete a purchase if the user places a bid that is higher than the current lowest open ask.

#### Acceptance Criteria:

- If a user inputs a bid amount that is higher than the lowest ask then they will be redirected to a checkout page where they can immediately purchase the product for the lowest asking price.
- The user is informed before submitting that they will instead immediately purchase the product.
- The new bid is not placed.
- The asking price is closed when the order is complete.

### Requirement 3.4: Automatically complete sale

#### Requirement:

The system could automatically complete a sale if the user places an ask that is lower than the current highest open bid.

#### Acceptance Criteria:

- If a user inputs an ask amount that is lower than the highest bid then they will be redirected to a checkout page where they can immediately sell the product for the to the highest bidder.
- The user is informed before submitting that they will instead immediately sell the product.
- The new asking price is not placed.
- The bid is closed when the order is complete.

### Requirement 3.5: Update Product details

#### Requirement:

The system could allow an admin to edit the static details of a product.

#### Acceptance Criteria:

- The admin settings area has a list of products in the system each with an edit button.
- When the edit button is clicked, the system will show a view with a form to change the name, slogan, description and product image.
- The changes made take effect immediately.

### 3.1.4. Will Not

### Requirement 4.1: Do not allow unauthenticated users to place bids, asks or orders

#### Requirement:

The system will not allow any user who is not logged in to place an ask, bid or order on any product.

#### Acceptance Criteria:

- When a user who is not logged in tries to place a bid, ask or order they are redirected to the login page.

### 3.1.5. Non-functional Requirements

#### Usability

##### *Acceptance Criteria:*

- A new user should be able to understand how to place a bid, ask, purchase or sell a product after 5 minutes of use without help.
- The user interface should be consistent and give concise and informative feedback to the user.

#### Performance & Scalability

##### *Acceptance Criteria:*

- The system should load a page in 5 seconds or less.
- The system should push new bids and asks to users viewing the product page in real-time without a page refresh. Real-time is defined as less than one second.
- The database schema should adhere to MongoDB's guidelines for scalability.

#### Security

##### *Acceptance Criteria:*

- The system must store passwords in a secure way in which they can never be read by anyone.
- The system must require users to authenticate themselves before placing bids, asks and orders.

#### Portability

##### *Acceptance Criteria:*

- The system can be accessed and used with the four largest desktop browsers: Chrome, Internet Explorer, Firefox and IE/Edge. [15]
- The system can be accessed and used with the two largest mobile browsers: Safari and Chrome

## 3.2. Design

### 3.2.1. Use Case Diagrams

Figure 3.1 provides a high level view of all of all of the actions a user can perform with the system. Additionally, key use cases have been expanded in their own separate diagrams below.

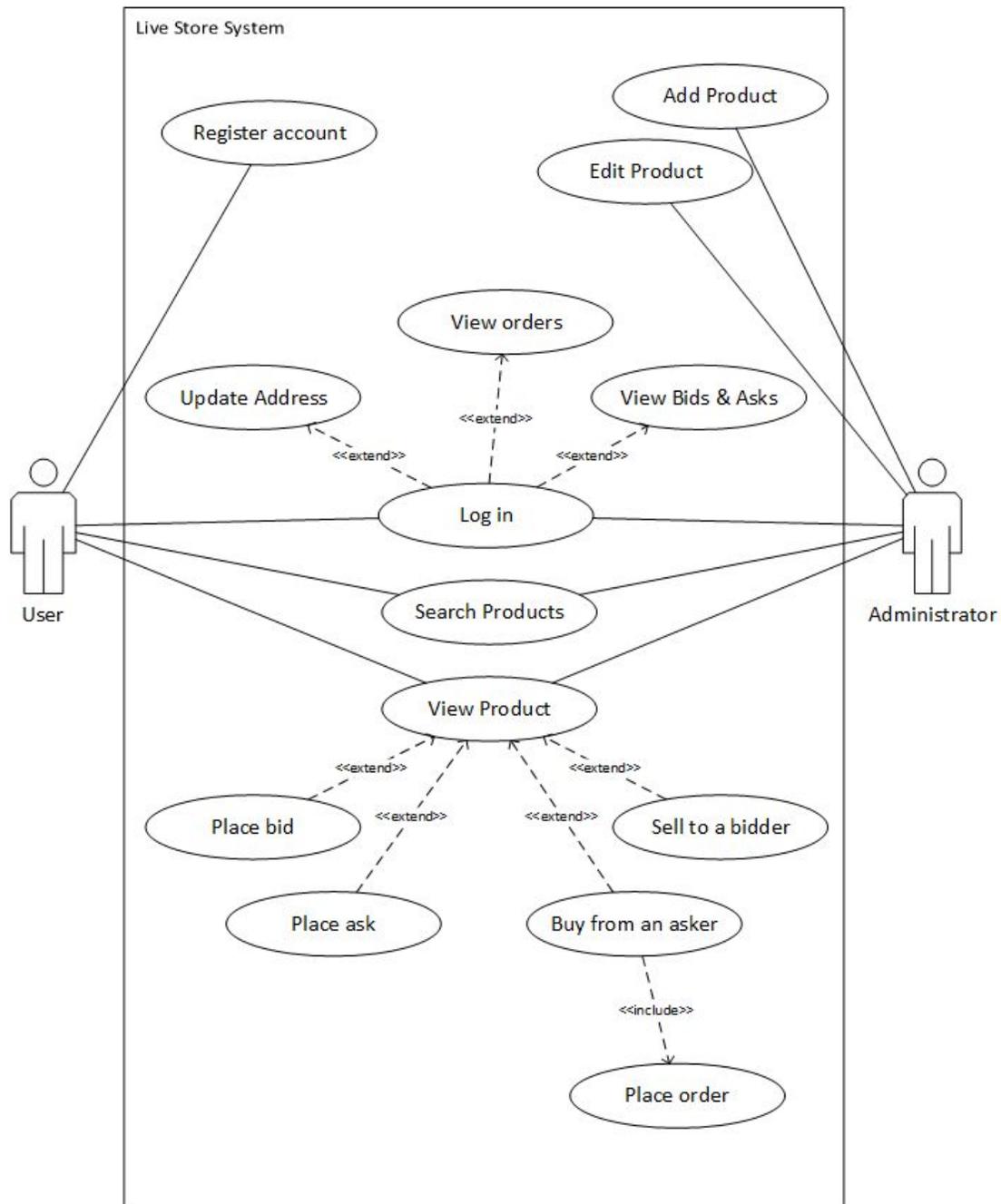


Figure 3.1: Use case diagram for the proposed system.

## Buy from an Asker

When an ask is accepted, there is no need to send a message through PubNub, as the ask is closed in the database so it cannot be fulfilled again.

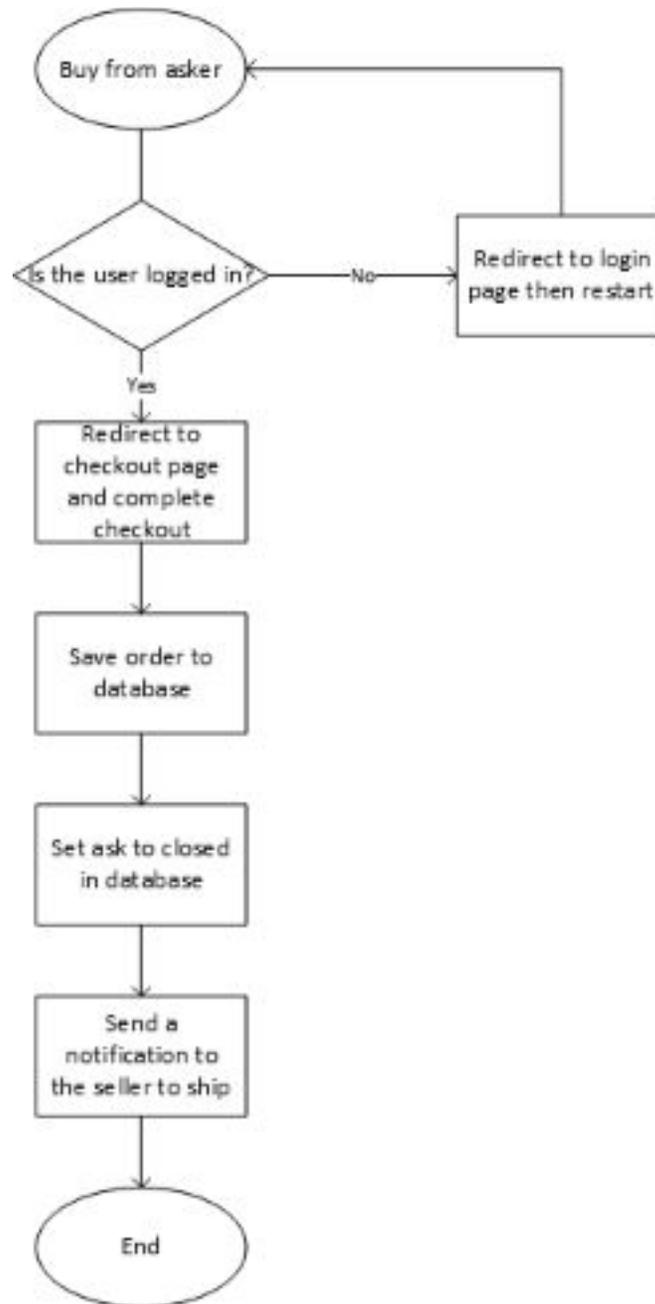


Figure 3.2: Flowchart for buying a product from another user's previously placed asking price.

## Sell to a Bidder

When an bid is accepted, the order is not immediately completed. The buyer is sent a notification to confirm their shipping address and payment details. The shipping address is passed onto the seller.



Figure 3.3: Flowchart for selling a product to another user for the user's previously placed bid.

## Place a New Bid

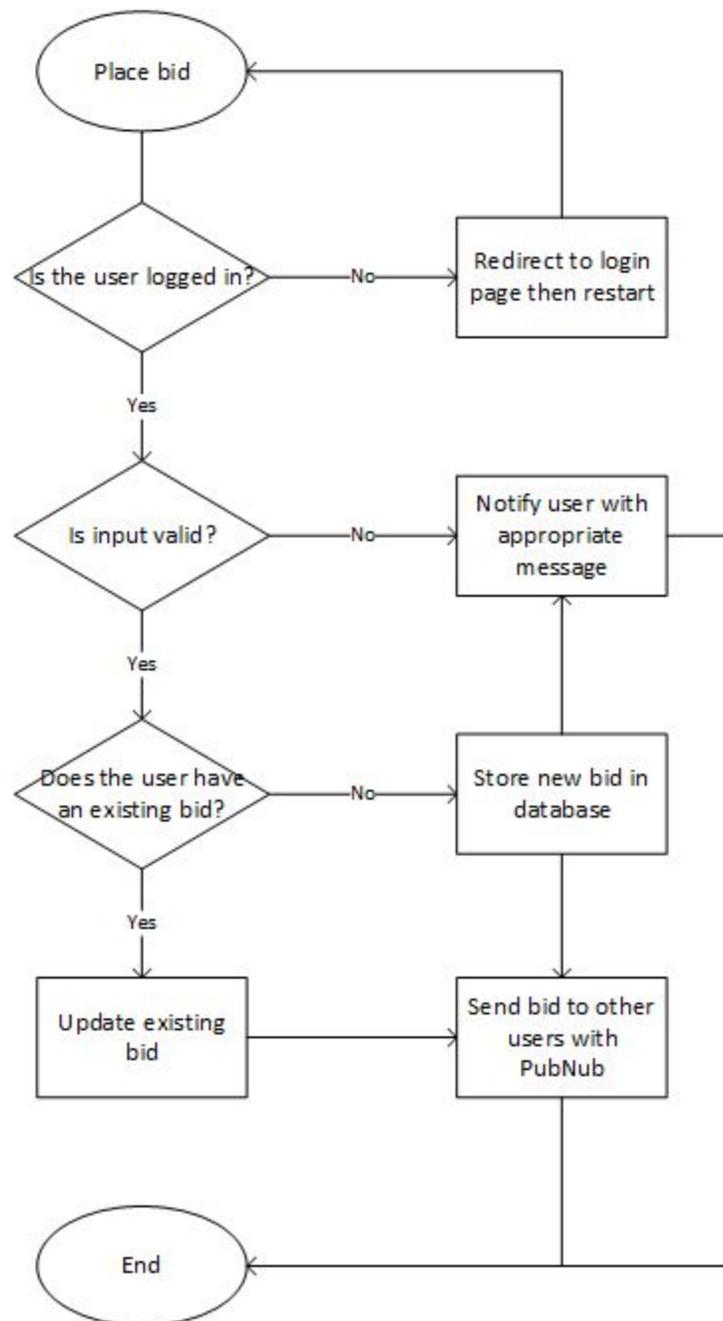


Figure 3.4: Flowchart for placing a bid on a product.

## Place a New Ask

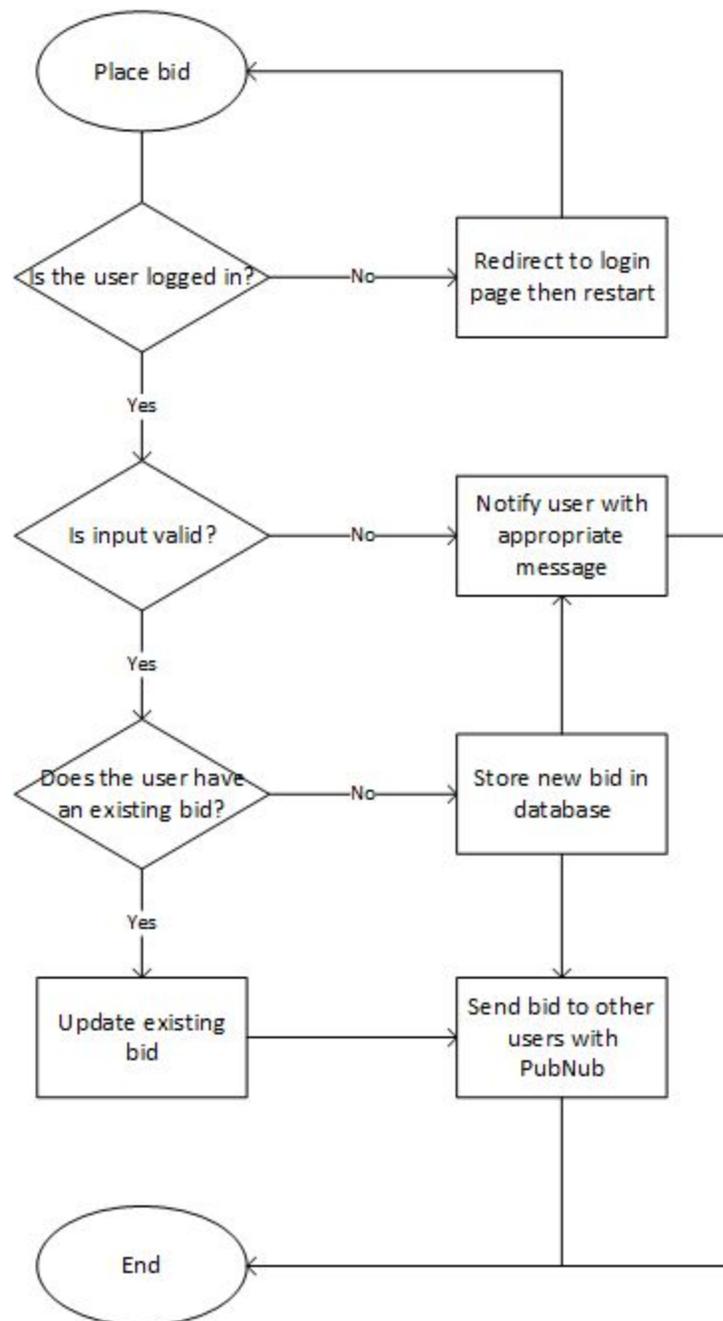


Figure 3.5: Flowchart for placing an asking price for a product.

### 3.2.2. User Interface Design

The user interface has been designed referring to the requirements specification given in section 3.2. The mockups have been created using a prototyping tool called JustInMind. Ben Shneiderman defined 8 golden rules for user interface design in the book *Designing the User Interface: Strategies for Effective Human-Computer Interaction* [16]:

1. Strive for consistency
2. Enable frequent users to use shortcuts
3. Offer informative feedback
4. Design dialogues to yield closure
5. Offer simple error handling
6. Permit easy reversal of actions
7. Support internal locus of control
8. Reduce short-term memory load

These guidelines applied in the creation of the following user interface designs. These designs were showed to my supervisor in a meeting for approval and additional feedback.

## Homepage

This is the first page that the user will be shown when they open the system. This means that all of the features of the system must be accessible from here. The designs all include a consistent navigation bar at the top of the page, which remains unchanged for the vast majority the views in the system.

The page provides the user with a prominent dialogue in the centre of the page explaining exactly what the system is designed to do, and includes a search function in a prime location in the page. Below the fold, the system displays a grid of products in the system. These grid elements contain concise and useful information for the product and link to the full product page:

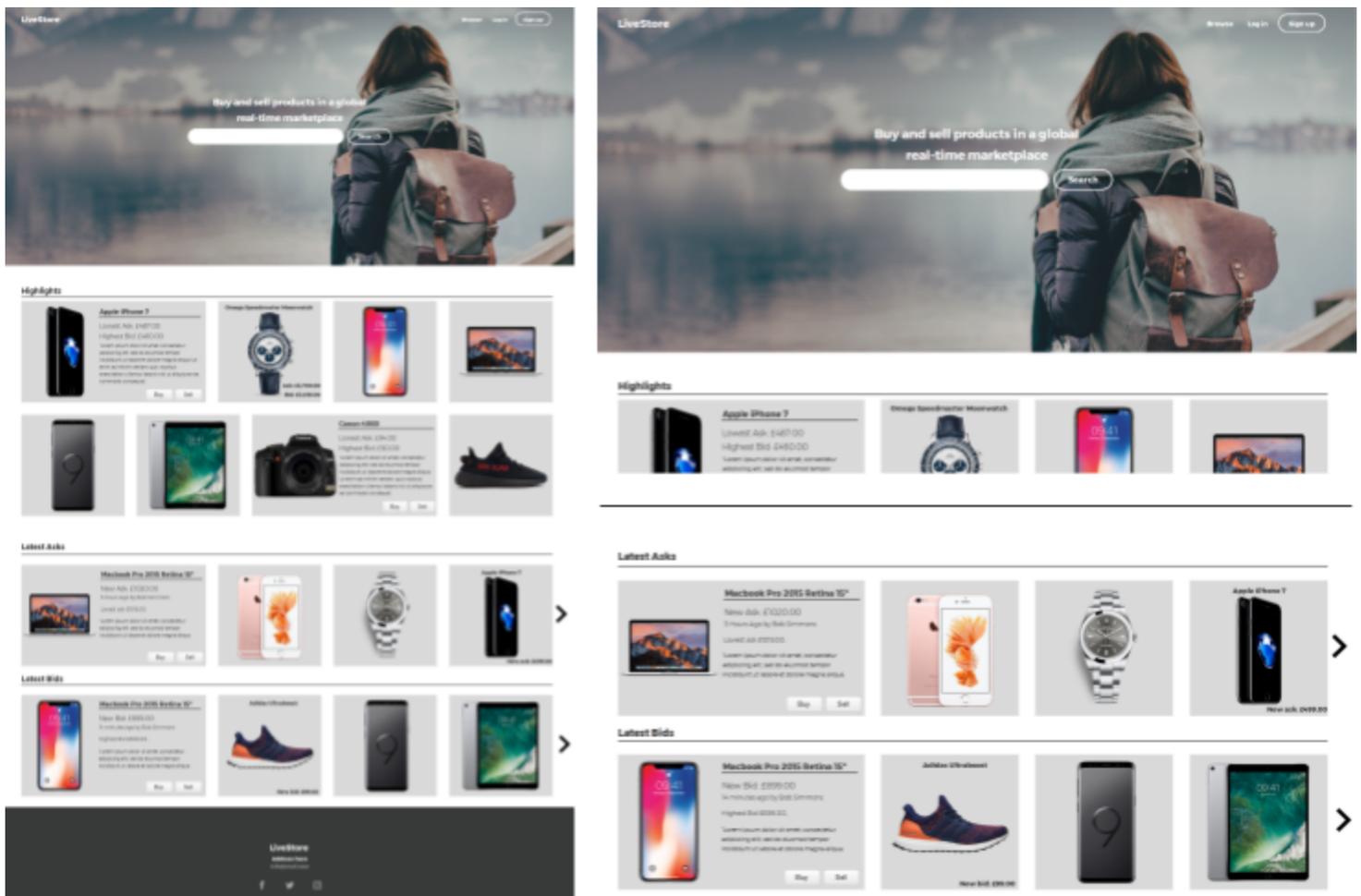


Figure 3.6: User interface design for the homepage. Left: full page, right-top: closer view of the navigation & search, right-bottom: closer view of the product grid.

### Product page

The product page maintains a consistent style and navigation bar as the homepage. The page is split into four sections: product information, buying options, selling options and historical market data. The market history originally was placed above the buy and sell options, however in the revised design it is placed below the fold to push the buy and sell options to a more prominent location. The page will give feedback for things such as placing new bids or if the user is the current highest bidder.

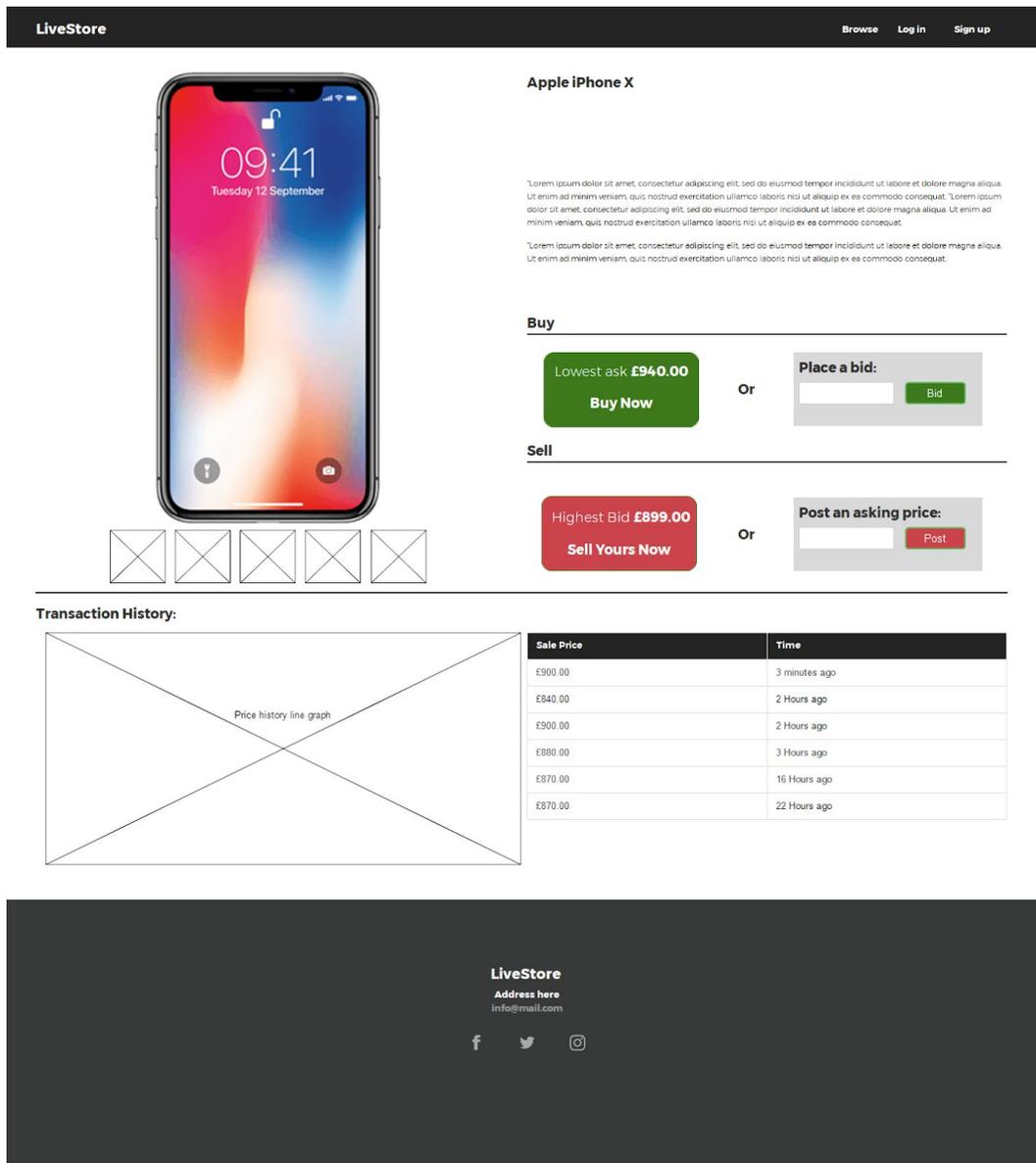


Figure 3.7: User interface design for the product page.

## Login

The login page contains a centred panel with with a simple username and password form.

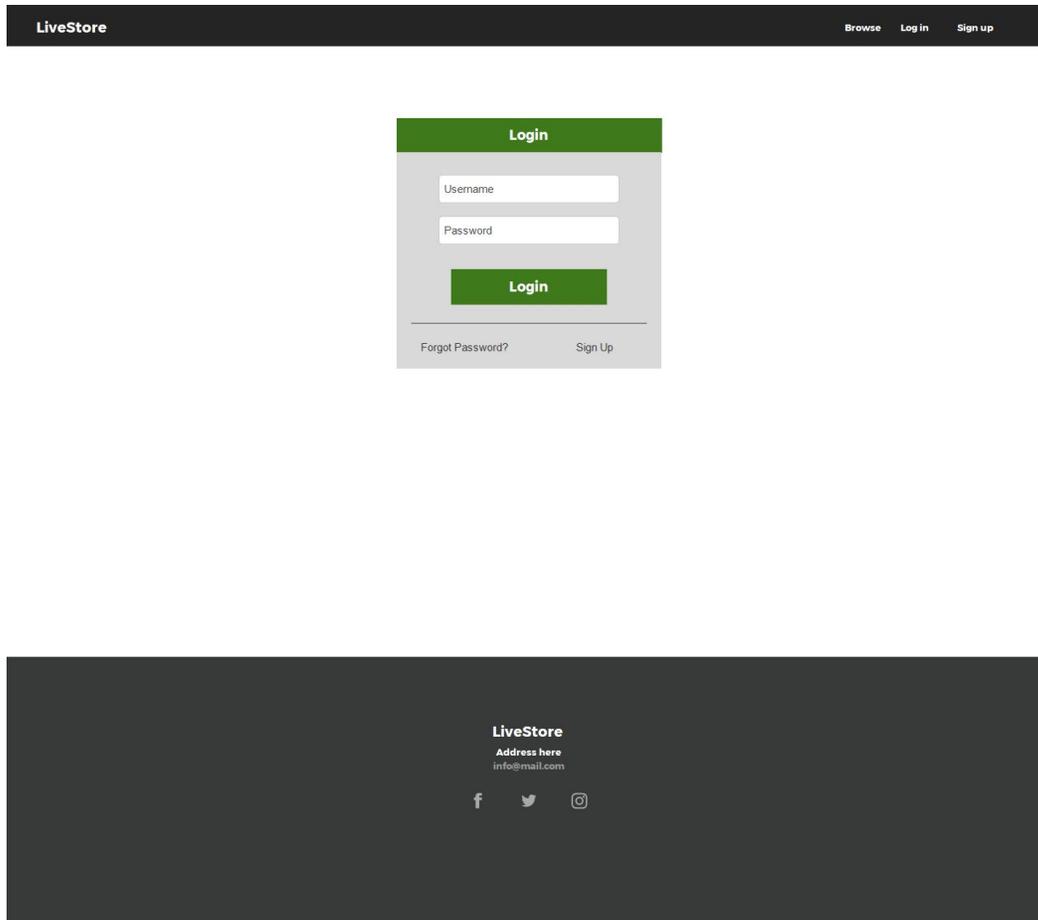


Figure 3.8: User interface design for the login form.

## Register

The register page maintains a consistent style to the login page however it has red colour coding as opposed to the login page green. This page is as simple as possible to make it easy to use, error messages are flashed at the top of the form.

The image displays the user interface for the account creation/register form. At the top, there is a dark navigation bar with the 'LiveStore' logo on the left and 'Browse', 'Log in', and 'Sign up' links on the right. The main content area features a 'Sign Up' form with a red header. The form includes four input fields: 'Username', 'Email', 'Password', and 'Confirm password'. Below these fields is a red 'Create Account' button. At the bottom of the page, there is a dark footer containing the 'LiveStore' logo, the address 'Address here', the email 'info@mail.com', and social media icons for Facebook, Twitter, and Instagram.

Figure 3.9: User interface design for the account creation/register form.

## Profile area

The profile area maintains the consistency of the navigation for the system. Additionally the profile area has a side navigation to separate the functionality to enable easy and clear navigation.

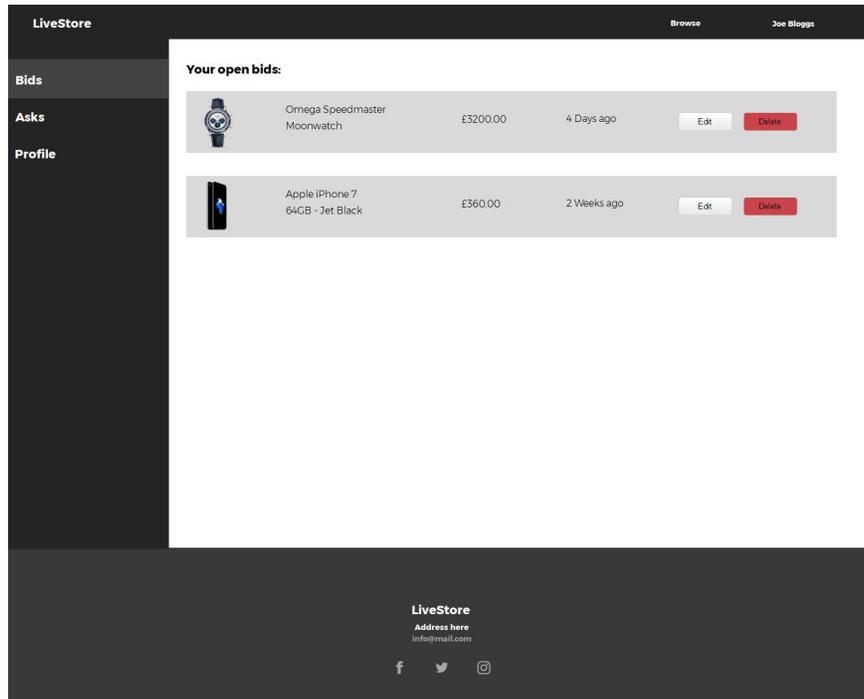


Figure 3.10: User interface design for managing your bids in a user profile.

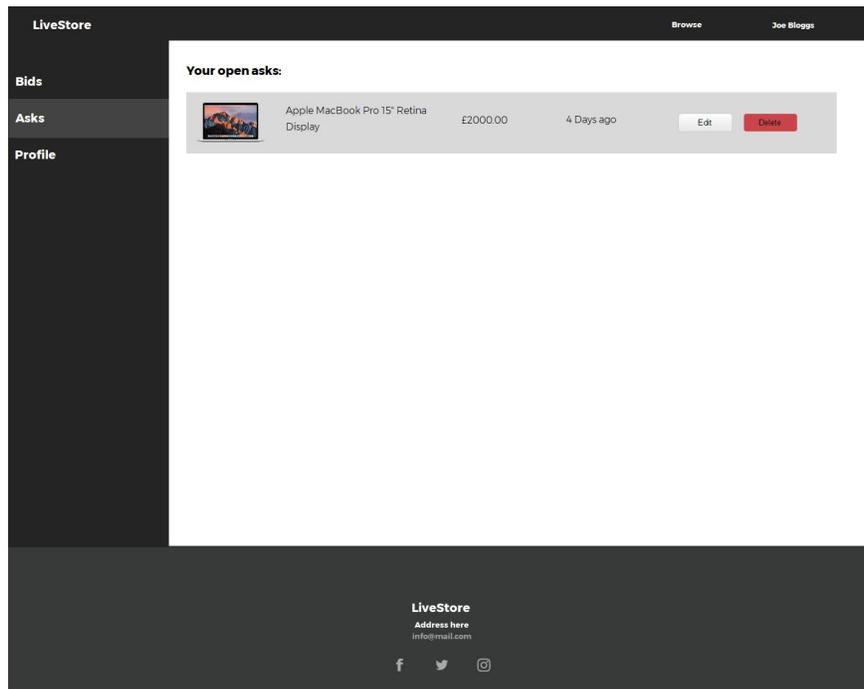


Figure 3.11: User interface design for managing your asks in a user profile.

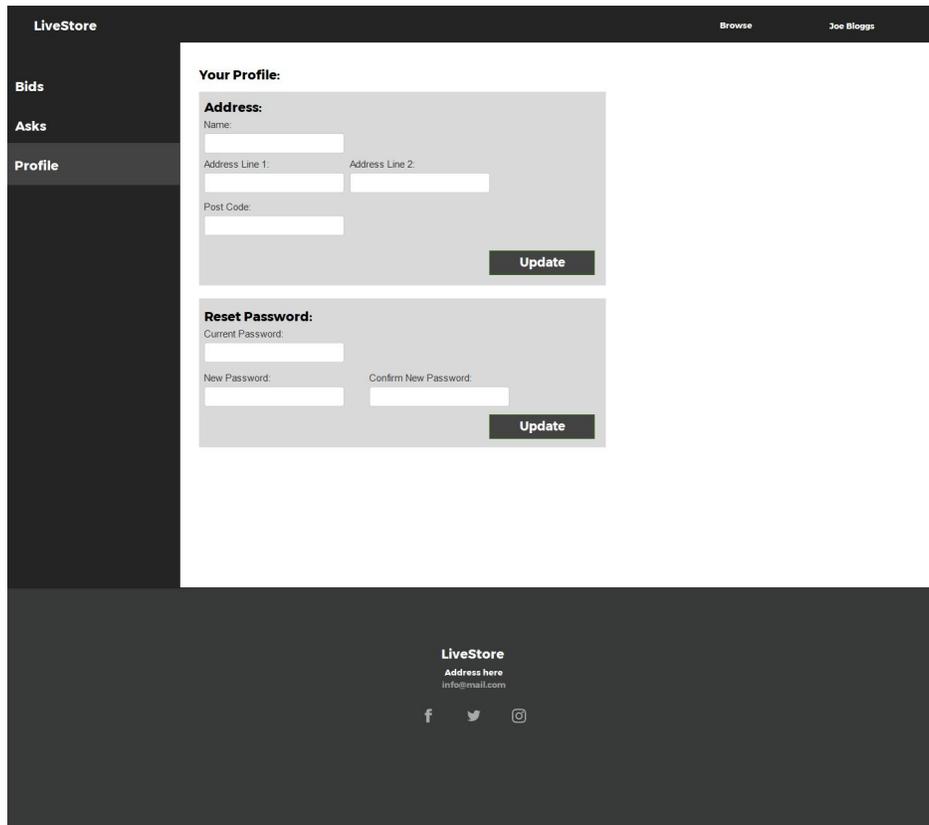


Figure 3.12: User interface design for the account settings page.

### 3.2.3. Application structure

The Model-View-Controller software architecture is the most common software architecture used for web applications [17]. The architecture separates the application logic into three separate parts: models, views and controllers.

- Models define all of the data that the application uses, for example they define the database structure and can sometimes handle input validation. Views get any data they need from the Models.
- Views control any user interface elements in the application, they can send user input to the appropriate controller.
- Controllers contain the bulk of the application logic. They often take user input and handle it as well as manipulate data from the models.

Figure 3.13 displays the way that the three components of the MVC architecture separate the application logic and interact with each other:

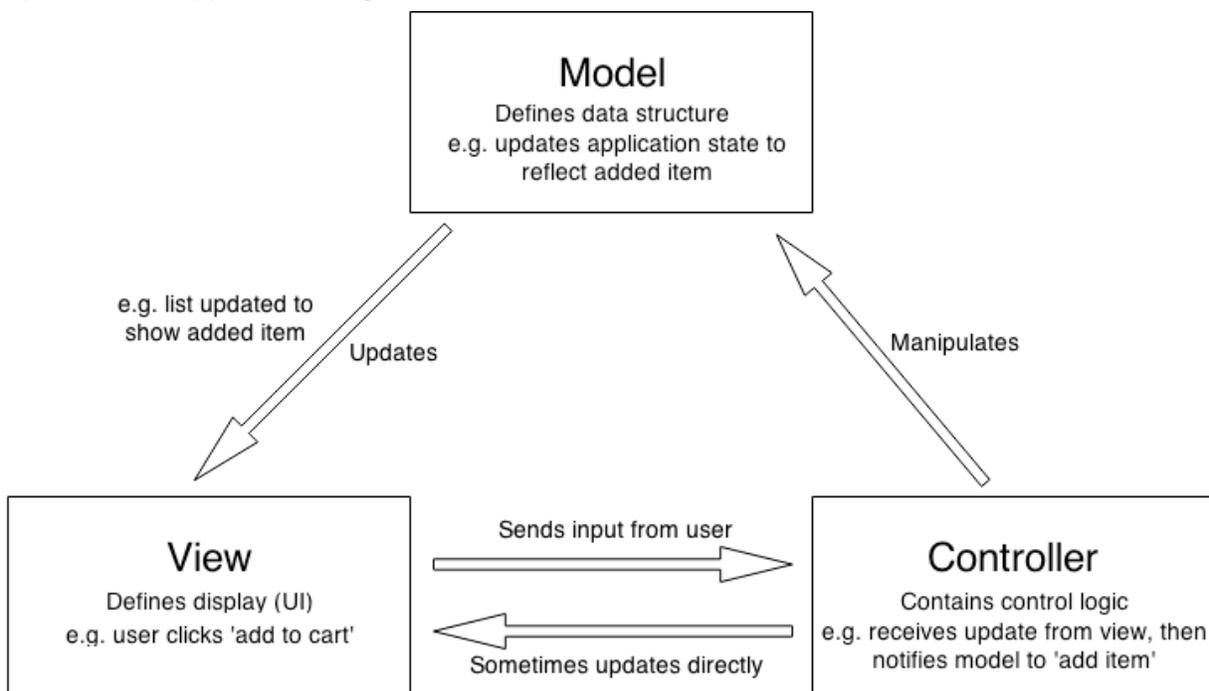


Figure 3.13: Diagram of the Model-View-Controller software design pattern. Courtesy of Mozilla Contributors [17]

The main advantage of this architecture is the way that it decouples all of the key application logic from the data handling and user interface, this makes the the application easier to develop, maintain and extend.

## The Final Application Structure

The final design for the application structure was to implement the Model-View-Controller architecture to make implementing the system as simple and as fast as possible. The MVC architecture enables maintaining states, which makes implementing authentication and authentication sessions for users much less complex. As an extension to the core application, in order to facilitate the ability to get data in real-time using AJAX with Javascript, I planned to create a small RESTful style API for the product data. Whilst my initial idea for the application structure was to create a fully RESTful application with a JavaScript frontend using a MEAN stack [18], for the final design I decided against this because of the JavaScript front-end requirement, which would introduce unnecessary complexity and reduce client compatibility.

The structure design utilizes the model-view-controller architecture to address the the second and third aims of the project defined in section 2.2:

- implement the system in a way that is maintainable, extensible and compatible
- create a secure, robust and scalable system

The MVC architecture is a suitable pattern to use to achieve this aims as it separates the program into single-responsibility modules that communicate with one another. This has a number of advantages: it gives the system inherent support for asynchronous operations, allows for parallel development which makes development faster and makes the code easier to comprehend and thus maintain and extend the program.

The product API functions as an entirely separate part of the rest of the system, however it queries the same database. It enables a client-side script to use AJAX to request for data in JSON format, it does not enable any creation, deletion or modification of product data. This means that there is no need to implement additional authentication middleware for the API which lets me keep things as simple and efficient as possible. The advantage of this approach is that it allows me to use a more traditional web application for the vast majority of the system which does not need any live data, making it easier to develop, maintain and extend the system.

As explained in section 2.6.2, the PubNub data stream network is used to push a newly placed bid, ask or order on a product to all of the clients currently viewing the product page. When a client opens a product page they are automatically subscribed to the PubNub channel for the given product. Each product has its own PubNub channel which it uses to these events to the subscribers and update the charts.

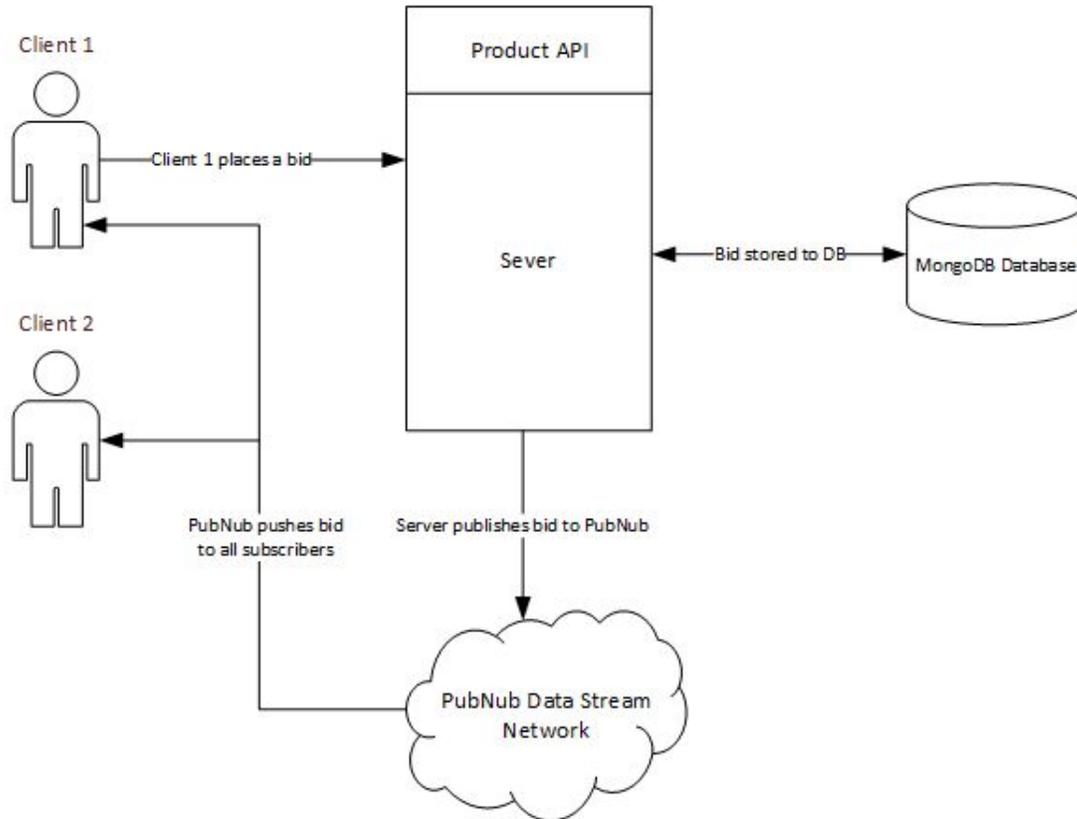


Figure 3.14: Diagram of the process of placing a bid and informing all users in real-time.

Figure 3.14 shows the process of placing a bid on a product. In this example, there are two users viewing the same product page Clients 1 & 2, and Client 1 places a bid on the product. The process is exactly the same for posting asking prices and consists of the following steps:

1. The first stage of the process is the client-side validation using JavaScript, before the bid is sent to the server, the client uses JavaScript to check if it is a valid input and whether the new bid will become the highest bid. If the input is valid then the bid is submitted to the server.
2. When the server receives the bid, it will validate the bid on the server-side for security purposes and then it will save the bid to the MongoDB database.
3. After saving to the database, the server then publishes the bid to the PubNub data stream network.
4. PubNub pushes the bid object to all of the clients viewing the product, which updates the product page
5. Finally, if the bid is the new highest bid, the client will send a new request to the product API to get the new bid so that the user can immediately sell to the bid.

The PubNub API uses the publish-subscribe pattern so it gives separate keys for subscribing to a channel and publishing messages to a channel. This ensures that the server is the only party who can publish to the network, so all of the messages published are valid messages from the server. Additionally, as PubNub uses the same JSON format for messages as MongoDB use, it eliminates the need to do any data processing to the bid, ask or order data before publishing to the network.

Implementing the MVC architecture in this system would produce a file structure for the application like so:

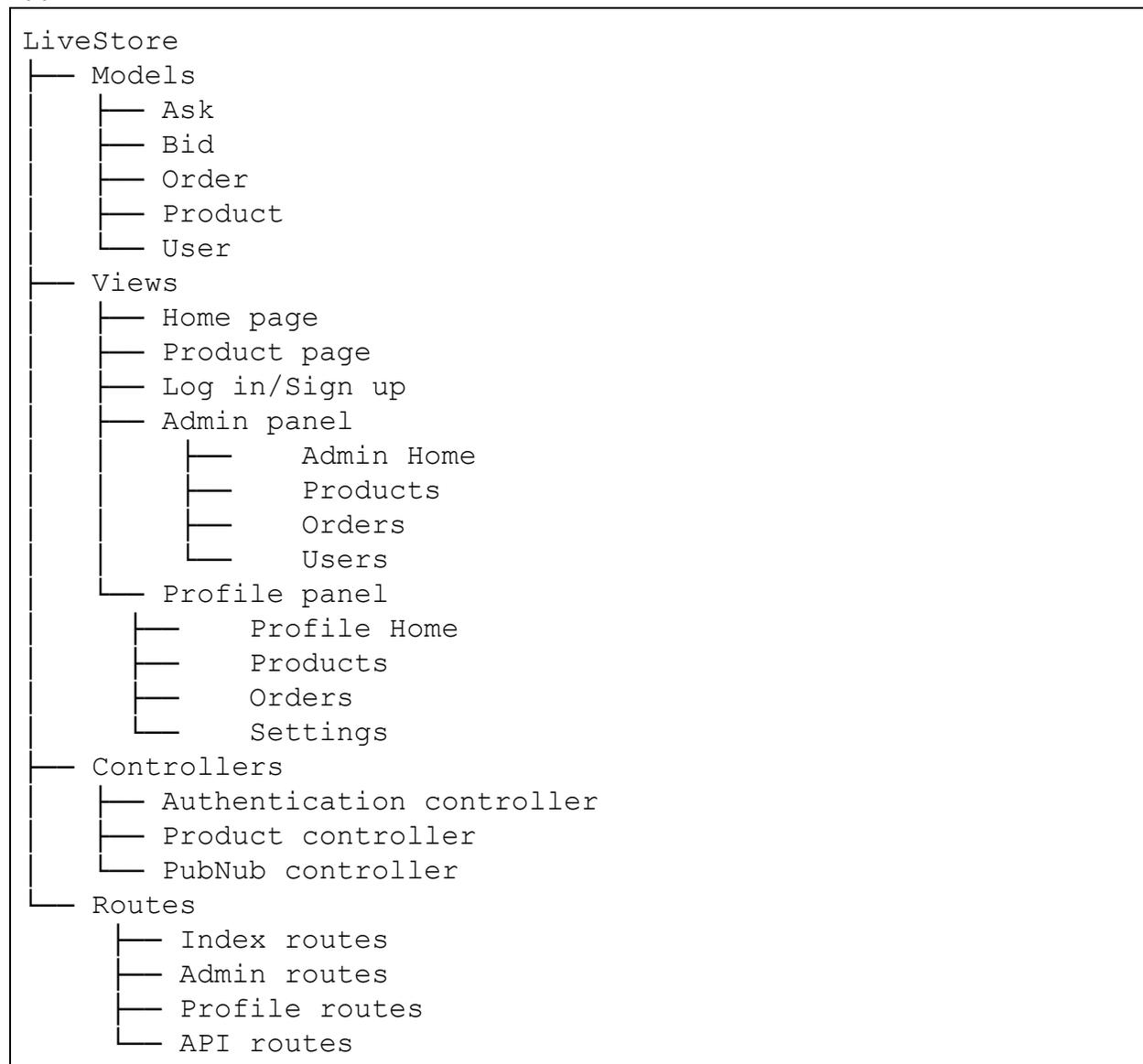


Figure 3.15: The proposed file structure of the application.

### 3.2.4. Database Design

I have chosen to use MongoDB for my database for the reasons stated in Section 2.6.3. MongoDB is a document oriented NoSQL database - which makes the design considerations different from traditional relational database systems such as MySQL or Oracle. As I have no previous experience with document-oriented databases I have referred to guidance given in the MongoDB documentation[19] and a white paper published by MongoDB entitled 'Performance Best Practices for MongoDB' that offers detailed guidelines for schema design that will perform well at scale[20]. I have included notable guidelines that I have referred to in my schema designs:

1. Store all data for a record in a single document
2. Avoid large documents
3. Avoid large arrays in indexed fields
4. Avoid long field names
5. Use caution when considering indexes on low cardinality fields
6. Eliminate unnecessary indexes

The schema has been optimised for performance over storage space, as storage is much cheaper and abundant than processing power, therefore the schema does, at times, contain duplicate data. The database design consists of 5 collections: Users, Products, Asks, Bids and Orders. Figure 3.16 shows the document schemas and the relations between each of the collections. It is important to note that MongoDB is not a relational database system therefore these relations are implied, they are not actually enforced by the database. They will be, however, enforced by Mongoose, the middleware that defines the schemas and relationships between the data, and what the system will use for querying the database.

The *Users* collection stores a document for each registered account on the system, this single document stores all of the data that is associated with any user. In order to optimise the queries when loading pages, the schema contains an array of reference IDs for all of the bids, asks and orders that the user has created, which allows the system to get the data using the Mongoose `.populate()` function without having to search through every bid, ask and order for the user ID.

The design also defines a schema for storing addresses, however addresses are not stored as a separate collection, they are stored as subdocuments to user and order documents. This allows the user to save their street to their profile for a quicker checkout process.

The *Products* collection stores all of the data associated with the product. Similarly to the *Users* collection, the schema includes all of the data necessary for the product page to load quickly using the `.populate()` function, making sure that a complex database query does not bottleneck the loading speed for the product pages. Additionally, for query efficiency, the database does not store the actual image files, as MongoDB enforces a 16 megabyte maximum document size and because this goes against the guidelines defined by MongoDB for efficiency. Instead, the database stores an array of file paths to the images, which are stored in a public directory on the server.

The *Asks*, *Bids* and *Orders* collections store all of the ask, bid and order data for each product. Whilst PubNub is used to communicate this data, PubNub does not enable as powerful querying functionality as MongoDB and does not, by default, store this data permanently, so it is necessary for a database to be implemented to store this data in a permanent and accessible way.

The database schema designs have given all of the collections and attributes concise, appropriate and consistent names and data types to help comprehension during the implementation.

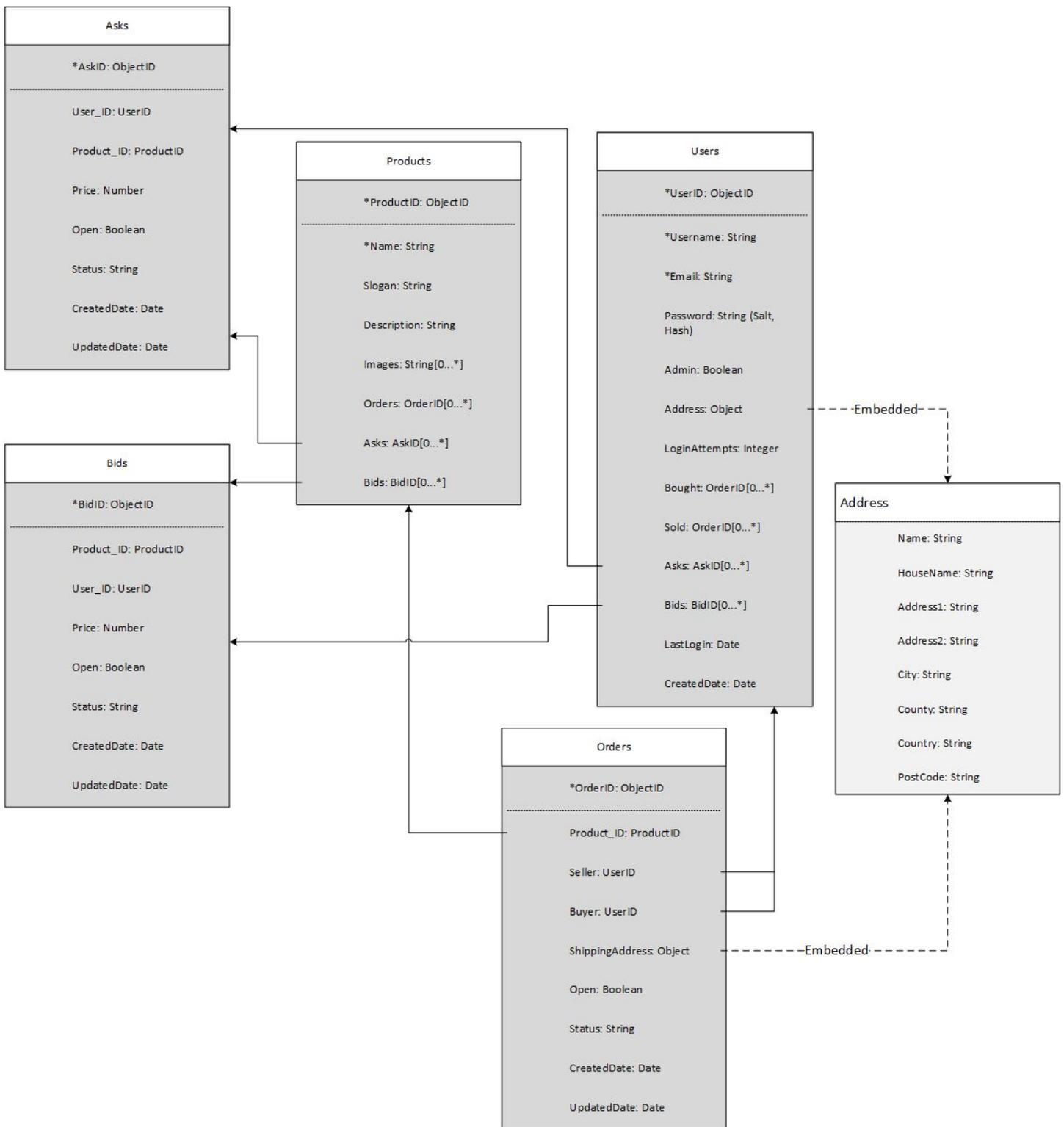


Figure 3.16: Overview of the database schema design. In the interests of simplicity, some relationships arrows have been omitted to increase readability.

### 3.3. Risk Analysis

It is vital to conduct a risk analysis before the implementation begins to minimise the potential for threats to impede progress on the project. Identifying potential risks enables them to be monitored and sensible contingency plans to be put in place.

	<b>Risk</b>	<b>Likelihood</b>	<b>Impact</b>
1	Data loss	Low	Large
2	Overly optimistic Scheduling/Estimation	Moderate	Moderate
3	Absence or Illness	Low	Large
4	Unclear specification	Low	Moderate
5	Changeability	Low	Moderate
6	Insufficient skill set	Moderate	Moderate

Figure 3.17: Table of the risks associated with the project.

	<b>Small Impact</b>	<b>Moderate Impact</b>	<b>Large Impact</b>
<b>Low Likelihood</b>		4: Unclear specification 5: Changeability	1: Data Loss 3: Absence/Illness
<b>Moderate Likelihood</b>		2: Optimistic schedule 6: Insufficient skill set	
<b>High Likelihood</b>			

Figure 3.18: A risk analysis map for the identified risks.

### 1. Data Loss

*Steps to prevent:*

Use the Computer Science School's GitLab system for version control and as a code backup. Additionally, use an off-site cloud storage solution to store periodic backups.

*Contingency:*

Restore from a previous backup or if not possible accept the risk.

### 2. Overly Optimistic Schedule

*Steps to prevent:*

Discuss progress updates with supervisor during regular meetings, Decompose sections into smaller, more attainable tasks.

*Contingency:*

Prioritise the functional requirements to product a minimum viable product. Focus on key aims for deliverables first and avoid feature-creep.

### 3. Absence or Illness

*Steps to prevent:*

Unpreventable, monitor absence and communicate with supervisor.

*Contingency:*

Re-organise tasks to deliver a minimum viable product that satisfies the functional requirements.

### 4. Unclear Specification

*Steps to prevent:*

Evaluate and adjust specification and design where necessary. Do not strictly adhere to the initial plan if a better option becomes available.

*Contingency:*

Communicate with supervisor frequently and adjust the specification to clarify any issues.

## 5. Changeability

### *Steps to prevent:*

Communicate requirements with supervisor effectively. Keep the design and implementation as simple as possible. Conduct regular code reviews and refactor accordingly.

### *Contingency:*

Dedicate time to implement the changes in functionality or adjust the specification to sacrifice other functionality.

## 6. Insufficient Skill set

### *Steps to prevent:*

Choose tools and methods that are familiar, if not then choose tools with good support and documentation. If required use resources from the internet or university library to learn how to use required tools and technologies.

### *Contingency:*

Adjust the aims and scope of the project accordingly and communicate concerns and changes to supervisor.

## 4. Implementation

*Note: The full code listings are provided in addition to the report. Instructions are included in Appendix A.*

This part of the report details the approach used to implement the program. Implementation was done entirely using *Sublime Text 3*, and for testing and debugging I used the following tools:

- *MongoDB Compass Community Edition* to view and query the database
- *Postman* to issue HTTP requests for testing
- *Morgan* NPM package to log HTTP requests
- *Google Chrome* and *Mozilla Firefox* developer tools

### 4.1 Authentication

*Passport* is an authentication package for NodeJS that abstracts the handling of account creation, authentication and sessions. It also enables multiple different types of authentication strategy from either a local database or from third party accounts such as Google or Facebook [21]. When handling local database accounts Passport handles the storing of passwords securely by salting and hashing them. As you can see in Figure 4.1, the `User.Register` Passport function takes the password as a separate argument after the user object to salt and hash before insertion:

```
/*
 * Register function, called when POST /register.
 * Creates a user account
 */
userController.register = function(req,res) {
  process.nextTick(function() { //to make sure the user does not exist first, process asynchronously
    //Find user
    User.findOne({ 'name' : req.body.name }, function(err, user) {
      if (err) return err;

      if (user) { /* make sure user doesnt exist */
        req.flash('error', "A user with that name already exists. Please choose another.");
        return res.redirect("/register");
      } else {
        //Username is free
        User.register(new User( {
          name: req.body.name,
          username : req.body.name,
          email: req.body.email
        }), req.body.password, function(err, user) {

          if (err) {
            console.log(err);
            res.render('register', { 'errorMessage' : err});
          } else {
            passport.authenticate('local')(req, res, function () {
              res.redirect('/');
            });
          }
        });
      }
    });

    req.flash('success', 'Successfully made your account. Please sign in:');
    res.redirect('/login');
  });
};
```

Figure 4.1: Code to register an account with the passport module.

It is important to ensure that the username provided is unique, so the database must be queried first to confirm this. The `process.nextTick()` function ensures that the query function is processed synchronously, breaking the event loop and making the program wait for the response to ensure that the block is complete to ensure there are no collisions [22]. Figure 4.1 shows that the process of creating a new user in the database is handled synchronously.

For protected routes, such as those to the user and admin areas, the *UserController* defines a simple middleware `isLoggedIn` function. This middleware is used for all protected routes in the system and redirects unauthenticated users to the login form and sends the referring page so that the user can be sent back to where they were before logging in. Figure 4.2 shows the middleware function in the *UserController* class (top) and an example of its usage in the routes for placing bids and asks (bottom).

```
/*
 * Middleware for accessing protected routes
 * ensures user is logged in.
 * if yes: next() function is called
 * if no: redirect to login page with referrer page
 */
userController.isLoggedIn = function(req, res, next) {
  if (req.user) {
    next();
  } else {
    req.back = req.originalUrl || '';
    res.redirect('/login?refer='+req.back);
  }
}
```

```
router.post('/product/:productid/bid', auth.isLoggedIn, bidController.newBid); //Place bid
router.post('/product/:productid/ask', auth.isLoggedIn, askController.newAsk); //Place ask
```

Figure 4.2: Authentication middleware to check if a user is logged in. Top: the middleware function to check if a user is logged in. Bottom: the middleware function is called in the route declaration.

The system also facilitates flash messages for the login and register pages to provide useful feedback for informing the user of events and errors as suggested by the user interface design (section 3.2.2). Figure 4.3 displays an example of this: the login page informing the user that the login credentials provided were incorrect.

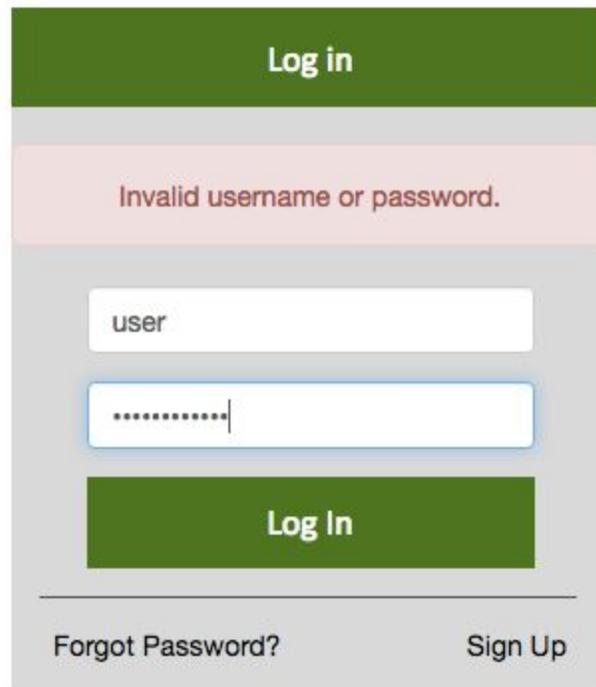


Figure 4.3: Flash message feedback example on the login page.

## 4.2. Placing a Bid or an Ask

For the same reasons as the authentication, placing bids and asks must be done carefully to ensure that no data integrity errors occur. When placing a bid this is done by first querying for an existing bid placed on the product by the user (figure 4.4). Only once this query is complete the place or update bid actions is completed. The method is identical for placing an ask.

To do this, the *BidController* gets the user document and finds all of the user's bids that are both open and match the product ID. If a matching bid exists then it will simply update the price and date for the existing bid. If a bid is not found then the system will create a new bid document and save it to the bids collection and the product and user documents. After the bid has been created or updated, the controller then passes the bid object to the *PubNubController* (section 4.3) which then publishes a message to PubNub. As stated earlier, the process is identical when placing asks.

```
//Check if user already has an open bid. if yes update that instead
User.findById(req.user.id).populate({ path: 'bids', model: Bid, match: {product: product_id, open: true }})
.exec(function(err, user) {
  if (user.bids.length > 0){
```

Figure 4.4: The query to check if a user has an existing open bid on the given product.

The product page gives four options for the user to buy or sell the product: place a bid, buy immediately from a current asking price, place an asking price or sell immediately to a current bidder (Figure 4.5). The user interface clearly displays the four options the user has and utilises colour and alignment to distinguish between buying (green) and selling (red) options, the purchase options are always at the top and the sell options at the bottom.

**Buy Ps4 Pro**  
Best immediate purchase option:  
Lowest Ask: £490  
**Buy Now**

**Bid on Ps4 Pro**  
The current highest bid is £400  
Placing the highest bid will increase your chances of a purchase  
Your bid.. **Place Bid**

**Sell your Ps4 Pro**  
Best immediate sell option:  
Highest Bid: £400  
**Sell Now**

**Place ask for Ps4 Pro**  
Your ask is still open!  
£490, placed on May 11th 2018, 12:30am  
Edit your ask:  
490 **Place Ask**

Figure 4.5: The buying and selling options presented to the user on the product page

The user interface displays informative colour coded flash messages in a prominent position on the page when events or errors occur. These messages include: successfully placed bid or ask, you are currently the highest bidder or lowest asker and form validation errors.



Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur?\*

You have successfully placed an ask for £1000

### Buy Macbook Pro 13" 2015

Best immediate purchase option:

### Bid on Macbook Pro 13" 2015

Figure 4.6: An example displaying position and styling of flash messages.

Additionally, if given the appropriate permissions, the system will use the browser's in-built push notification functionality where supported to send users who have the product page open a notification when a new lowest bid or a new highest ask is placed on the product (figure 4.7).

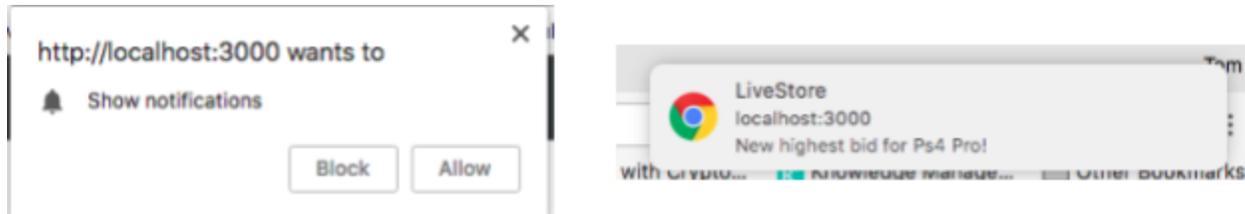


Figure 4.7: An example of the system asking for permissions (left) and displaying a push notification for a new highest bid that has just been made (right).

### 4.3. PubNub

The *PubNubController* class contains all of the code that initializes the PubNub channels and creates and sends messages to the appropriate PubNub channel for each product, it is also responsible for creating the Eon object for the charting engine to plot the data. This class is used once the bid, ask or order has been validated and saved to the database. The Schema for every message published is shown in figure 4.8.

Attribute	Comments
eon	This contains the data used for charting in the format required by the Eon charting system.
bid_id or ask_id	Stores the bid_id or ask_id so the system can be extended to enable PubNub to automatically place orders in the future.
price	The price value for the bid or ask.
time	The timestamp.
user	The ID of the user who placed the bid or ask.

Figure 4.8: The schema for messages published to PubNub.

The other responsibility of the *PubNubController* is to set up the channels for each product, including the storage of messages and the security configuration, the creation of messages and the publishing of messages.

## 4.4. Security & Validation

Security and validation is a primary concern for developing web applications, even more critical with ecommerce systems. While the constraints (section 2.3) do mention that the timeframe of the project does not allow for full security testing, during the implementation I have taken steps to create a secure application. Section 4.2.1 describes the authentication subsystem for the project in more detail, however the while the authentication of users is the primary concern in creating a secure application, there are many other security concerns to address.

Using existing frameworks and modules for securing applications is typically best practice, as creating custom security measures for your application can take an enormous amount of time and be extremely dangerous. Many popular frameworks, including Express & NodeJS, have proven security track records with security researchers and open source contributors with far more experience and knowledge always working to keep the framework secure.

Securing the data in transit is a vital security measure for web applications, at present the system does not use HTTPS as it has not been deployed, however if and when it is deployed this is a necessary step. An effective way of doing this would be to use an NGINX web server to handle HTTPS traffic and run the NodeJS application on top of it. A security feature currently implemented in the system is the creation of a Content Security Policy (CSP) to detect and mitigate attacks. The *helmet* package in NPM abstracts much of this process, and can help against attacks such as cross site scripting.

The system can never trust the integrity of any request coming from a client so it is vital that data be cleaned and validated server-side before it is processed. Throughout the system, user input is minimised to the essential, any data that can be retrieved from elsewhere is done so. Where user input is essential, the input is validated with positive validation, as an additional step, *mongoose* validates the data being inserted to the database. Figure 4.9 displays one example of pre-save validation carried out before updating the database. Additionally, the *BidController* and *AskController* classes perform some basic validation on the bid and ask forms to check if the input is a valid number and it is greater than 1.

```
ProductSchema.pre('save', function(next){
  //Validate name is unique
  var self = this;

  Product.findOne({name : this.name}, 'name', function(err, result) {
    if (err){
      next(err)
    } else if (result) {
      self.invalidate("name", "Product name must be unique!");
    } else {
      next(); //name is unique so continue
    }
  });
});
```

Figure 4.9: Product schema pre-save validation function for inserting new documents into the Products collection.

## 4.5. Placing Orders

As explained in section 4.2, there are four options presented to the user on the product page (figure 4.5). Two of these options enable a user to immediately place an order to either purchase or sell the product, these options are only available if there is currently an open ask or an open bid respectively. The orders are placed by either purchasing the product for the lowest current asking price, or selling the product to the highest current bidder.

To purchase an item from the lowest asker, the user simply needs to click on the buy now button. This button takes the user to a checkout page that contains the order details and an address input form, to increase the speed of the checkout process the address form will be automatically populated if the user has an address saved to their profile. The limitations of the project (section 2.3) mean that the system does not take payment information at this stage. Once the user confirms the order the seller will receive a notification in their profile area (figure 4.10).

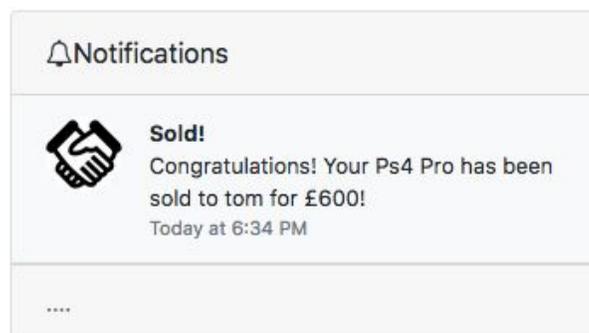


Figure 4.10: Notification sent to a seller when their asking price is accepted by a buyer.

To sell an item immediately, the process is exactly the same but in reverse. The seller who accepts a bid will be taken to a confirmation page displaying the details of the order. The buyer/bidder will be notified in their profile area in the same way.

The *OrderController* class is singly responsible for all of the functionality for placing orders. It separates orders into two classes: buy and sell, because the process is very slightly different for each. The class places each order synchronously to ensure that no collisions occur. The first step of placing the order is always to set the associated bid or ask to closed to ensure there can only ever be a one-to-one bid/ask to order relationship. The class then does the following:

- Create the order document and save to the *Orders* collection.
- Update the orders array in the product document.
- Update the bought-orders array in the buyer's document. Adds a notification to the users profile.
- Update the sold-orders array in the seller's document. Adds a notification to the users profile.
- Finally, it redirects to an order success page.

## 4.6. Problems Encountered

The way that NodeJS queries the database asynchronously means that if improperly utilised the program can have null pointer errors. This means careful consideration of the order of operations is required to ensure robustness and data integrity. This problem can be exacerbated by MongoDB's lack of transaction management. Some features of the system had to be implemented synchronously, notably inserting critical data into the database, such as user account creation (Section 4.1) and placing orders (Section 4.5). This problem was overcome by using synchronous queries where needed to ensure data integrity.

Implementing the user interface for the system, particularly the product page, presented some challenges, particularly when trying to facilitate the required real-time data communication features in a robust way. Due to my lack of experience with JavaScript on the client, particularly with AJAX, this section of the implementation took much longer than planned. To speed up the development process I used the jQuery framework to handle the AJAX requests. Additionally, for the styling I relied on Bootstrap to make the website design responsive and usable on mobile.

## 5. Results & Evaluation

### 5.1. Changes to the Initial Plan

#### 5.1.1. Application Structure

The initial application structure design specified just three controller classes: *ProductController*, *UserController* and *PubNubController*, however over time these three controllers grew unnecessarily large and as a result I refactored the code by separating them into the following 7 controller classes:

- *AdminController* - contains middleware that checks if the user is an admin.
- *UserController* - contains all of the methods for registration, logging in, logging out and middleware to check if the user is logged in.
- *ProductController* - contains the code to create, read, update and delete products.
- *AskController* - handles all validation for placing asks, stores the ask in the database and sends data to the *PubNubController* class.
- *BidController* - handles all validation for placing bids, stores the bid in the database and sends data to the *PubNubController* class.
- *OrderController* - creates order documents and closes the associated bid or ask.
- *PubNubController* - contains the code to create and publish the new bid, ask and order messages to the appropriate PubNub channel.

This new structure makes the application much easier to develop, maintain and extend, addressing the third aim of the project (section 2.2). Each controller class now better follows the single responsibility principle of object oriented design [23] [24], as each one now handles only one specific area of the application logic.

#### 5.1.2. PubNub Configuration

The initial plan for using the PubNub network for live communication was to have just two channels: *bids* and *asks*, through which the appropriate messages for all products would be sent. During the implementation this plan changed for two reasons: to reduce the amount of messages sent and to easier facilitate the graphing of the data. The revised plan was to have two channels for each product in the system. PubNub does not have a limit for the amount of channels, however it does charge more based on the amount of messages that are sent, so the new method of implementation actually increases efficiency massively by reducing the amount of messages broadcast.

The original plan would have distributed every single bid and ask message to every client currently connected to the system. The vast majority of these messages would have been ignored by the clients as the message only pertains to users on the specific product page. Additionally, separating the messages into product-specific channels allows easier implementation of real-time graphs with the PubNub Eon charting engine. This new usage configuration for PubNub increases efficiency, performance and scalability which directly addresses part of the second specified aim of the project (section 2.2) for the system to be robust, performant and scalable.

## 5.2. Testing Functional Requirements

### 5.2.1. Exploratory Testing

The first stage of testing the implementation was to conduct exploratory testing. While this was done informally throughout the development process, it is still vital to do this after development is complete to identify any bugs and edge cases. This testing some critical bugs that were address before continuing onto the test cases.

The most frequent bugs identified were those associated with input validation. These appeared in the place bid and ask functionalities as well as the add product functionality in the admin panel. These bugs were addressed before completing the test cases.

Another bug that was identified through exploratory testing was in the bid and ask update code on the client side. Due to the asynchronous nature of AJAX, if the request was slow, variables could occasionally be undefined if the AJAX request was taking too long. This was fixed by creating a function to update the page and calling that function in the AJAX callback so the page waits for the data first.

### 5.2.2. Test Cases

To evaluate the completed system I created a number of test cases and included them in full in Appendix B. The test cases were devised to test each of functional requirements (section 3.1) to objectively evaluate if the implementation was a success. The results have been tabulated and summarised in figure 5.1:

ID	Title/Purpose	Pass/Fail	Comments
1	User creation	Pass	
2	User authentication	Pass	
3	Admin authentication	Pass	
4	Adding a product	Pass	
5	Viewing the product page & pricing data	Pass	
6	Placing a bid	Pass	
7	Placing an ask	Pass	
8	Sell to the highest bid	Pass	
9	Buy for the lowest ask	Pass	
10	View bids and asks in user profile	Pass	
11	View orders in user profile	Pass	
12	Update a product	Fail	Time constraints forced me to de-prioritize this requirement
13	Search products	Pass	Simple regular expression search.
14	Notifications in profile for orders	Pass	
15	Push notifications for highest bid and lowest ask	Pass	Uses browser push notification features to achieve this.
16	Product page updates in real-time	Pass	

*Figure 5.1: Summary of the results of the use case testing carried out. The full tests are included in Appendix B.*

As can be seen from figure 5.1 the vast majority of test cases carried out passed - all of them apart from one. Additionally, every *Must* and *Should* requirement specified in section 3.1 was successfully fulfilled. I have attributed this success to the organisation

of the requirements in the planned chronological order of implementation. This prioritisation mitigated the risks of scope creep and gold plating (section 3.3).

The only test case that failed was test case 12. This test case pertained to requirement 3.5: *Update Product Details*. While the failure of this requirement is disappointing, the requirement is a *Could* requirement and the very last requirement in the list. This failure has been attributed to time constraints in the development of the project due to the problems encountered implementing the user interface (Section 4.6).

## 5.3. Testing Non-Functional Requirements

### 5.3.1. Usability

One of the constraints of the project (Section 2.3) was the inability to do a proper usability evaluation of the completed system. Unfortunately this means that the usability testing is outside of the scope of the project. Nevertheless, the user interface designs were guided by and have made considerations for human computer interaction principles defined by *Designing the User Interface: Strategies for Effective Human-Computer Interaction* [16] (Section 3.2.2). The interface does fulfil the second requirement for usability, as it is consistent across the whole system and does give concise and informative feedback to the user.

### 5.3.2. Performance & Scalability

#### Performance

The *Morgan* debugging package used for development logs the time that the system takes to respond to a request and the HTTP response code it returns. This can give a rough estimate of the efficiency of database queries and the application logic, this can identify any outliers to be addressed. Additionally, the requirements specification for performance (Section 3.1.5.) state that the system should load every page within 5 seconds. To test this requirement, I used the *AppTelemetry* [25] plugin for Firefox to time the loading of pages whilst testing the application and had an average page load time of 1.59 seconds and a maximum load time of 3.12 seconds over 47 page loads.

The other requirement specified for performance is for new bids and asks to be pushed to the other clients viewing the product page within one second, without having to refresh the page. While this is difficult to test objectively, PubNub guarantees delivery of messages in 250ms and the product page does indeed update the values without refreshing the page.

## Scalability

Every effort has been made in the design and implementation of the system to make sure it will scale well. This is a key non-functional requirement for the system (Section 3.1.5), because if the project does not scale it cannot be used as a base to be extended into a deployable system. Unfortunately the scope and timeframe of the project does not give me the resources to test this fully, however the following features and design decisions demonstrate an awareness of the requirement for horizontal scalability.

The system uses the MongoDB database management system which has a proven track record of being highly performant at massive scales, using its sharding technology to distribute the database horizontally across clusters of servers. I designed the database schema following MongoDB's documentation for schema design best practices (section 3.2.4).

The key factor in the the decision to use the PubNub data stream network as opposed to building my own system for real-time communication was because of the infrastructure that PubNub has in place already. They can provide a massive amount of infrastructure that can be scaled alongside the application. This robustness and scalability would take a huge amount of time and resources to implement yourself.

The choice of NodeJS itself was also dictated by its lightweight design and proven performance at scale (section 2.6.1), its asynchronous design for I/O operations, such as database queries, allow the program to continue running while waiting for a response meaning that no processing resources are wasted.

### 5.3.3. Security

As discussed in greater detail in section 4.4, the system uses *PassportJS* for handling all of the user account creation and authentication. Passport salts and hashes the password using Bcrypt when the account is created so the password can never be recovered, which is the standard for password storage. A simple middleware function ensures that protected actions such as placing bids and asks require an authenticated user. This middleware function will redirect to a login page if the user is not logged in.

While not specified in the non-functional security requirements (section 3.1.5) and the constraints (section 2.3), the system takes additional measures to the secure the system. These measures are discussed in section 4.4. The implementation of the system successfully fulfils the security requirements specified in section 3.1.5.

#### 5.3.4. Portability

The system has been tested in and is fully functional in Mobile Safari and Chrome as well as desktop Chrome, Firefox and Microsoft Edge. The system uses only ubiquitous and widely supported features within JavaScript engines in all popular, modern browsers. It also uses the Bootstrap CSS framework to create a responsive design that adapts to the screen size on the device. This means that the implementation of the system successfully fulfils the portability requirements specified in section 3.1.5.

## 6. Future work

Due to the time constraints of the project, there are a number of features that I would like to implement in the future. The system has been designed and implemented to make it easily extensible and maintainable. This section of the document details some additional features that could be implemented on top of the current system. Some of the features have been identified in the project constraints however there are some ideas that presented themselves during the implementation of the project.

### 6.1. Payment processing

One of the key limitations identified at the beginning of the report (section 2.3) was the inability to provide a method for paying users when purchasing items. Given more time, I would have liked to complete the system by implementing a method to send payments by creating a subsystem from scratch or by using a third-party payment processor such as Braintree or Stripe, this option would reduce the liability with respect to storing payment data.

### 6.2. Product Categories and Variants

The current implementation does not support some of the product catalogue features that are included in existing ecommerce solutions such as osCommerce (section 2.5.1) and PrestaShop (section 2.5.2). In the future I would like to implement categories for products and product variants such as colours or sizes.

The product categories feature could be implemented as an extension by simply adding a new field to the product document to store category information for each product. This feature would be simple to implement, product categories could be defined by the administrator and have various metadata associated with them.

Product variants would be slightly less trivial to implement in the future, the simplest way of accomplishing this feature would be to store each variant of a product as a separate product document, and then reference to each variant in an array field. This would mean that the core functionality for placing asks, bids and orders would not be to updated to support the new feature.

### 6.3. Using the Data

If deployed, the system could potentially be collecting and storing vast amounts of detailed, temporal data on the value of a given product. If the dataset grows large enough, mining this data could create some valuable information on the state of the marketplace for any item, which could be used to provide a range of tools and services that the system could provide to the user.

The system has an advantage over platforms such as eBay due to the consistency of the pricing data. To harvest data from eBay you would need to account for all of the possibilities for different listings, which could make the dataset noisy and incomplete.

The data could be used to accurately determine the value of every product in the system, the more data it collects the more accurate it becomes. The value of items can be determined by combining a number of metrics including but not limited to the bid-ask spread, order volume over time and price volatility.

This data could be used to create a detailed market model for a product that could analyse trends and estimate how much the product could be worth at any time. Additionally, the model could then be applied to predicting the prices of similar products in the marketplace with machine-learning technologies. For example: the system could use the model for previous iPhones to predict the best time window to purchase the product for the best deal. This information could be made available to the user on the product page.

### 6.3. Portfolios

As the system currently stores data on all of the products that users have placed an asking price for and the products that they have purchased, it is trivial to generate, for each user, a portfolio of the products that they own. The system could then calculate the value of each product and the portfolio as a whole. Additionally, given the original retail price of products or the price the user purchased a product at, the system could calculate metrics such as return on investment for the portfolio. This functionality would open up other options for interacting with the marketplace. As it is essentially the same as the stock market, users could potentially invest in items by buying and selling futures or shorts.

## 6.4. Monetisation Strategies

Section 2.4 provides a number of potential ideas for the business case for the system. I have identified three possible methods to monetise the deployment of the system: Commission, Membership and charging for placing bids and asks. Implementing a commission fee for completed orders would likely be the best course of action to minimise the up-front investment. This could be done simply in the order code by calculating a percentage or flat fee and adding it to the total. Requiring users to make a payment before being able to use the system could turn people away.

## 6.5. Usability Testing

The initial plan document stated that an aim for the project was to design and create a user interface for the system that would be evaluated by usability testing. During the implementation of the project the aims were adjusted away from user interface design and usability testing due to the time constraints (Section 2.3.2). Given more time, a key requirement for the development of the system would be to design a user interface using HCI and UX principles and to carry out a full usability assessment of the system.

## 7. Conclusion

Throughout this document, I believe that I have demonstrated that the project has been completed successfully. Section 2.2 defined 5 core aims derived from the motivation for the system, all of which I believe have been proven to be satisfied. The system provided is a good first step to a system that could be deployed in the future.

The evidence for achieving the first aim - to create an application that enables a user to purchase, sell, bid or ask on a product - is the system's completion of the test cases. All of the test cases apart from one passed (section 5.2.2) and the system fulfills the vast majority of the requirements specified (section 3.1), including all of the *must* requirements. The only requirements that have not been fulfilled are some of the *could* and *should* requirements, additionally, the third aim - the extensibility and maintainability of the system - makes it possible to easily complete these requirements in the future. Additionally, I believe I have shown through the testing carried out that the application satisfies the fourth aim of the project - to facilitate real-time communication of bids and asks in real-time.

Throughout the design of the system, an awareness of the need for the security, robustness and scalability has been demonstrated. While unfortunately I lack the time and resources to test these attributes, I believe the decisions made during the design and implementation of the system verify the achievement of the second aim - to create a secure, robust and scalable system, without compromising usability or performance.

The fifth and final aim of the project was to design and implement a simple and usable user interface for the system. While the project scope did not include full usability testing and analysis the interface was designed with the consideration of good HCI principles and the designs provided were approved by the project supervisor.

There are some issues with the system that have been identified, at it is by no means a product ready for deployment, however the completed work is a good base that can be extended and improved upon in the future. Additionally, I believe I can take many of the teachings of the project into any other projects I undertake in the future.

## 8. Reflection on Learning

In this section I will identify and critically evaluate some areas of success and some areas of improvements around the project. Using the double-loop learning methodology I have identified the areas of the approach that were successful and could be applied in the future and the areas that were not successful, where the approach could be altered.

At the beginning of the project, I had no previous knowledge or experience with many of the tools and technologies used: NodeJS, MongoDB, PubNub etcetera. The completion of this project allowed me to learn about these technologies and provided an opportunity to apply the new knowledge I had learned to a project with a purpose. This has enabled me to develop technical skills and confidence that I can transfer to new challenges in the future.

Whilst the project overall was a success, with hindsight I would change some of my actions during the initial stages of the project. I spent a large amount of time trying to decide what languages and frameworks I would use for the implementation, and ended up returning to my original idea. While this did allow me to make an informed decision in the end, I spent far too much time researching and learning multiple languages and frameworks that would have been better spent actually implementing the system. It is important not to get bogged down in deciding on the absolute best approach to a project if it eats into the time to actually undertake it.

Again, with hindsight, I can see that I did not manage the time afforded to me efficiently. I feel I made a slow start to the project, in part due to the reason stated previously and my approach to development was ill-advised. My approach was to complete the development of the system before beginning the report. While I feel I did manage development time effectively by splitting the process into subtasks and giving myself milestones, If I were to do this project again, I would start the development much earlier and at the same time include weekly subtasks pertaining to the report writing in addition to development.

During development I found that it was useful to draw flowcharts detailing the functionality for a subsystem. I did this for many of the subtasks I had defined during the development. This made the development much easier to accomplish, as when I would come back to code at a later date, the flowcharts would explain the code in a much more comprehensible way that commenting could, additionally, I could tick off sections of the flowchart as they were completed.

Additionally, I believe a key factor toward the project's success was the regular meetings with the project supervisor. These meetings provided me with useful feedback and guidance as well as helped me develop communication skills and project organisation skills. These meetings required me to set my own deadlines for subtasks and prioritise tasks appropriately, a skill that I believe will be extremely valuable when undertaking projects in the future.

Finally, the act of gathering the materials and writing this report has caused me to develop beneficial skills in researching and presenting said research in a sensible and structured way. It has given me confidence in my ability to take an initial problem and research, design, implement, and finally, present my solution to it. This skill can be transferred into new projects I will tackle in the future.

# Appendix

## A: Instructions

Provided alongside this document is an archive of the complete code listings for the project. The instructions for installing and running the system are listed below.

### Prerequisites

- NodeJS and NPM installed.
- Optional: your own MongoDB server.

### Install & Run

1. Uncompress the archive  
Optional: Open `config.js` and your own values for the MongoDB URI, secret, and port values.
2. Open a terminal in the folder and enter `npm install`
3. After the installation is complete enter `npm run start`
4. Open a browser and navigate to `localhost:3000`
5. If using the default MongoDB server provided, the following credentials have admin access:  
Username: `admin`  
Password: `password`

**B: Test Cases:**

Test case ID: 1		Purpose: User creation	
Environment: Firefox browser, Windows 10			
Preconditions: There is not an existing user with the given username or email			
Step	Action	Response	Pass/Fail
1	From any page, click on the sign up button in the nav bar.	The registration page is displayed	Pass
2	Enter a username	The username field is filled	Pass
3	Enter an email	The email field is filled	Pass
4	Enter a password	The password field is filled but not visible	Pass
5	Re-enter the password in the confirmation field	The second password field is filled but not visible	Pass
6	Click the submit button	The login page is displayed with a success message	
Comments:			
Related Tests: 2			

Test case ID: 2		Purpose: User authentication	
Environment: Firefox browser, Windows 10			
Preconditions: A user has been created			
Step	Action	Response	Pass/Fail
1	From any page, click on the log in button in the nav bar.	The login page is displayed	Pass
2	Enter your username	The username field is filled	Pass
3	Enter your password	The password field is filled but not visible	Pass
4	Click the submit button	The system redirects to the homepage. The username appears in the top right of	Pass

		the nav bar	
5	Click on the username in the navbar	The profile page is displayed	Pass
Comments:			
Related Tests: 1			

Test case ID: 3		Purpose: Admin authentication	
Environment: Firefox browser, Windows 10			
Preconditions: An admin user has been created			
Step	Action	Response	Pass/Fail
1	From any page, click on the log in button in the nav bar.	The login page is displayed	Pass
2	Enter your admin username	The username field is filled	Pass
3	Enter your password	The password field is filled but not visible	Pass
4	Click the submit button	The system redirects to the homepage. The username and an admin button appear in the top right of the nav bar	Pass
5	Click on the username in the navbar	The profile page is displayed	Pass
Comments: Currently there is no way to create the first admin user without access to the database.			
Related Tests:			

Test case ID: 4		Purpose: Adding a product	
Environment: Firefox browser, Windows 10			
Preconditions: An admin user is logged in. The tester has a suitable product image on their machine.			
Step	Action	Response	Pass/Fail
1	From any page, click on the	The admin settings panel is displayed	Pass

	admin button in the nav bar.		
2	In the sidebar, click on products	The products page is displayed, with a table of all the products in the system.	Pass
3	Click on the add product button	A form is displayed requesting a title, slogan, description and an image.	Pass
4	Fill in the form	The form fields are filled	Pass
5	Select "upload image" and choose a product image	A file explorer window opens and lets you choose a file. When a file is chosen the filename appears in the box.	Pass
6	Click the "add" button	The system redirects to the previous products page. The table now has a new product in it. There is a green notification box that says the product has been added.	Pass
Comments:			
Related Tests:			

Test case ID: 5		Purpose: Viewing a product's price data	
Environment: Firefox browser, Windows 10			
Preconditions: None			
Step	Action	Response	Pass/Fail
1	Open the system and navigate to a product page. Scroll to the statistics section	The page displays a line graph of bids and asks, it also displays a table containing the highest bid, lowest ask, and the number of bids, asks and orders. The values update as new bids and asks come in.	Pass
2	Scroll to the bottom and click on the bid tab.	The page displays a table of all of the historical bids placed on the item. The columns are price, date and status. The table is searchable and sortable.	Pass
3	Click on the ask tab	Similar to the bids tab, the product displays a table of all of the historical	Pass

		asks placed on the product. The columns are price, date and status. The table is searchable and sortable.	
4	Click on the orders tab	The page displays a table of all of the previous orders for the object. It displays the price and the date. The table is searchable and sortable.	Pass
Comments:			
Related Tests:			

Test case ID: 6		Purpose: Placing a bid	
Environment: Firefox browser, Windows 10			
Preconditions: Registered an account but remain logged out until step 3			
Step	Action	Response	Pass/Fail
1	Open the system and navigate to a product page	The product information and image is displayed and there are 4 options below: Buy, Bid, Sell or ask	Pass
2	In the place bid form enter 100 into the field and submit.	The system redirects to the login page.	Pass
3	Enter login credentials and submit.	The system redirects back to the previous product page.	Pass
4	In the place bid form enter 0 into the field and submit.	The system responds with an invalid input message	Pass
5	Enter -1 into the same field and submit	The system responds with an invalid input message	Pass
6	Enter 100 into the same field and submit	The system responds with a message saying the bid was placed successfully.	Pass
Comments:			
Related Tests:			

Test case ID: 7		Purpose: Placing an ask	
-----------------	--	-------------------------	--

Environment: Firefox browser, Windows 10			
Preconditions: Registered an account but remain logged out until step 3			
Step	Action	Response	Pass/Fail
1	Open the system and navigate to a product page	The product information and image is displayed and there are 4 options below: Buy, Bid, Sell or ask	Pass
2	In the place ask form enter 100 into the field and submit.	The system redirects to the login page.	Pass
3	Enter login credentials and submit.	The system redirects back to the previous product page.	Pass
4	In the place ask form enter 0 into the price field and submit.	The system responds with an invalid input error message	Pass
5	Enter -1 into the same field and submit	The system responds with an invalid input error message	Pass
6	Enter 100 into the same field and submit	The system responds with a message saying the bid was placed successfully.	Pass
Comments:			
Related Tests:			

Test case ID: 8		Purpose: Sell to a bidder	
Environment: Firefox browser, Windows 10			
Preconditions: User is logged in. Tester has another user's credentials who has already placed the highest bid on the product.			
Step	Action	Response	Pass/Fail
1	On the product page scroll down and click the red sell now button.	The system redirects to an order confirmation page	Pass
2	Click the button to confirm the order details and place the order.	The user is redirected to a order success page	Pass

3	Go to the user profile and navigate to the orders panel	The sold table in the orders panel shows the order that has just been placed.	Pass
4	Go to the product page and go to the orders table at the bottom	The table displays the price and date of the order that has just been placed.	Pass
5	Log in to the second user account (the buyer's) and navigate to the profile	The notification panel on the profile main page shows a notification saying that a user has accepted the bid and the order is complete.	Pass
Comments:			
Related Tests:			

Test case ID: 9		Purpose: Buy from an asker	
Environment: Firefox browser, Windows 10			
Preconditions: User is logged in. Tester has another user's credentials that has previously placed the lowest ask for the product.			
Step	Action	Response	Pass/Fail
1	On the product page scroll down and click the green buy now button.	The system redirects to an order confirmation page with order details and an address form.	Pass
2	Fill out the address and confirm the order by clicking the submit button.	The user is redirected to a order success page	Pass
3	Go to the user profile and navigate to the orders panel	The bought table in the orders panel shows the order that has just been placed.	Pass
4	Go to the product page and view the orders table at the bottom	The table displays the price and date of the order that has just been placed.	Pass
5	Log in to the second user account (the seller's) and	The notification panel on the profile main page shows a notification saying that a	Pass

	navigate to the profile	user has accepted the ask and the bought the product.	
Comments:			
Related Tests:			

Test case ID: 10		Purpose: View your own bids & asks	
Environment: Firefox browser, Windows 10			
Preconditions: The user is logged in. The user has placed at least one bid or ask in the past.			
Step	Action	Response	Pass/Fail
1	Log in to the system	The system redirects the user to homepage.	Pass
2	On the navigation bar, click on the username.	The system displays the user profile area.	Pass
3	From the profile page sidebar click on the bids & asks link	A page is displayed with two tables, one for bids and one for asks. The tables are paginated, sortable and searchable.	Pass
Comments:			
Related Tests:			

Test case ID: 11		Purpose: View your orders	
Environment: Firefox browser, Windows 10			
Preconditions: The user is logged in. The user has placed at least one order in the past.			
Step	Action	Response	Pass/Fail
1	Log in to the system	The system redirects the user to homepage.	Pass
2	On the navigation bar, click on the username.	The system displays the user profile area.	Pass

3	From the profile page sidebar click on the orders link	A page is displayed with a two tables of all of the orders that you have bought and sold.	Pass
Comments:			
Related Tests:			

Test case ID: 12		Purpose: Update a product	
Environment: Firefox browser, Windows 10			
Preconditions: Admin user is logged in.			
Step	Action	Response	Pass/Fail
1	Login and navigate to the admin panel	The system redirects the user to the admin homepage.	
2	Navigate to the products page and click the edit button on a product in the table.	The system displays a form with the product name, slogan and description fields filled in.	Pass
3	Edit the name field	The field input changes.	Pass
4	Upload a different image	There is no option to upload an image on this page.	Fail
5	Submit the form	The system redirects back to the product table page and displays a success message	Pass
Comments: This case fails as the form does not give the option to change the product image.			
Related Tests: 4			

Test case ID: 13		Purpose: Search for a product	
Environment: Firefox browser, Windows 10			
Preconditions: None			

Step	Action	Response	Pass/Fail
1	In the search field on the home page, search for the term "phone" and submit	The system redirects to a page with search results. There is one result for iPhone X.	Pass
2	Click on the first result	The system redirects to the iPhone X product page	Pass
Comments: The search functionality is functional however it is not fully featured.			
Related Tests:			

Test case ID: 14		Purpose: Notifications for orders in profile	
Environment: Firefox browser, Windows 10			
Preconditions: Two user accounts, one of which has placed the lowest ask for a product			
Step	Action	Response	Pass/Fail
1	Log into the system and navigate to the product page.	The product page is displayed with an option to buy for the lowest ask.	Pass
2	Click on the buy now button and place the order	The system shows an order confirmation and then an order complete page.	Pass
3	In another window, open the system and log in to the user who placed an ask.	The system redirects to the home page.	Pass
	Navigate to the user profile area	The system shows the profile area with a notification section. There is a notification saying that the product has been bought and the ask fulfilled.	Pass
Comments:			
Related Tests:			

Test case ID: 15		Purpose: Push notifications for new high bids and low asks	
------------------	--	--	--

Environment: Firefox browser, Windows 10			
Preconditions: Two windows with the system open. Allow notifications when requested			
Step	Action	Response	Pass/Fail
1	In one window, login to the system and navigate to a product page.	The system displays the product page	Pass
2	In the other window, navigate to the same product page and when prompted click allow notifications.	The system displays the product page.	Pass
3	In the first window with the user logged in, place a new highest bid.	The page refreshes and the page now displays a message saying the bid has been placed and the user is currently the highest bidder. In the other window a notification appears saying that there is a new high bid.	Pass
4	In the first window with the user logged in, place a new lowest ask.	The page refreshes and the page now displays a message saying the ask has been placed and the user is currently the lowest asker. In the other window a notification appears saying that there is a new low ask.	Pass
Comments:			
Related Tests:			

Test case ID: 16		Purpose: Product page updates in real-time	
Environment: Firefox browser, Windows 10			
Preconditions: Two windows open on the same product page, one user logged in.			
Step	Action	Response	Pass/Fail
1	In one window, login to the system and navigate to a product page.	The system displays the product page	Pass
2	In the other window, navigate	The system displays the same product	Pass

	to the same product page. Scroll to the bottom to view the price graph	page.	
3	In the first window with the user logged in, place a new bid.	The page refreshes and the page now displays a message saying the bid has been placed. The price graph in the other window updates to show the new bid price.	
4	In the first window with the user logged in, place a new ask.	The page refreshes and the page now displays a message saying the ask has been placed. The price graph in the other window updates to show the new ask price.	
Comments:			
Related Tests:			

## References

- [1] Netimperative 2016, *15m Brits have an old mobile phone gathering dust in drawers at home* [Online]. Available: <http://www.netimperative.com/2016/12/15m-brits-old-mobile-phone-gathering-dust-drawers-home> [Accessed: 08- May- 2018].
- [2] Node.js Foundation, *Overview of Blocking vs Non-Blocking*, 2018. [Online]. Available: <https://nodejs.org/en/docs/guides/blocking-vs-non-blocking> [Accessed: 06- May- 2018].
- [3] Stack Overflow, *Stack Overflow Developer Survey 2018*, 2018. [Online]. Available: <https://insights.stackoverflow.com/survey/2018> [Accessed: 08- May- 2018].
- [4] Modulecounts, 2018. [Online]. Available: <http://www.modulecounts.com> [Accessed: 02- May- 2018].
- [5] M. Carbou, *Reverse Ajax, Part 1: Introduction to Comet*. IBM, 2011. [Online]. Available: <https://www.ibm.com/developerworks/library/wa-reverseajax1/wa-reverseajax1-pdf.pdf> [Accessed: 02- May- 2018]. pp. 3-7.
- [6] J. Hanson, *What is HTTP Long Polling?*, PubNub, 2018. [Online]. Available: <https://www.pubnub.com/blog/2014-12-01-http-long-polling> [Accessed: 08- May- 2018].
- [7] A. T. Holdener, *Ajax, the definitive guide: Interactive Applications for the Web*. Beijing: O'Reilly, 2008. pp. 325-329.
- [8] A. Bhatt, *A Comparison of Push vs Pull Ajax*, InfoQ, 2007. [Online]. Available: <https://www.infoq.com/news/2007/07/pushvspull> [Accessed: 01-May-2018].
- [9] L. Golab and M. Özsu, *Issues in data stream management*, ACM SIGMOD Record, vol. 32, no. 2, pp. 5-14, 2003.
- [10] *Frequently asked questions*. RethinkDB, [Online]. Available: <https://www.rethinkdb.com/faq/> [Accessed: 06-May-2018].

- [11] Slava Akhmechet. *RethinkDB is shutting down*. RethinkDB Blog, 2016. [Online]. Available: <https://rethinkdb.com/blog/rethinkdb-shutdown/> [Accessed: 01-May-2018].
- [12] *PubNub Publish/Subscribe Tutorial*, PubNub, 2018. [Online]. Available: <https://www.pubnub.com/docs/tutorials/pubnub-publish-subscribe> [Accessed: 11- May-2018].
- [13] A. Mardan, *Practical Node.js*, Chapter 7: *Boosting Your Node.js Data with the Mongoose ORM Library*, APRESS, 2014, pp. 149-172.
- [14] Pavlou, Liang and Xue, *Understanding and Mitigating Uncertainty in Online Exchange Relationships: A Principal-Agent Perspective*, MIS Quarterly, vol. 31, no. 1, 2007.
- [15] *Market share for mobile, browsers, operating systems and search engines*, Netmarketshare.com, 2018. [Online]. Available: <https://netmarketshare.com> [Accessed: 11- May- 2018].
- [16] B. Shneiderman, *Designing the user interface: strategies for effective human-computer interaction*. Boston: Pearson, 2017. ISBN: 9780134380384
- [17] Mozilla Contributors, *Modern web app architecture: MVC Architecture*, Mozilla Developer Network Web Docs [Online]. Available: [https://developer.mozilla.org/en-US/Apps/Fundamentals/Modern\\_web\\_app\\_architecture/MVC\\_architecture](https://developer.mozilla.org/en-US/Apps/Fundamentals/Modern_web_app_architecture/MVC_architecture) [Accessed: 07-May-2018].
- [18] *Mongo Express Angular Node*. [Online]. Available: <http://mean.io/> [Accessed: 11-May-2018].
- [19] *Data Model Design*, MongoDB Documentation. [Online]. Available: <https://docs.mongodb.com/manual/core/data-model-design/> [Accessed: 01-May-2018].
- [20] MongoDB, Inc. *Performance Best Practices for MongoDB*, MongoDB, 2017 [Online]. Available: [https://webassets.mongodb.com/\\_com\\_assets/collateral/MongoDB-Performance-Best-Practices.pdf](https://webassets.mongodb.com/_com_assets/collateral/MongoDB-Performance-Best-Practices.pdf) [Accessed: 02- May- 2018]. pp. 3-7.
- [21] *Passport*, [Online]. Available <http://www.passportjs.org/> [Accessed: 02-May-2018].

[22] Node.js Foundation, *The Node.js Event Loop, Timers, and process.nextTick()*. [Online]. Available: <https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/> [Accessed: 06- May- 2018].

[23] OODesign.com , *Design Principles*. [Online]. Available: <https://www.oodeesign.com/design-principles.html> [Accessed: 06-May-2018].

[24] J. Martin, *Principles of Object-Oriented Analysis and Design*. Englewood Cliffs, NJ: Prentice-Hall, 1993. ISBN: 978-0137208715

[25] *AppTelemetry Page Speed Monitor*, FabaSoft. [Online] Available: <https://www.fabasoft.com/en/products/fabasoft-apptelemetry> [Accessed: 10-May-2018].

[26] P. Krill, *Socket.IO JavaScript framework ready for real-time apps*, InfoWorld [Online] Available: <https://www.infoworld.com/article/2607757/javascript/socket-io-javascript-framework-ready-for-real-time-apps.html> [Accessed: 10-May-2018].

The following references are not directly cited in the text, however they did provide valuable guidance throughout the project:

[27] P. Mulder & K. Breseman, *Node.js for Embedded Systems: Using Web Technologies to Build Connected Devices*. Beijing : O'Reilly, 2017. ISBN: 9781491928998

[28] K. Chodorow, *MongoDB: the Definitive Guide*, Beijing : O'Reilly, 2013. ISBN: 9781449344689

[29] J. Purushothaman, *RESTful Java web services - Second Edition*. Birmingham : Packt Publishing, 2015. ISBN: 9781784399092

[30] L. Richardson & S. Ruby, *RESTful web services*. Beijing : O'Reilly, 2008. ISBN: 9780596529260