CS3219 Final Report

CS3219 Team 04

Cai Qifan David	A0200001U
Michelle Yong Kai Wen	A0189286J
Bhadani Simran	A0201225B
Princess Priscilla Paulson	A0201363W

Table of contents

1. Background and purpose of project

Auction platform: E-Auction

A) What is our project about?

Our project is a web application that allows the auctioneer to auction their items and bidders to bid for items.

B) Background and Motivation

Due to the current covid situation, it is difficult or impossible to coordinate public events, especially auctions where there is a lot of verbal activity. Alternative solutions like online communication platforms such as zoom do not cater specifically to auctions and may lead to complications in online events - for example, a bidder's bid could be easily drowned in the chat and go unnoticed by the auctioneer.

Charities often conduct auctions for fundraising. However, given the restrictions, they have been unable to organize physical auctions with a large number of attendees. Covid-19 regulations have also made the logistics of organizing a physical auction difficult to manage.

To solve this issue, we plan to create an online auction platform where potential bidders can easily find auctions they are interested in, and where auctioneers can easily conduct auctions.

This can be achieved through having a well synchronized auction room in the web application that removes the hassle of organizing auctions. We want to simplify the steps needed for organizing and joining auctions by buyers and sellers to just a few clicks. This application also allows for small scale auctions to be held without the high costs and barriers to entry associated with conventional large-scale auctions. We will implement scalability features to allow large and small events alike to be held on our platform.

2. Individual Contributions to the project

Names	**Contributions**
Cai Qifan David	● Frontend (Sign Up/Login Page, Auction Room Page) ● Backend(Auctiondetails API, Auctionroommanager microservice) ● Deployment ● Documentation
Michelle Yong Kai Wen	● Frontend (Home Page, Create Auction Page, Profile page)

	• Backend (UserAccount API and Testing) • Documentation
Bhadani Simran	• Backend (UserAccount API, AuctionRoom API and Testing) • Setting up Kubernetes resources • Deployment • Documentation
Princess Priscilla Paulson	• Backed Testing • API gateway • Deployment • Documentation

For a detailed breakdown of tasks completed for each sprint, refer to Appendix A.

3. Requirements

3.1 Functional and non-functional requirements

S/N	Functional Requirement	Priority
User Authentication		
F1.1	The platform should allow users to **register** for an account using username, email and password.	High
	F1.1.1 The platform should only allow users with **emails ending with @u.nus.edu** to register for an account.	Medium
F1.2	The platform should allow users to **sign into** their account using email and password.	High
F1.3	Users can choose to stay signed in or log out after the session.	Low
F1.4	The platform should allow users to **logout** of their account.	High
Auction Management		
	Auction Listings	
F2.1.1	The platform should allow logged in users to **view** all upcoming auctions.	High
	F2.1.1.1 The platform should allow logged in users to view all auctions including past auctions.	Medium
F2.1.2	The platform should allow logged in users to **search** for upcoming auctions based on the **auction name**.	Medium

F2.1.3	The platform should allow logged in users to **filter** for upcoming auctions based on the selection of **item name**, **auction category** and **price range** indication of the auctions.	Medium
	Auction Creation	
F2.2.1	The platform should allow logged in users to **create** an auction.	High
F2.2.2	The platform should allow logged in users to **view** a summary of all auctions they had previously created.	Medium
F2.2.3	The platform should allow logged in users to **edit** details of an auction they created if it is not ongoing (before the auction starts and after the auction ends).	High
F2.2.4	The platform should allow logged in users to **delete** an auction they created if it is not ongoing and **end** an auction if it has already started.	High
F2.2.5	The platform should allow logged in users to **set a minimum increment** for each auction.	Medium
F2.2.6	The platform should allow logged in users to **upload images of jpeg format** for each auction.	Low
	Auction Session - Bidding Management	
F2.3.1	The platform should allow bidders to **bid** a higher price than the current highest bid or starting bid.	High
	F2.3.1.1 The platform should allow bidders to **update** their bids by placing a higher bid than the current highest bid.	High
F2.3.2	The platform indicates the **amount** of the highest bid currently.	High
F2.3.3	The platform indicates the **minimum amount** that the bidders can currently bid (either the minimum starting bid, or the current highest bid + the minimum increment)	High
F2.3.4	The platform should indicate who **won the bid** after the auction ends, as well as the winning bid.	High
F2.3.5	The platform should display the **minimum increment** amount for subsequent bids.	High
F2.3.6	The platform should display the **item details** for user reference : Item name, item description, category and item image .	High
F2.3.7	The platform should display the **end time** of the auction.	High
F2.3.8	When the auction ends, all users will be notified and redirected to the home page.	High
F2.3.9	Auction owners are able to end the auction prematurely before the specified end time of the auction.	High

F2.3.10	If the auction is ended without a bid, the auction owner will be notified that no user had bid for the item, and no transaction will take place.	Medium
	Auction Session - Chat Management	
F2.4.1	The platform should have an auction room chat function that is functional after the start time of the auction and is disabled once the auction ends.	Low
F2.4.2	The platform should allow all logged in users to submit a chat message through that chat function.	Low
F2.4.3	The platform should have a rate limiter to block users for 30 seconds when sending more than 5 messages within an interval of 10 seconds.	Low
F2.4.4	The platform should highlight the auctioneer's name with a tag in the chat log if they were to post on the chat feature.	Low
F2.4.6	The platform should indicate the username of the user that input the message.	Low
Currency Management		
F3.1	The platform should ensure that the bidder has **sufficient** currency in their wallet before they place a bid.	High
F3.2	The platform should **credit** the auctioneer's account with the virtual currency they receive from the sale of the auction item.	High
F3.3	The platform should **deduct** the successful bid from the highest bidder's account once the auction is closed.	High
F3.4	Users can **top up** currency into their account.	High

D) Non-Functional Requirements (NFRs)

S/N	Non-Functional Requirement	Priority
Performance		
NF1.1	The platform should update the **chat** almost instantly (within 1 second) when there are a **maximum of 10 users present**.	Low
NF1.2	The platform should update the **highest bid** almost instantly (within 1 second) when there are a **maximum of 10 users present.**	High
NF1.3	The auctioneer should be able to view their auction listing within 3 seconds after **publishing** it.	Medium
Security		
NF2.1	The platform should **only allow authenticated users** to perform functions, including setting up of an auction or making a bid.	Medium
NF2.2	**Passwords should be hashed** before storing into the database.	High
Scalability		

NF3.1	Each user should be able to see their message appear in the chat within **5 seconds** after sending a message when there are a **maximum of 100 users** sending **one message within 20 seconds** while an auction is occurring.	High
NF3.2	Database can store details for at least **1000 users** and **100 auction details**.	High
NF3.3	Each user should be able to see their message appear in the chat within **5 seconds** after sending the message when there are a **maximum of 30 users who are actively chatting** (sending 5 messages one after the other) while an auction is occurring.	High
Usability		
NF4.1	95 percent of auctioneers who have not hosted an online auction before should be able to publish an auction listing within 15 minutes given that they have all the information ready at hand.	Low

3.2 Testing of non-functional requirements

3.2.1 Scalability requirements

NF3.1

This requirement specifies that each user should be able to see their message appear in the chat within 5 seconds after sending a message when there are a maximum of 100 users sending one message within 20 seconds while an auction is occurring.

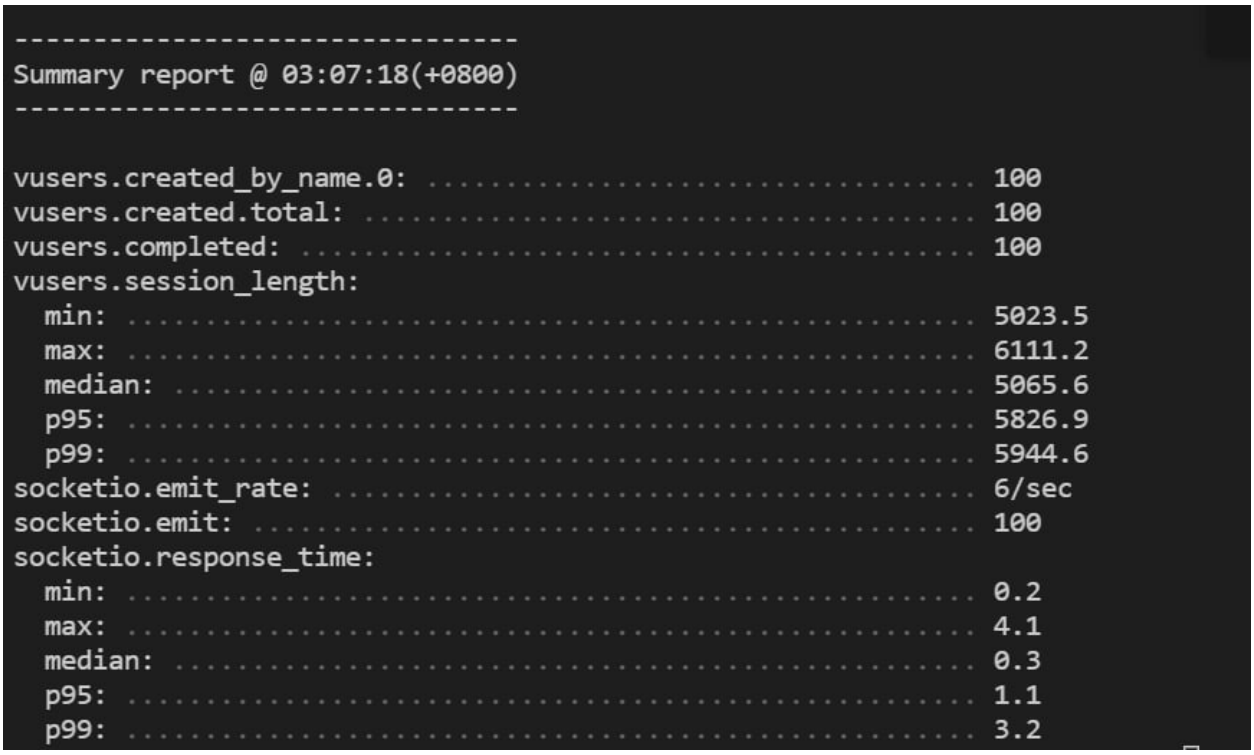
Artillery, a load testing tool for Socket.IO was used to test this requirement. Using Artillery, we simulated a 100 users sending a message in the chat within a duration of 20 seconds.

The following code snippet shows the Artillery test used for this:

```
config:
 target: "http://34.124.164.87/"
 socketio:
  transports: ["websocket"]
 phases:
  - duration: 20
    arrivalCount: 100
scenarios:
 - engine: "socketio"
```

```
flow:
  - emit:
      channel: "newChatMessage"
      data: "Hello! {{ $randomString() }}"
  - think: 5 # do nothing for 5 seconds, then disconnect
```

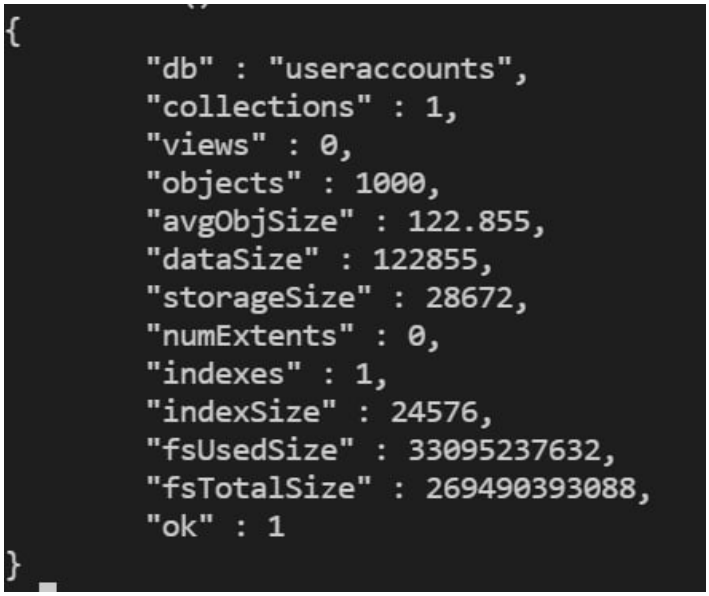
The test results obtained from artillery were as follows:



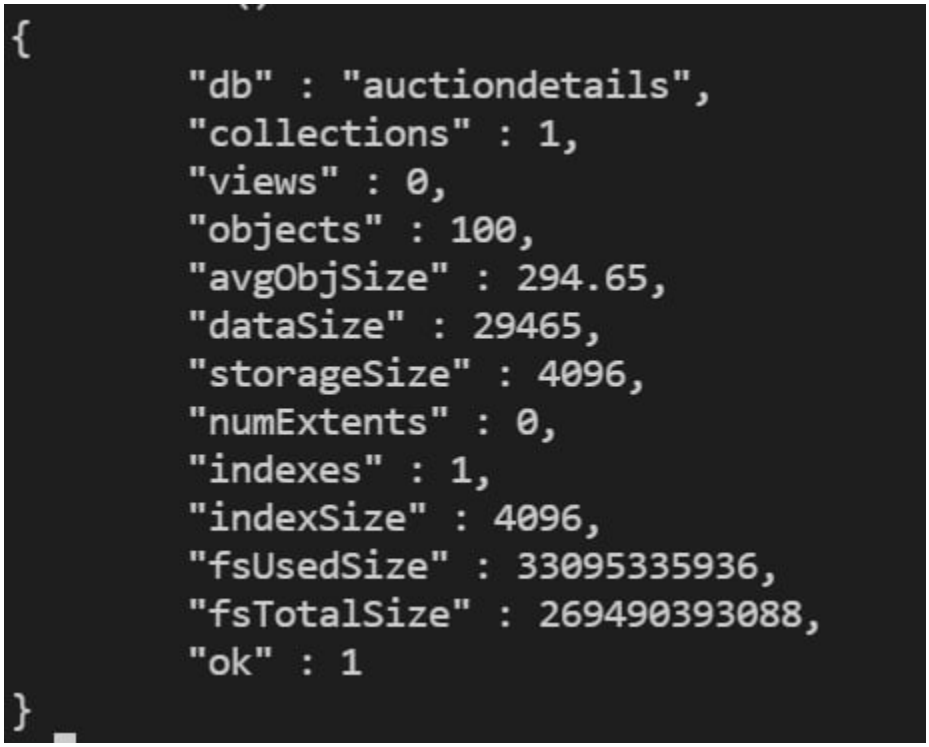
As can be seen from the screenshot above, the total number of users created was 100 and all were able to send a chat message successfully. The response time is indicative of the amount of time taken for the message to be displayed in the chat. The median socket response time is 0.3s. The maximum socket response time is 4.1 s which fits well within the response time of 5s stated in the requirements.

NF3.2

This requirement specifies that the database should be able to store details for at least a 1000 users and 100 auction details. In order to test these requirements, mock data consisting of a 1000 users and 100 auction details was generated and inserted into the user accounts and auction details databases respectively. The mongo shell command db.stats() was used to retrieve statistics for both the user accounts and auction details database as shown in the screenshots below:



In the screenshot above, the objects field represents the number of records in the useraccounts database which is equal to 1000. The dataSize represents the number of bytes taken up by these records which is equal to 122855 or approximately 0.12 MiB. The storage requested by the Persistent Volume kubernetes resource for the useraccounts database is 1 GiB. Thus 0.12 MiB of data can easily be stored.



In the screenshot above, the objects field represents the number of records in the auctiondetails database which is equal to 100. The dataSize represents the number of bytes taken up by these records which is equal to 29465 or approximately 0.028 MiB. The storage requested by the Persistent Volume

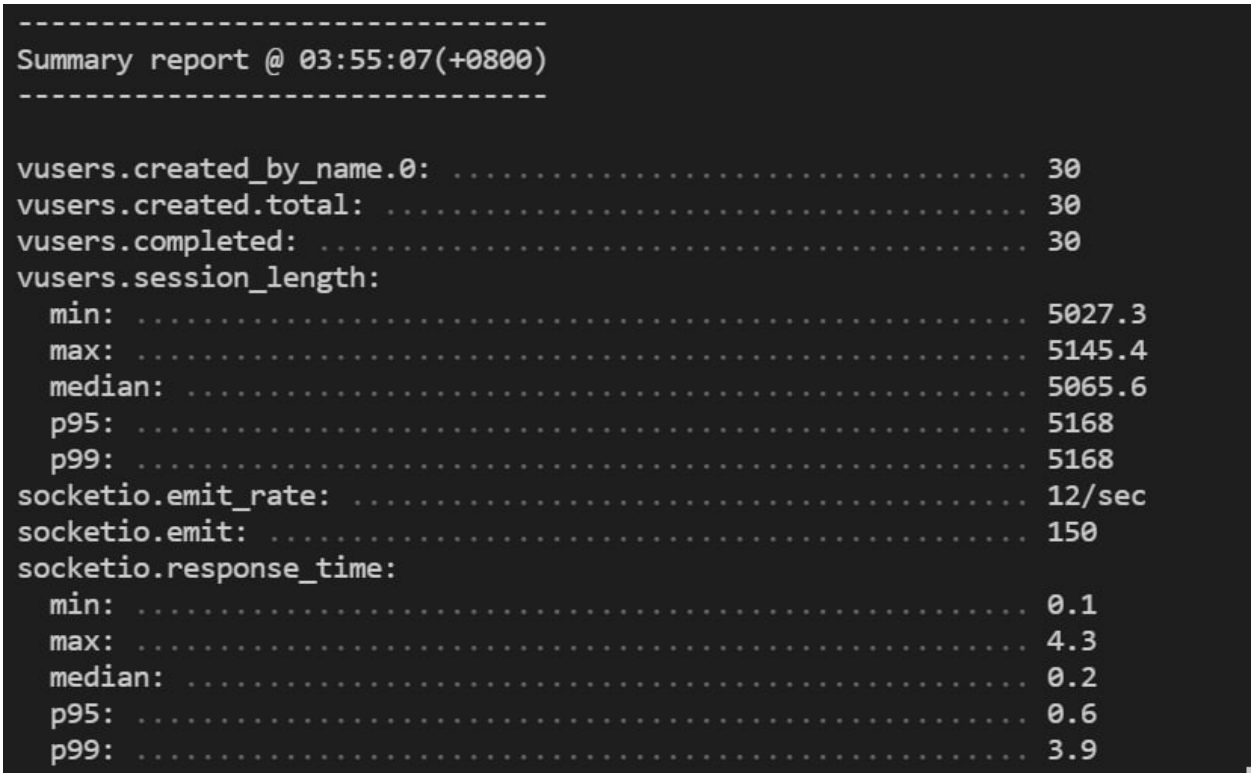
kubernetes resource for the auctiondetails database is 1 GiB. Thus 0.028 MiB of data can easily be stored.

NF3.3

This requirement specifies that each user should be able to see their message appear in the chat within 5 seconds after sending the message when there are a maximum of 30 users who are actively chatting (sending 5 messages one after the other) while an auction is occurring.

Using Artillery, we simulated 30 users sending 5 messages in a loop in the chat.

The test results obtained from artillery were as follows:



As can be seen from the screenshot above, the total number of users created was 30 and all were able to send chat messages successfully. The response time is indicative of the amount of time taken for the message to be displayed in the chat. The median socket response time is 0.2s and the maximum socket response time is 4.3 s which fits well within the response time of 5s stated in the requirements.

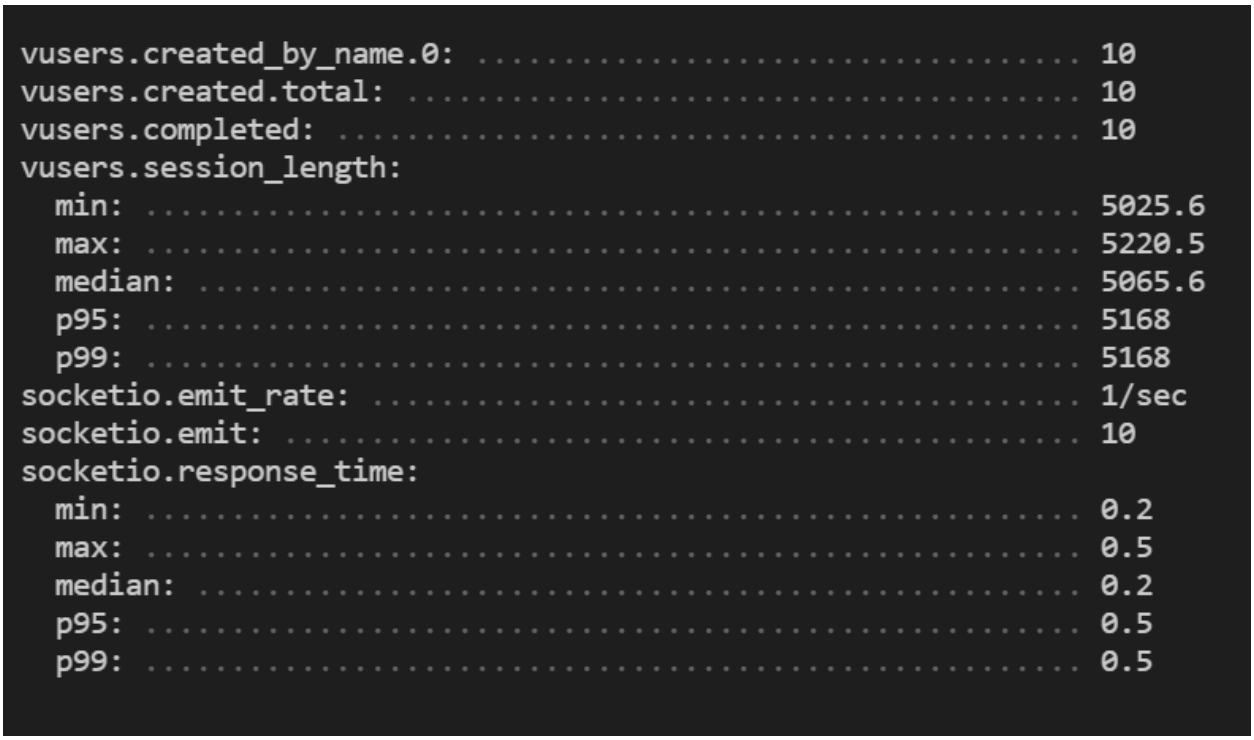
3.2.2 Performance requirements

NF1.1

This requirement specifies that the platform should update the chat almost instantly (within 1 second) when there are a maximum of 10 users present.

Using Artillery, we simulated 10 users sending a message in the chat.

The test results obtained from artillery were as follows:



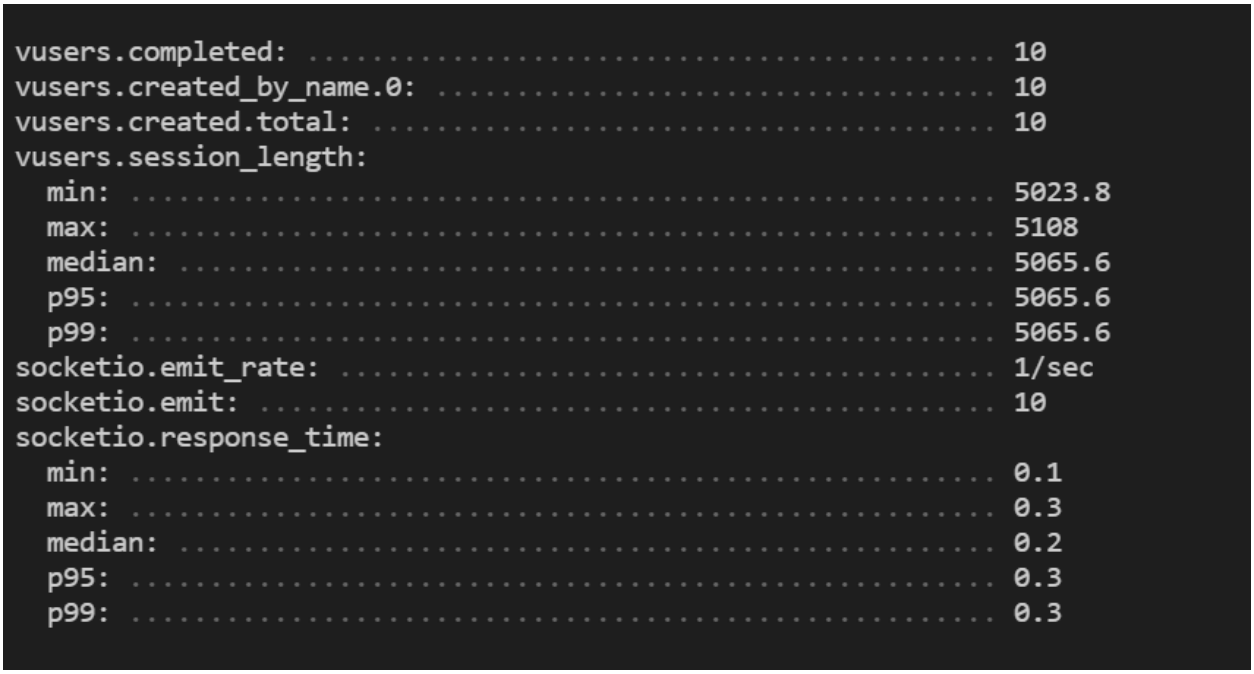
As can be seen from the screenshot above, the total number of users created was 10 and all were able to send chat messages successfully. The response time is indicative of the amount of time taken for the message to be displayed in the chat. The median socket response time is 0.2s and the maximum socket response time is 0.5s which fits well within the required response time of 1s.

NF1.2

This requirement specifies that the platform should update the highest bid almost instantly (within 1 second) when there are a maximum of 10 users present.

Using Artillery, we simulated 10 users sending a bid in the auction room.

The test results obtained from artillery were as follows:



As can be seen from the screenshot above, the total number of users created was 10 and all were able to send bids successfully. The response time is indicative of the amount of time taken for the bid to be displayed in the auction room. The median socket response time is 0.2s and the maximum socket response time is 0.3s which fits well within the required response time of 1s.

4. Development Process

4.1 Sprint Process

The project followed an Agile workflow with 2 week sprints. The project was planned at the start, into the product backlog.

Sprint planning was conducted at the start of each sprint, with the purpose of:

1. Assess tasks to be added into the current sprint and bring over uncompleted tasks from the previous sprint, including the discussion of time estimated for each task.
2. Ensure that all the features are working before working on new tasks.

During each sprint, standup was conducted every few days to keep the team members updated on the progress and resolve any issues that a team member might face. Our team did not opt for a daily sprint as not much work would have been done for any fruitful standup session.

Sprint Review was conducted at the end of each sprint, with the purpose of:

1. Identifying the areas of improvements of the sprint process.

2. Ensure that all the tasks for the week have been completed or discuss why it could not be completed.

See Appendix A for a breakdown of individual sprint tasks.

4.2 Tech Stack

Tech choice	Rationale
Frontend: React	● Fast Learning curve ● Highly popular and supported ● Fast, scalable and simple ● Creation of reusable UI components
Backend: NodeJS/Express	● Non-blocking model allows for serving of multiple simultaneous requests and easy scaling ● Reduces need to learn additional technologies if developed alongside react ● Highly popular and supported
Database: Firebase/MongoDB	● Firebase for easy image storage capabilities and quick deployments for trivial databases required for microservices ● Mongodb is built to scale up quickly ● Nosql allows for faster pace of development
Pub-Sub Messaging: Socket.io	● Scalable to fit microservice architecture needs
Containerization Tool: Docker	● Popular containerization tool to create microservice images that can be easily scaled
Container Orchestration: Kubernetes	● Popular way to scale microservices ● Highly supported
CI/CDs: Github Actions	● Readily available and accessible for github for easy configuration
Deployment Platform: Google Cloud Platform	● Free $300 credits ● Allows quick serverless deployments for components
Project Management Tool: Jira	● Popular and familiar project management tool ● Easy to use interface that allows for project planning and execution

4.3 Project Management

4.3.1 JIRA

Jira was used as the project management tool to facilitate the sprint planning.

Task workflow was set up on Jira and followed throughout the project.

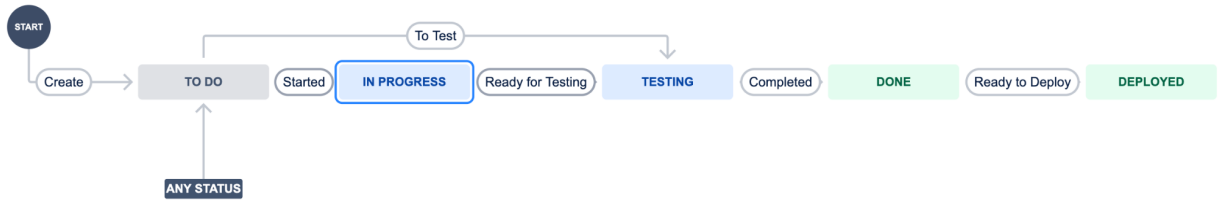


Fig 4.3.1.1 Task workflow on Jira

Task was written into the backlog and added to each sprint during the sprint planning. Team members update their board while working on the task to keep each other updated on the progress on days without standup.

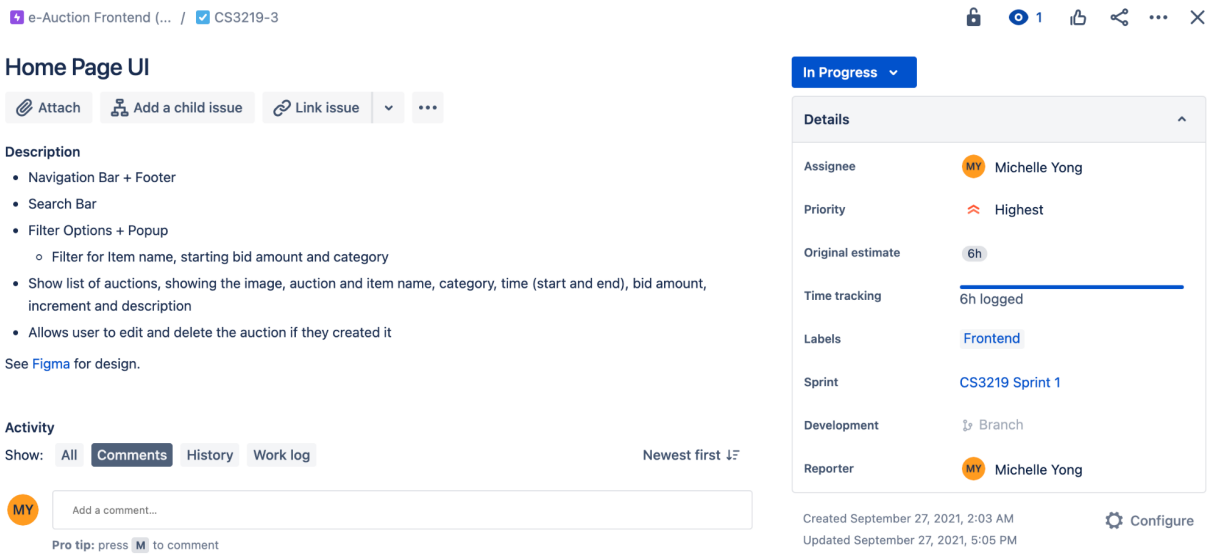


Fig 4.3.1.2 Example of task created on Jira

4.3.2 CI/CD

The team used github actions for continuous integration and continuous deployment.

```yaml
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  build:

    runs-on: ubuntu-latest
    strategy:
      matrix:
        node-version: [12.x, 14.x, 16.x]
        # See supported Node.js release schedule at https://nodejs.org/en/about/releases/

    steps:
    - uses: actions/checkout@v2
    - name: Use Node.js ${{ matrix.node-version }}
      uses: actions/setup-node@v2
      with:
        node-version: ${{ matrix.node-version }}
        cache: 'npm'
        cache-dependency-path: ./backend/auctiondetails/package-lock.json
    - run: npm i
      working-directory: ./backend/auctiondetails
    - run: npm run build --if-present
      working-directory: ./backend/auctiondetails
    - run: npm test
      working-directory: ./backend/auctiondetails
```

Fig 4.3.2.1 Continuous integration configuration file example

To achieve continuous integration, a configuration file was written such that a workflow will be run every time new code is pushed to the main branch of our project repository. The workflow includes building the project and testing the various microservices to ensure that there are no code errors, allowing team members to frequently perform code merges.

```yaml
name: Build and Deploy auctiondetails

on:
  push:
    branches:
      - main

env:
  PROJECT_ID: ${{ secrets.GCP_PROJECT }}
  GKE_CLUSTER: cluster-1    # TODO: update to cluster name
  GKE_ZONE: asia-southeast1-a  # TODO: update to cluster zone
  DEPLOYMENT_NAME: auctiondetails # TODO: update to deployment name
  IMAGE: auctionbackend-auctiondetails

jobs:
  setup-build-publish-deploy:
    name: Setup, Build, Publish, and Deploy
    runs-on: ubuntu-latest
    environment: production

    steps:
    - name: Checkout
      uses: actions/checkout@v2

    # Setup gcloud CLI
    - uses: google-github-actions/setup-gcloud@v0.2.0
      with:
```

Fig 4.3.2.2 Continuous deployment configuration file example

To achieve continuous deployment, a configuration file was written such that a workflow will be run every time new code is pushed to the main branch of our project repository.

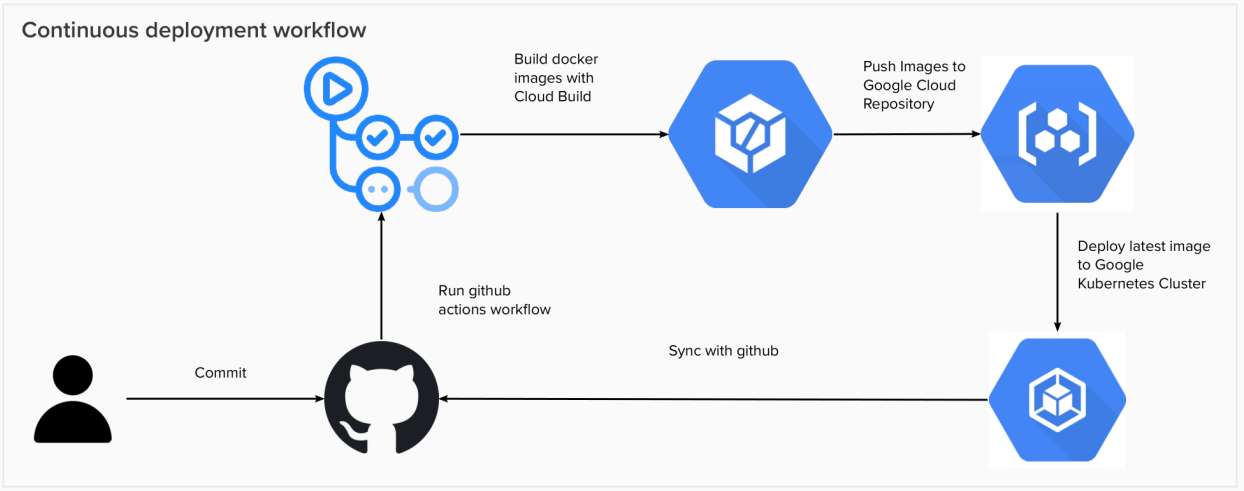


Fig 4.3.2.3 Illustration of continuous deployment workflow for backend microservices

Microservices backend deployment

For microservices deployment, the workflow has a few steps. Firstly, docker images are built. Next, the built images are published the the google cloud repository(gcr) and lastly, the images are deployed on the google kubernetes engine(gke) according to the specified kubernetes configuration files written in the project to ensure scalability and availability.

Frontend and Aggregator/proxy server deployment

For frontend and aggregator/proxy deployment, the workflow was similar. However these were deployed on google cloud run which did not require or need custom kubernetes configuration. This was because we did not see the need for these deployments to have scalability micromanagement. Therefore, google cloud run was chosen to remove the need for auto scaling configuration via kubernetes.

4.3.3 Manual Deployment for development and testing

Shortcut steps to get everything deployed and running

Assuming the repo has been cloned locally and docker desktop and kubectl are available,

1. Go to the root folder and run 'docker-compose build' to build images.
2. To run the metrics server for the HPA to work run `kubectl apply -f ./k8/hpa/metrics-server.yaml` from the root folder.
3. Also, ensure that nginx-ingress controller is configured properly by running `kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.0.4/deploy/static/provider/cloud/deploy.yaml`
4.
5. Make sure the `./k8/setup.sh` and `./k8/teardown.sh` files are in LF mode for End of line sequence
6. Then from a wsl/linux terminal, run `./k8/setup.sh` . This will ensure that the backend microservices are running in the docker-desktop Kubernetes cluster and are exposed through the Ingresses we configured.
7. To run the frontend, go to the ./frontend directory from the root folder.
8. Run yarn add all to install the necessary node modules.
9. Finally, run yarn start to run the frontend. The frontend client should be running on http://localhost:3000/

The next section runs through the commands we have compiled in setup.sh.

Explanation

There are three parts to the local deployment.

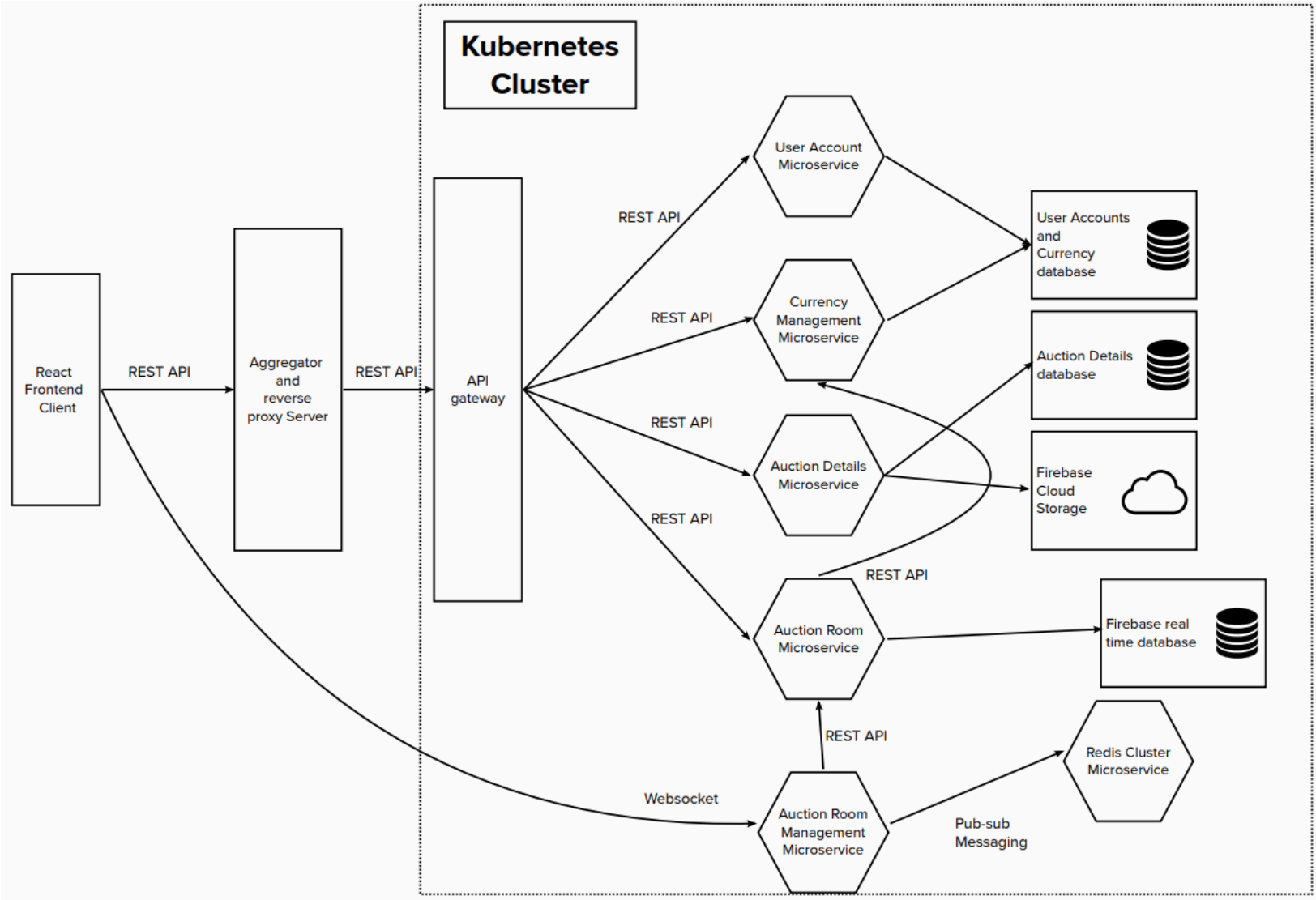
1. Create the kubernetes resources on the local cluster. This includes the ingress resources needed for the Api Gateway.*
2. Deploy the AggregateProxy docker container using docker-compose. This will make requests to the Api Gateway.
3. Deploy the frontend docker container using docker-compose too.

Part	Steps	Commands used
1	1. Relevant docker images need to be created 2. The necessary kubernetes resources must be applied	docker-compose build kubectl apply -f ./k8/services/3-useraccount.yaml; kubectl apply -f ./k8/services/5-auctiondetails.yaml; kubectl apply -f ./k8/ingress/unauth-ingress.yaml ; kubectl apply -f ./k8/ingress/auth-ingress.yaml ; (this is just a short excerpt of the kubectl apply commands used. The entire list can be found in ./k8/setup.sh)
2	1. Run the nodejs AggregateProxy server	cd k8\aggregate\aggregateproxy; npm ci && npm start
3	1. Stop the frontend container if there is an existing one 2. Then run the frontend container	docker rm auctiondocker; docker run -d --publish 3000:3000 --name auctiondocker -i -t auctionfrontend:latest

*Some prerequisites include

- Installing the ingress-nginx controller

5. Architecture Overview



This is the overall architecture of ***e-auction***. ***e-auction*** adopts a microservice architecture and each microservice implements a business capability within a bounded context. The **user account** microservice handles the management of user accounts. The **currency management** microservice handles transactions between users and user currency. The **auction details** microservice handles the management of auction information, such as auction item descriptions and auction scheduling. The **auction room management** microservice handles the real time proceedings of an auction, such as chat capabilities and bidding capabilities. The **redis-cluster** microservice allows for synchronization across all microservice instances for the

auction room management microservice. This is important to allow for scalability of the **auction room management** microservice and will be elaborated on in a later section. Finally, the **auction room** microservice handles the storage of auction proceeding details.

The microservices are exposed outside the kubernetes cluster via an API gateway using the ingress kubernetes resource. An aggregator service then connects to the API gateway externally which serves as an aggregator for multiple api calls and simultaneously a reverse-proxy for security reasons. This will be elaborated on in the later sections of this report. The frontend client will then make api calls through the aggregator/reverse proxy server in order to access the various microservices. Our choosing of the microservice architecture will also be explained below.

6. Microservices

6.1 Microservice Domain Driven Design (DDD)

From our understanding of microservices, we listed out the following design philosophies for the design of our microservice architecture:

1. Microservices should be designed around business capabilities rather than multiple horizontal layers.
    a. Each microservice will be derived from domain models and bounded contexts
2. Microservices should be loosely coupled .
    a. Developers should be able to work on different microservices simultaneously without the need to make changes to other microservices
3. Microservices should be highly cohesive.
    a. Each microservice has a single, well-defined purpose
    b. Functionality within each microservice is highly related.

6.1.1 Domain Analysis

We used domain analysis to model our microservices. This analysis was performed to help us define the boundaries of each individual service.

6.1.2 Domain Model

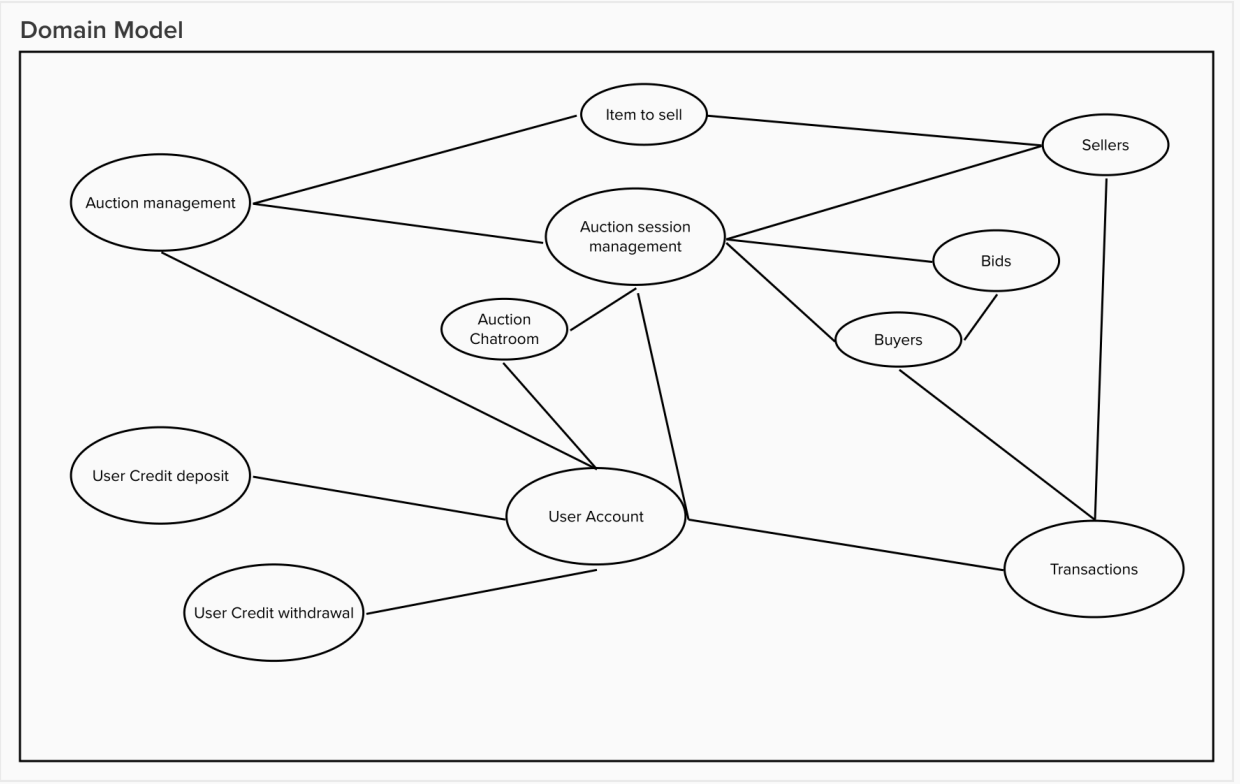


Fig 6.1.2.1 Diagram of Domain Model

- Auction session is placed in the center of the diagram and it is the core domain.
- Auction chatroom is a functionality that is closely related to Auction session
- User accounts, transactions and auction management are subdomains that support the core auction session

6.1.3 Bounded Contexts

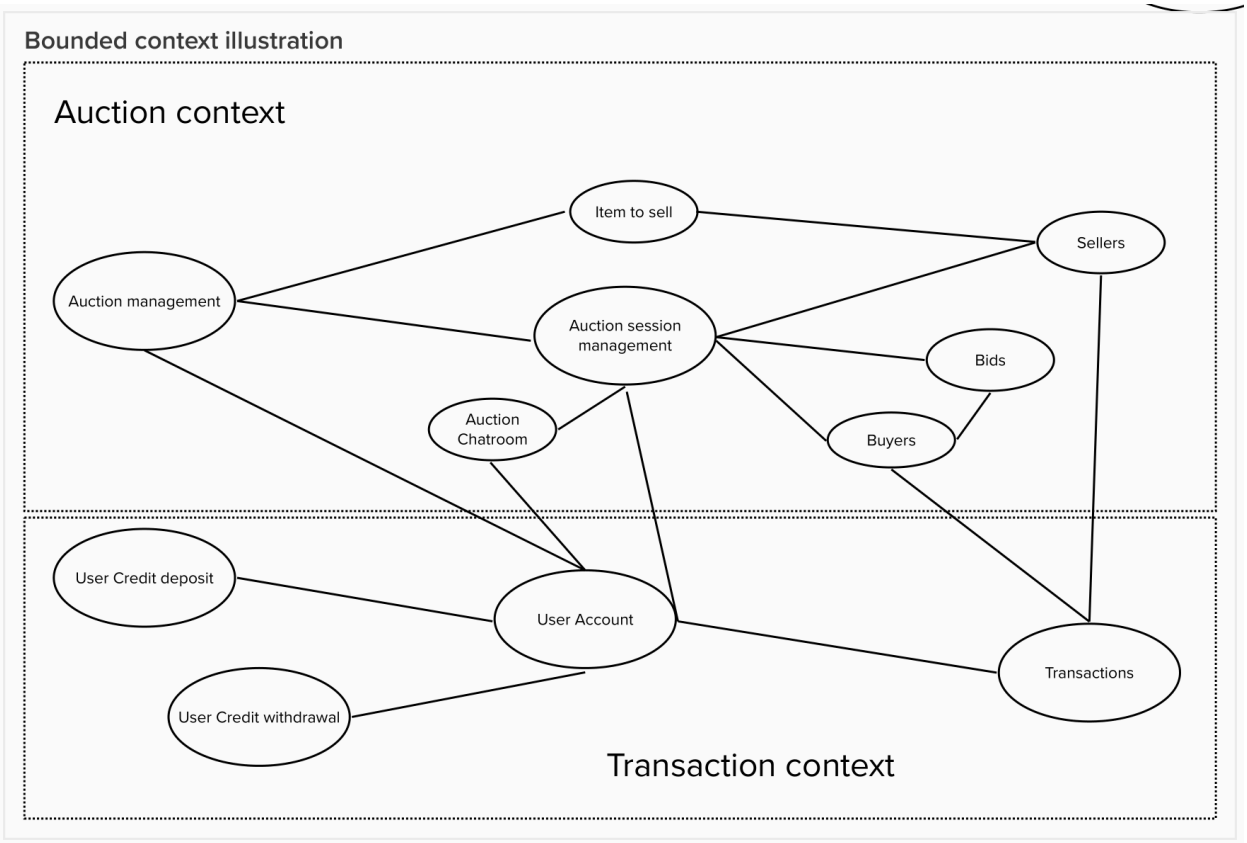


Fig 6.1.3.1 Illustration of Bounded Contexts

6.1.4 Identify microservice boundaries and functionalities

Based on the bounded contexts identified and the interactions between them, we came up with the following entities :

- Account
- Auction session
- Auction details
- Transaction

Based on the above entities, we identified 4 microservices. Non functional requirements also led the team to create one additional microservice : Auction room. The system needs to ensure the smooth proceeding of auction and safeguard against potential data loss due to server failure during the auction session. Therefore, the auction room microservice was created to enable persistent storage of the state of the auction which includes information such as the highest bid and highest bidder.

Microservice	Functionalities
User accounts	• User account creation • User account management
Auction Details	• Storage of information regarding the auction.
Auction room manager	• Management of auction processes, such as making bids and ending the auction. • Notifying all participants of the state of the auction, such as highest bid and end of auction. • Auction chat functionality for users to communicate during the auction session.
Currency management	• User credit deposits and withdrawals. • Transaction between auction buyers and sellers.
Auction room	• Persistent storage of auction state to safeguard against microservice failures

6.1.5 Microservices Evaluation

The identified microservices were evaluated against the following criteria before we reached the final iteration as displayed above. These criteria were sourced from the following **guide** :

- Each service has a single responsibility.
- There are no chatty calls between services. If splitting functionality into two services causes them to be overly chatty, it may be a symptom that these functions belong in the same service.
- Each service is small enough that it can be built by a small team working independently.
- There are no inter-dependencies that will require two or more services to be deployed in lock-step. It should always be possible to deploy a service without redeploying any other services.
- Services are not tightly coupled, and can evolve independently.
- Your service boundaries will not create problems with data consistency or integrity. Sometimes it's important to maintain data consistency by putting functionality into a single microservice. That said, consider whether you really need strong consistency. There are strategies for addressing eventual consistency in a distributed system, and the benefits of decomposing services often outweigh the challenges of managing eventual consistency.

6.2 Microservices Schema

The way we came up with the schema was to initially draft the properties related to each entity, such as a User, that might be relevant from the schema. Over time, as we started implementing the features, we refined the schema to include more properties that would be convenient to have in the schema.

This includes features such as currency, which was later added to the UserAccounts schema even though it is part of the Currency Management Schema because...

Below is the final schema used by the various databases in our microservices architecture.

6.2.1 User Accounts and Currency Management Schema (MongoDB)

The same schema is used by both models. Though we considered the database per service pattern, we decided to make a compromise in this case as the data is highly related. Both microservices are in charge of different entities (one performs requests related to a user and the other related to the balance they have), however, these entities are related to each other as a user has a specific balance of currency and the currency belongs to a user.

Because of this relationship, had we used two separate databases, then every time a new user is created, we would also have to create an entry in the currency management database leading to too many extra operations, additional overhead, and possible inconsistencies in the databases.

Fields (elaboration for less obvious fields)	**Properties**
username	required, trim
email	required, lowercase, trim
password (hashed by bcrypt)	
currency (balance remaining in the user's account)	
date (account creation date)	default: Date.now

6.2.2 Auction Details Schema (MongoDB)

Fields (elaboration for less obvious fields)	**Properties**
room_display_name	required
auction_item_name	required

owner_id	required
start_time (start time of the auction)	required
end_time (scheduled end time of the auction)	required
description (description of the auction)	-
increment (minimum difference between new bid and current highest bid needed)	required
minbid (minimum starting bid for the auction)	required

Note: Firebase Cloud Storage was used for storing and retrieval of images for the auction details microservice and thus images uploaded by the user are not part of the MongoDB schema.

6.2.3 Auction Room Schema (Firebase)

Fields (elaboration for less obvious fields)	**Properties**
roomname (same as id of auction detail pertaining to the room)	required
highest (contains user id of the highest bidder and the value of the highest bid)	default value set to 0
owner (user id of the auctioneer who owns the room)	required
bids (a collection of bids containing the user id of the bidder and the amount bid by the user)	only present when a bid has been placed

6.3 Microservice Features

6.3.1 Scalability using Horizontal Pod Autoscaling and Google Kubernetes Engine

One of the reasons microservices architecture was chosen was to ensure easy scalability. To this end, we configured the kubernetes component Horizontal Pod Autoscaler(HPA) for each of our microservices.

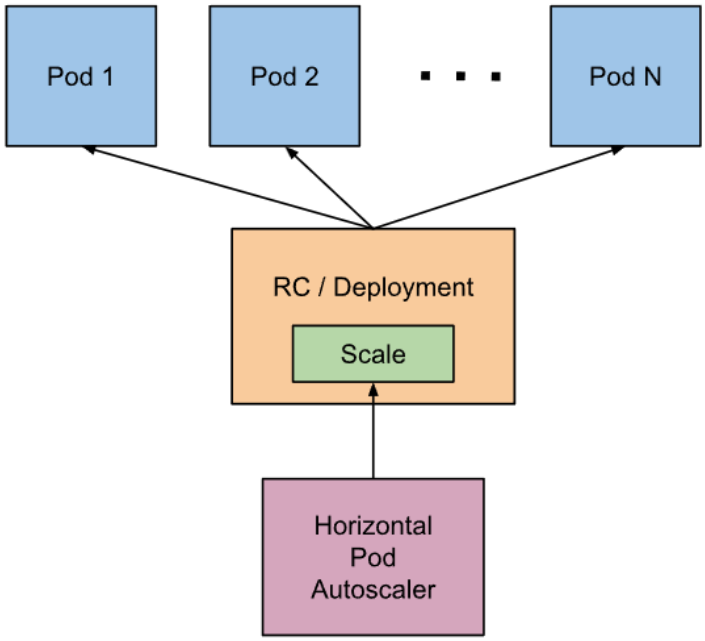


Fig 6.3.1.1 Illustration of Horizontal Pod Autoscaling in Kubernetes

For each microservice deployment configuration, the HPA will automatically scale the number of replicas for that deployment according to the min and max values provided.

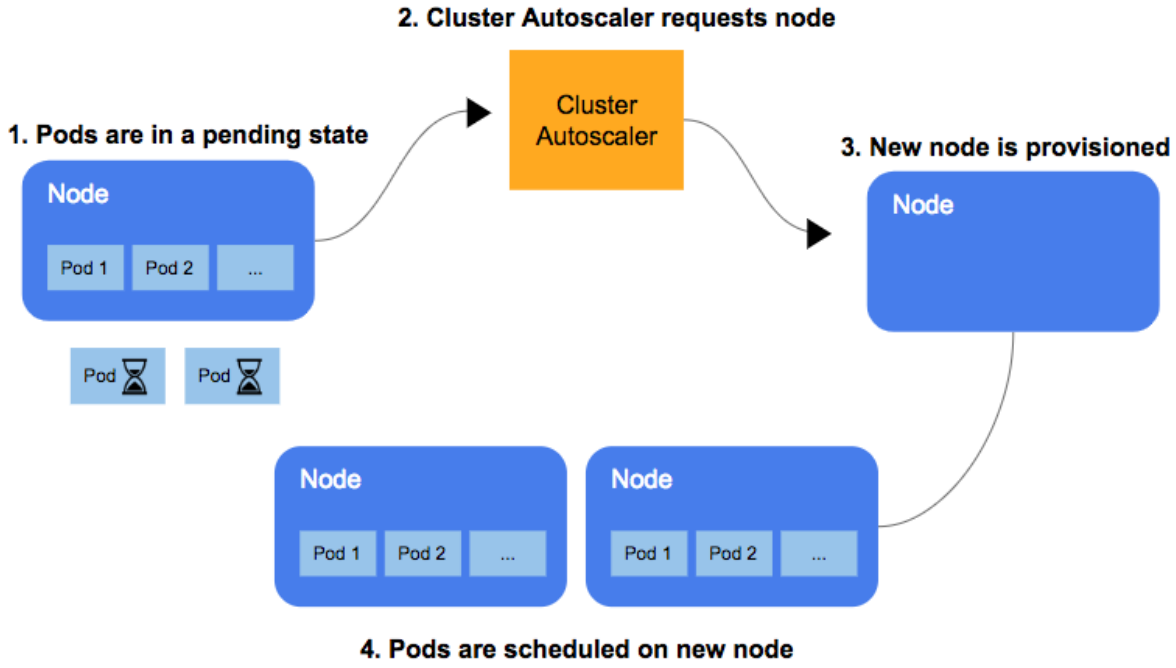


Fig 6.3.1.2 Illustration of Cluster Autoscaler in Google Kubernetes Cluster

At a higher level, we used the cluster autoscaler feature on Google Kubernetes Engine(GKE) to scale our kubernetes cluster. This is because each node also has a limit to the number of pods it can run. In order to accommodate more requests, the number of nodes has to be scaled as well to ensure scalability. This is where the cluster autoscaler comes in handy. According to the GKE documentation, “GKE's **cluster autoscaler** automatically resizes the number of nodes in a given node pool, based on the demands of your workloads. “ This enables us to easily scale the number of provisioned nodes to minimize costs and achieve scalability on demand.

6.3.1.1. Configuration details

The deployed kubernetes cluster was configured to have a minimum of 1 node and a maximum of 5 nodes. For local deployment and testing on the docker-desktop kubernetes cluster, only 1 node is configured.

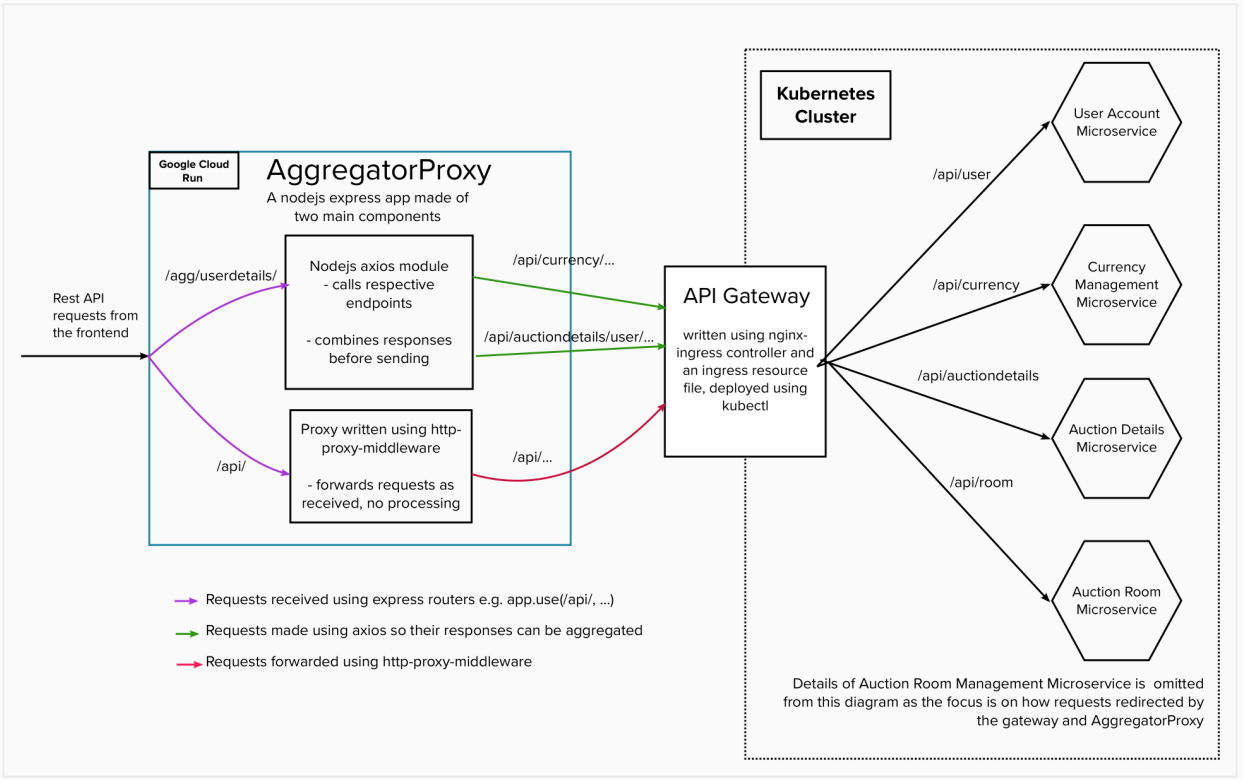
For scaling of individual microservice deployments, HPA was configured to scale up to 10 replicas. The scaling metric being used is observed CPU utilization of 80%.

6.3.2 API gateway

In microservice architecture, microservices tend to be deployed in different endpoints at different ports. When a frontend developer requests for the list of endpoints to call, their life will be made very difficult if each api call needs to call a different port in the url. There is also too much coupling and changes to the microservices on the backend can result in changes on the frontend. For these brief reasons, an abstraction needs to sit in front of the frontend and backend to redirect requests appropriately.

In our application, we have two layers sitting in between the frontend and backend, which we will term AggregatorProxy and the Api Gateway. The naming and why this architecture is as such will be made clearer as we explore the design choices later in section 6.5.4 Designing the api gateway and 'aggregateproxy'.

The Api Gateway sits at the endpoint of the kubectl cluster, while the AggregatorProxy is a Google Cloud Run application with its own externally-accessible url.



A view of how the aggregator interacts with the frontend and api gateway, with the arrows labelling which requests follow those arrows

6.3.2.1. Implementing of The Api Gateway Using Ingress

We implemented the api gateway mainly using three ingress resources and the kubernetes ingress-nginx controller.

One of the resources contain the paths that are only allowed by authenticated users, followed by the auth-url annotation that contains a url which the controller can send a get request to authenticate the user. If the controller gets a 200 OK, the request is forwarded by the controller to the appropriate microservice that matches the path. For any other response code, the request stops at this api gateway layer and a 401 Unauthenticated response code is sent to the frontend. This will allow the frontend client to redirect the user to a login page.

Excerpt below: Ingress resource for endpoints that need authentication

The ingress resource for endpoints that need no authentication to access like the login are similar, except without the auth-url annotation

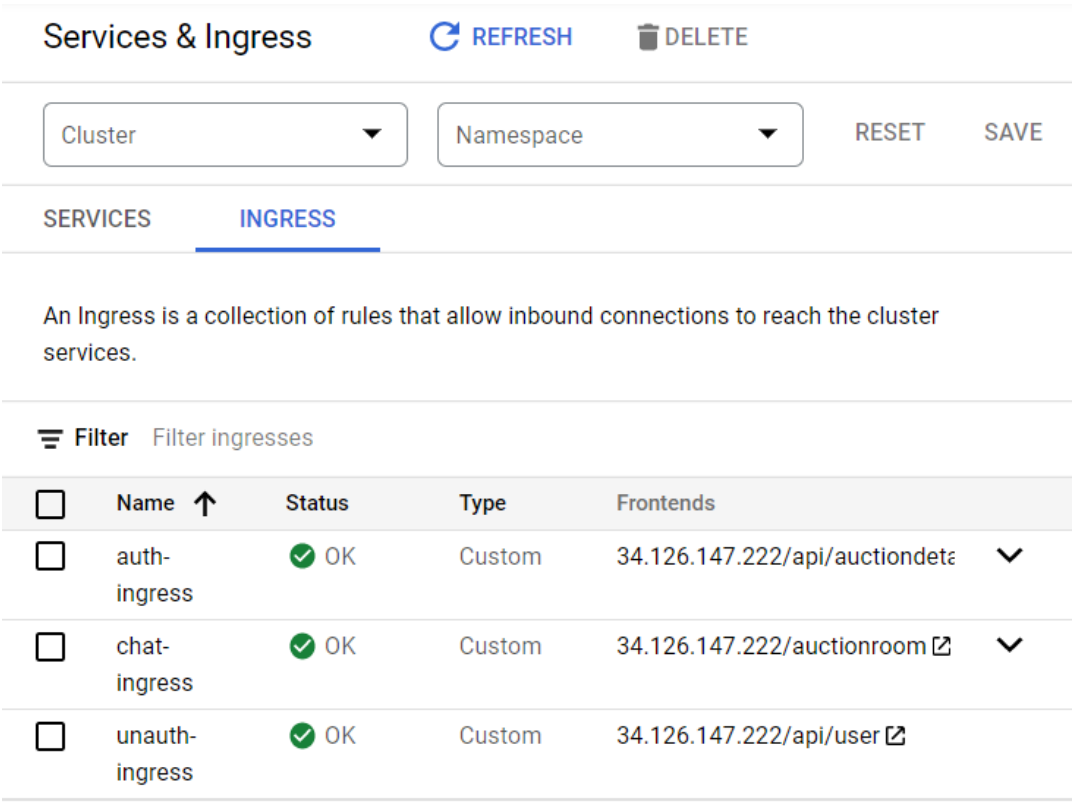
```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
 name: auth-ingress
 namespace: default
 annotations:
  kubernetes.io/ingress.class: "nginx"
  nginx.ingress.kubernetes.io/auth-url:
    http://useraccount.default.svc.cluster.local:8080/api/user/user
  # (extracted away)
  # enable cors
  nginx.ingress.kubernetes.io/enable-cors: "true"
  nginx.ingress.kubernetes.io/cors-allow-methods: "PUT, GET, POST, OPTIONS"
  nginx.ingress.kubernetes.io/cors-allow-origin: "*"
  nginx.ingress.kubernetes.io/cors-allow-credentials: "true"
spec:
 rules:
  - host: kubernetes.docker.internal
  - http:
     paths:
      - path: /api/auctiondetails
        pathType: Prefix
```

```
    backend:
      service:
        name: auctiondetails
        port:
          number: 8081
    - path: /api/currency
      pathType: Prefix
    # (extracted away, other paths are configured similarly)
```

6.3.2.2. Deploying The Api Gateway Remotely

The api gateway should be deployed on the same GKE cluster as the rest of our microservies, so that the ingress resources forming the api gateway can

- Communicate with the client (in our case, the AggregateProxy)
- Interact with the internal services in the cluster so that their respective microservice deployments can process the clients' requests.



6.3.2.3. Implementing The Aggregation And Proxy Using A Node Server

This aggregation and proxy is structured like any nodejs and express server.

For example, here's part of the code that handles proxying requests and responses that can be forwarded as received:

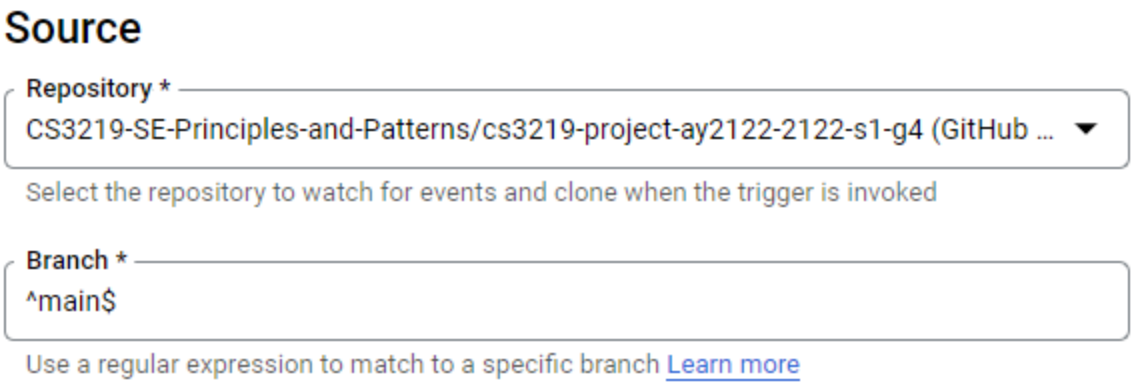
```
/**
 * Forward exactly as received. There's no processing or additional setup needed.
 * This http-proxy-middleware also helps forward the headers as received.
 * https://www.twilio.com/blog/node-js-proxy-server
*/
app.use('/api', createProxyMiddleware({
   target: FORWARDING_URL, // forwarding_url refers to the api gateway url
   changeOrigin: true,
}));
```

6.3.2.4. Deploying the AggregateProxy Remotely

The aggregate proxy was deployed as a serverless application on Google Cloud Run (see justification in earlier section). It was set up using the Google Cloud UI (web interface).

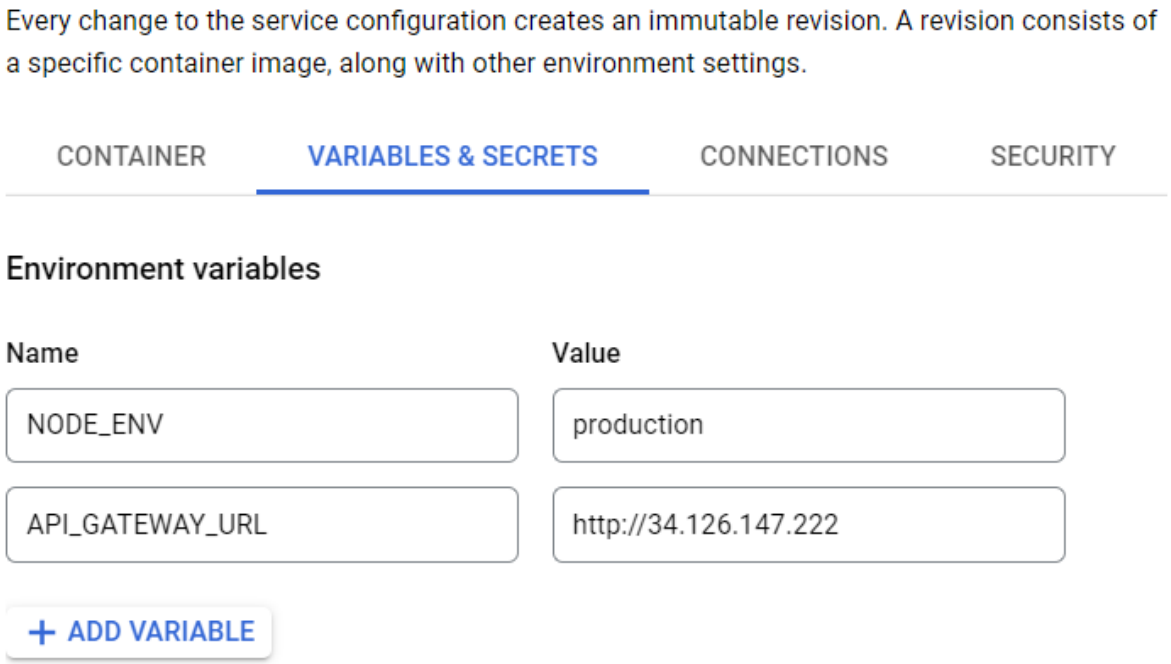
Continuous deployment is also enabled using the help of the UI and an inline cloudbuild.yaml file (inline as it is not saved on the repo, but on Google Cloud). This allows changes pushed to the repository to be seamlessly deployed.

This figure shows that the cloud run app has been configured to redeploy on any push to the main branch



The frontend sends requests to this cloud run application's url, https://cs3219-project-ay2122-2122-s1-g4-x6yyb7nmgq-as.a.run.app The path /agg/test can be used to test whether this endpoint is running.

The application is also set with some environment variables so that it can connect to the api gateway. Using environment variables helps us with easy modifications in case we ever needed to change the api gateway's url.



6.3.3 Service Discovery

Microservice instances in kubernetes have dynamic addresses and assigned network locations due to autoscaling, failures and upgrades. Therefore, there is a need for service discovery in order to connect to and utilize the microservice. Fortunately, Kubernetes has built in service discovery capabilities. A code snippet of a service in **e-auction** is shown below.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: useraccount
  name: useraccount
  namespace: default
spec:
  ports:
  - nodePort: 30198
    port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
    app: useraccount
  type: NodePort
```

In kubernetes, pods are ephemeral and the set of IP addresses pointing to them are unstable. Therefore, a kubernetes Service is created which uses labels and selectors defined in a Kubernetes deployment in order to expose pods via a stable IP address. This allows clients to access a microservice from a singular network endpoint.

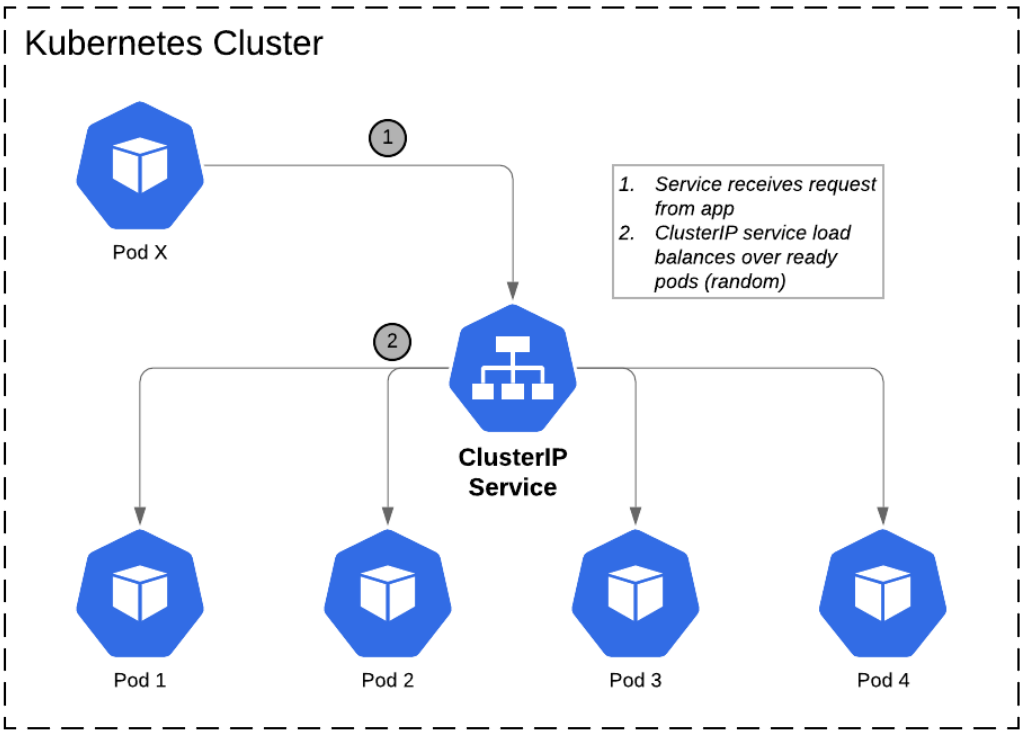


Fig 6.3.3.1 Illustration of Kubernetes Service Discovery

6.4 Microservice Design Patterns used

6.4.1 Database per service pattern

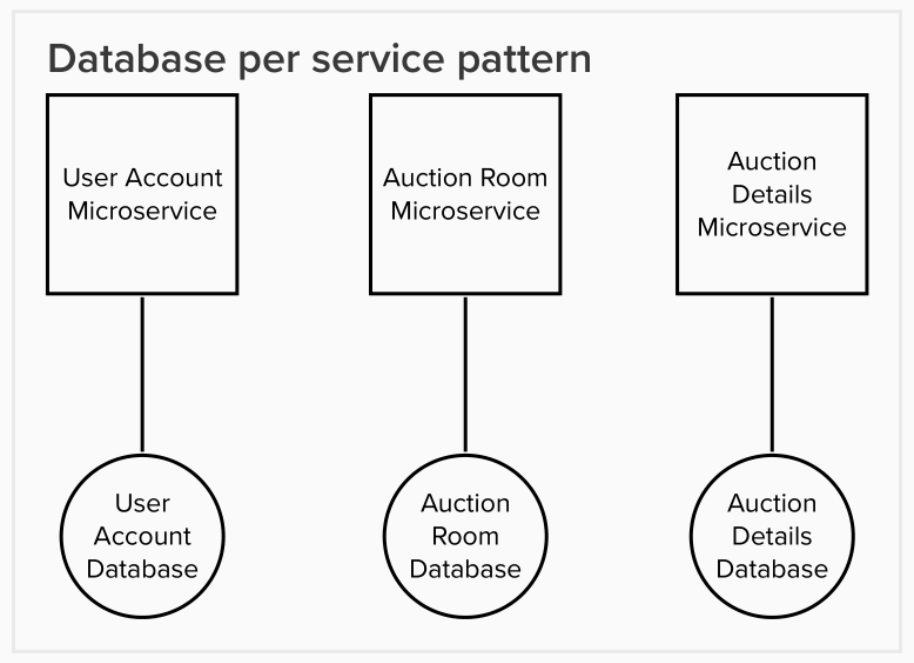


*Fig 6.4.1.1 Illustration of database per service pattern in **e-auction***
Currency Management microservice was omitted due to special design considerations

Within the microservice architecture, we implemented the database per service pattern. Aside from some design compromizations (see section 6.2 Microservices Schema for an exception made to this pattern), each microservice has its own database schema. This allows to enforce the loose coupling design of the microservice architecture such that each service can be scaled and developed independently.

6.4.2 Pub-Sub Pattern

The pub sub pattern was used to implement auction bidding and auction chat functionalities. This was done using Socket.io which establishes a websocket connection between clients and the auction room manager server. This allows for real time communication between buyers and sellers within an auction session. Refer to **auction room manager** design section for more details.

6.5 Non-trivial Design Decisions/Implementations

6.5.1 Microservice architecture vs Monolithic architecture

	Choices	**Possible benefits**	**Concerns**
1	Monolithic architecture	+ Simplicity and ease of testing, building and deployment as a whole. + Potentially more performant due to shared memory allowing for faster inter-component communication as compared to inter-service communications in microservices.	- Tight coupling of components reduces productivity over time. - Difficult and time consuming to make changes. - Developers need a good understanding of each component instead of specializing.
2	Microservices architecture	+ Independently deployable which allows for more team autonomy and easy distribution of development workload. + Independently scalable and resources can be directed specially at components with higher workload. + Easier maintenance of code as each developer is more specialized at a particular microservice.	- Debugging is more challenging due to each microservice having its own trace of logs. - Integration testing is more difficult since components are distributed and developers may be less familiar with components they are not working on. - There may be added complexity in terms of communication between different microservices.

Table 6.5.1.1 Comparison table of architecture choices

Both architectures are highly popular and commonly used, however we had to select one that we felt suited the needs for our project the most. In the end, the team decided to proceed with the microservices architecture. One of the main reasons was the time constraint of the project. Given a relatively short time span of the semester, we felt that it was more efficient if each developer could specialize and work on a few, rather than every component of the project, which would result in much technical overhead. The microservices architecture perfectly addressed this concern and allowed each developer to independently work on a microservice. Another important reason was that scaling microservices was more cost efficient than scaling a monolithic application. When scaling a monolithic application, the application is simply duplicated multiple times, even when some components have low workload. This may incur additional unwarranted costs which the microservices architecture had no

problem with since each microservice can be scaled on demand. Therefore, after much consideration, we decided on the microservice architecture.

6.5.2 Redis adapter for pub-sub scaling

Due to the scalability requirements, auction session proceedings which are managed by the **auction room manager** microservice may require each client to connect to a different microservice instance/pod. This means that synchronization is needed between all instances that serve clients within the same auction session. In order to accomplish this, a redis-cluster microservice was introduced which allows publishing of all auction-related events within a redis channel and received by all microservice instances. This prevents two clients connected to different microservice instances from not being able to receive events from each other.

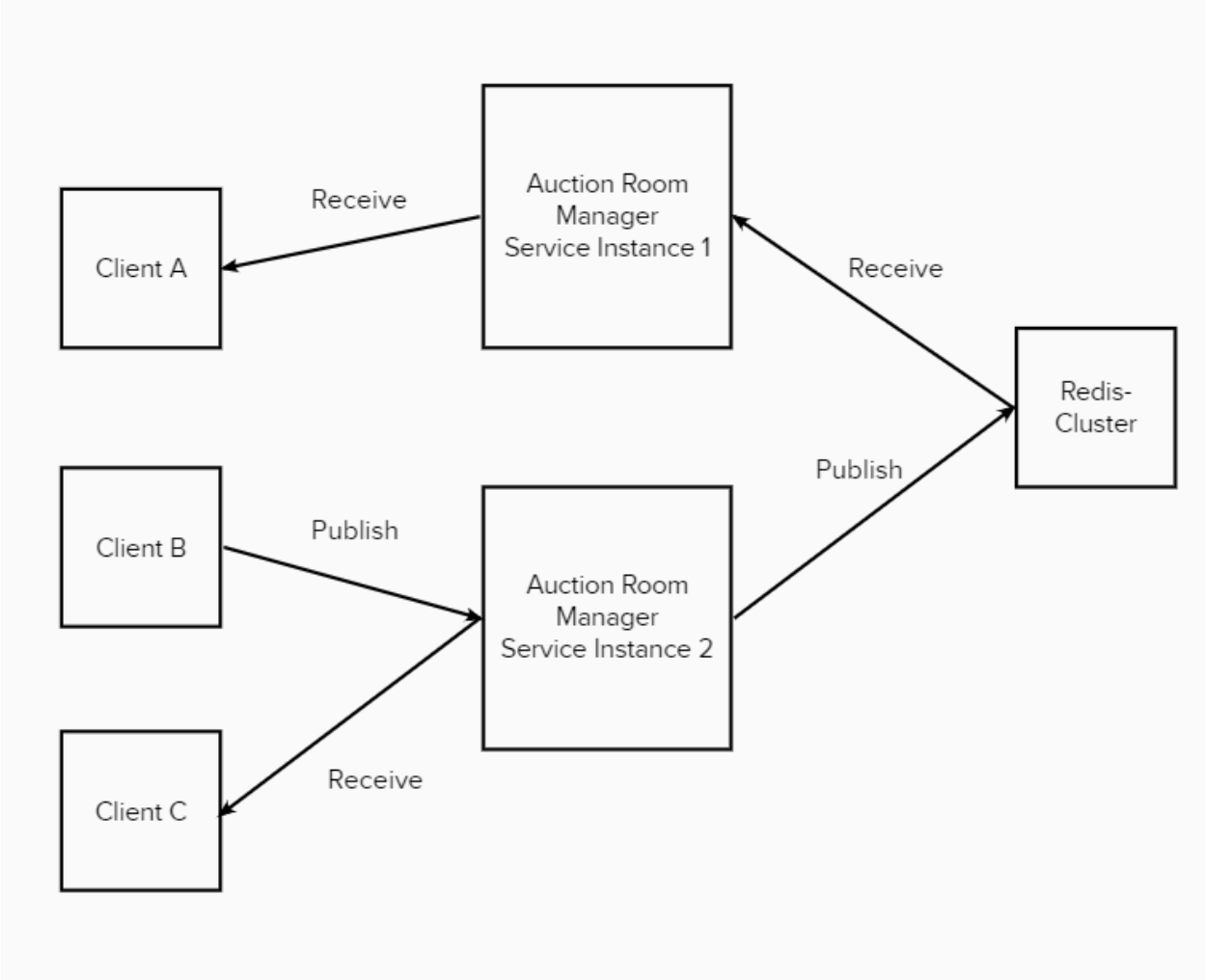


Fig 6.5.2.1 Illustration of redis-adapter purpose

This diagram illustrates the pub-sub pattern at work. Suppose client A, B and C are all participating in the same auction and have joined the session by connecting to the **auction room manager** microservice. When client B publishes an auction event, such as a bid, client C

will definitely receive the event since they are both connected to the same service instance. Note that there is another pub-sub pattern between the service instance and the client. With the redis-cluster available, service instance 1 is also notified of the event, and in turn notifies client A. This enables synchronization across all service instances when the microservice is scaled up and has many instances serving multiple clients.

6.5.3 Interservice communication between auction room and currency management microservice

In the current implementation, when an auction has ended, the frontend calls the auction room microservice to facilitate the deletion of the auction room. Before the auction room is deleted, if bids have been placed, a transaction needs to take place between the owner of the auction room and the user with the highest bid. In order for this transaction to take place, the auction room microservice needs to make a call to the API exposed by the currency management service (refer to the sequence diagram). Thus, the auction room service and currency management service need to communicate with each other.

There are two options available to facilitate communication between microservices: synchronous and asynchronous messaging. The following table compares these two different types of communication.

	Choices	**Possible benefits**	**Concerns**
1	Synchronous messaging: Service directly calls an API of another microservice using HTTP or gRPC and waits for a response.	+ Easier to implement especially when a response is required from the receiving microservice. + The sender microservice receives a response back from the receiver microservice which helps to confirm that the receiver microservice has carried out the required processing.	- Tight coupling between the communicating microservices. - If the receiver microservice fails then the sender microservice will also fail as it needs to wait for the response from the receiver.
2	Asynchronous messaging: One service sends a message and other services receive the message and carry out the necessary	+ There is lesser coupling involved as the microservice sending the message does not require any knowledge about the receiving microservice + The sender microservice only	- More complex to implement than synchronous messaging. - Need to implement an additional queue to keep track of responses if the sender

	processing.	needs to send the message once even if multiple services need to receive the message + Messages can be stored in a queue which acts like a buffer to prevent the microservice at the receiving end from being overloaded	microservice needs to receive back the response which further adds to the complexity. - Not suitable in cases where the sender microservice needs to wait for a response from the receiver microservice - If message and response queues get full then latency will be high

Table 6.5.3.1 *Comparison table for interservice communication methods*

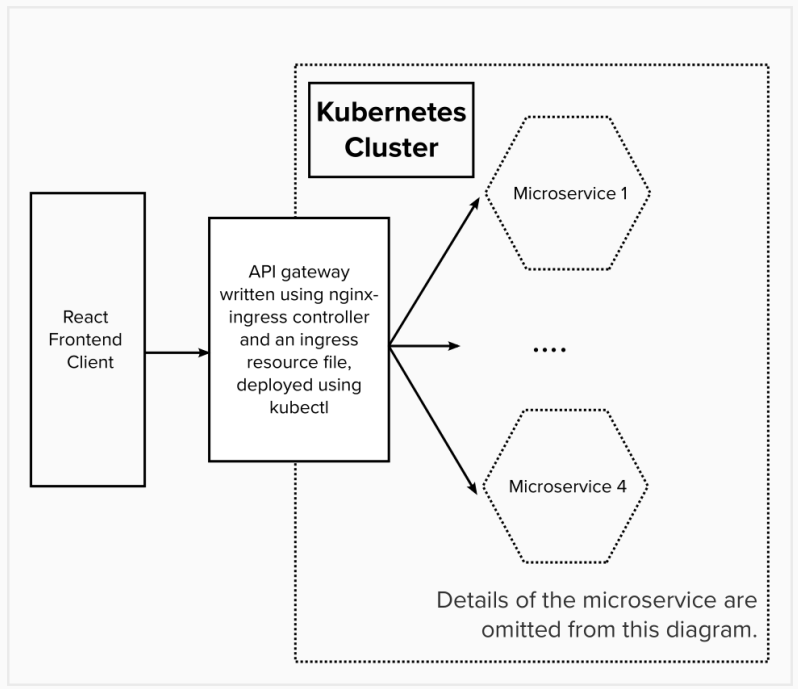
After comparing both types of communication patterns, we decided to implement the synchronous messaging pattern to facilitate communication between the auction room and currency management microservices. The main reason behind choosing synchronous messaging was that the auction room needs to wait for a response from the currency management microservice before proceeding to delete the room. The reasoning behind this is that if the currency management microservice fails then the auction room should not be deleted from the database. This is because the auctioneer's user id is stored in the auction room on the database and will be required for the transfer of money between the highest bidder and auctioneer to take place.

6.5.4 Designing The Api Gateway and AggregateProxy

6.5.4.1. Early draft of the api gateway design

Initially we intended to make one api gateway layer to redirect requests from the frontend to the various microservices. Our choices were between doing a simple api gateway layer from scratch or using an ingress resource and controller to function as our api gateway.

The ingress-nginx controller by kubernetes provided many abstractions that we thought were helpful. For example, there were stable annotations that would enable us to check if the user is authenticated at the api gateway level before redirecting the requests to the microservices. Because of how abstracted it was, we started creating an api gateway layer using an ingress resource and the ingress-nginx controller.



6.5.4.1.1 Initially planned architecture involving the api gateway

The gateway sits as the entrypoint of the kubernetes cluster and receives requests from the frontend, and forwards it to the appropriate microservice.

6.5.4.2. A new need for an aggregator

Meanwhile, the frontend was also taking shape. While implementing the user profiles frontend page, we realised there were multiple microservices that held information pertaining to one user. This meant that the frontend would be making multiple calls to various endpoints. This poses a lot of inconvenience for the frontend team so we decided to aggregate all the backend information about the user on the server side.

Here's a timeline of events that take place when the user profiles page needs user details, to explain the aggregator's role:

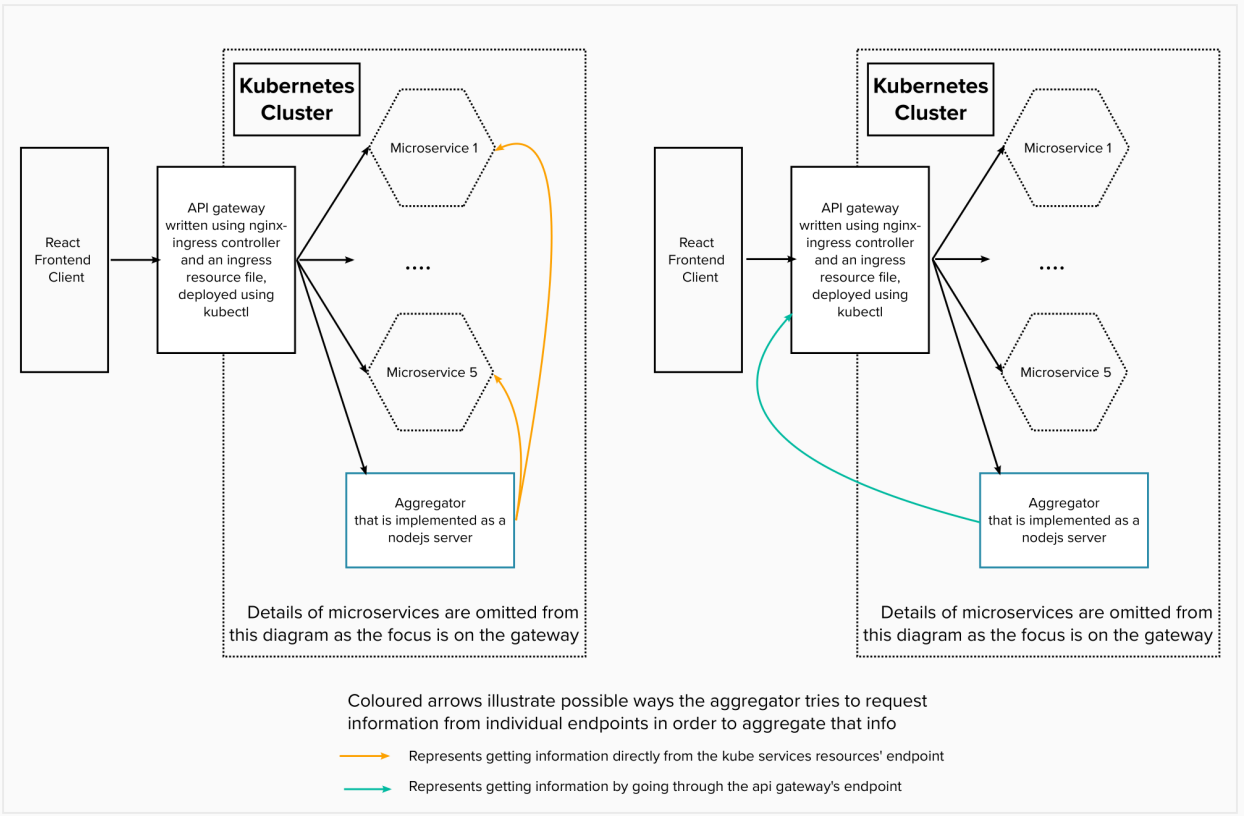
1. User opens the user profiles page
2. Frontend calls an endpoint that is responded to by the aggregator, with a userid parameter
3. The aggregator receives the frontend's request and gets information from multiple microservices that hold user information.
4. After receiving a response from each microservice, the aggregator will process the information into a json object with convenient properties for the frontend and respond to the frontend with this object.

In summary, the aggregator is the abstraction of collecting various user details from different microservices.

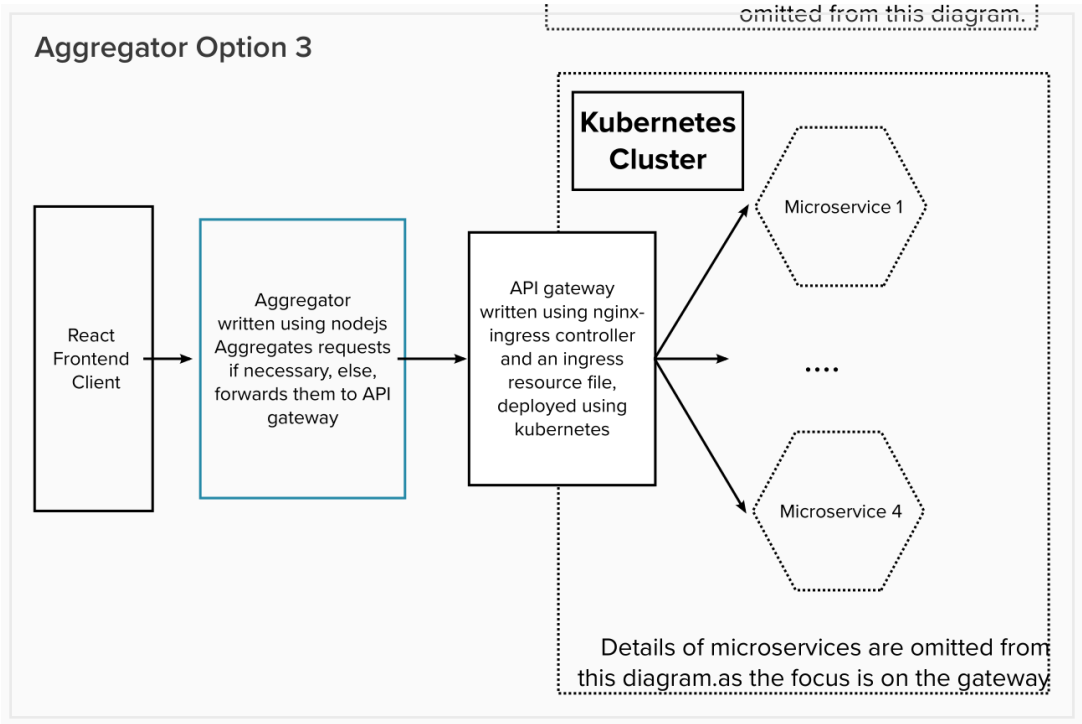
6.5.4.3. Evaluating how to tweak our design

Here are the choices our group came up with and some of the considerations. For options 2 and 3, to better visualise, we've included some diagrams below.

	Choices	**Possible benefits**	**Concerns**
1	Integrating some aggregation logic into an api gateway such that only one layer sits in between the frontend and the deployed cluster for the backend	+ With one sole layer, there are less network hops so faster response times. + Less configuration and testing to do when both aggregation and request redirection are integrated in the same layer.	- There seemed like there was little documentation except for specific use cases so it could have been difficult to find the relevant lua code to customize it further for our project. We wanted to avoid this causing future problems when we have to extend our code. - Lua was not a scripting language everyone was familiar with so it might be harder for teammates to review code and customise it after it was first written
2	A nodejs server that's like a microservice, used to aggregate calls, sits behind the API gateway (See diagram below)	+ Many sources did use the example of placing an aggregator behind the api gateway + The api gateway is still the layer facing the frontend, so the inbuilt features of ingress-nginx such as the authentication annotations can be used first	- This presents an extra network hop between the frontend and the microservices.
3	An aggregation service that sits in front of the api gateway (so the frontend communicated with it first) (See diagram below)	+ Can authenticate that the api gateway is only being communicated from this + On the frontend, the address of the api gateway won't be exposed so easily. This could help give us an extra layer of security	- This also causes an extra network hop - Not a lot of sources talked about having an extra server sitting in front of the api gateway



6.5.4.3.1 The above figure illustrates two ways of implementing option 2.



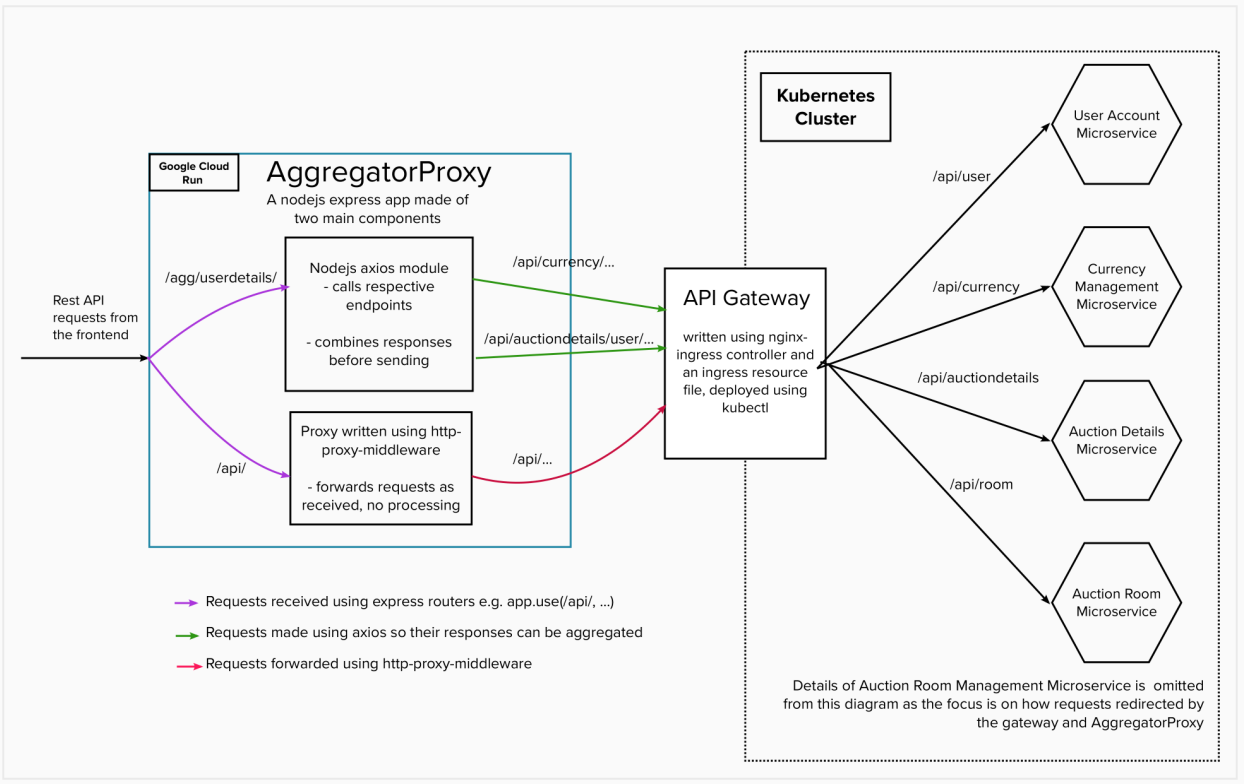
6.5.4.3.2 This figure illustrates where the aggregator sits in option 3.

The benefits of option 2 and option 3 did not seem to outweigh each other so they seemed like equivalent picks to try. There weren't many sources that discouraged us from trying one option over the other. As such we chose option 3, the aggregation service that sat in front of the api gateway.

6.5.4.4. Implementation process and rationale

After choosing this architecture, we had to decide how to implement an aggregator and proxy using a nodejs server. We did not want to rewrite existing abstractions, as such, we made use of existing nodejs modules such as http-proxy, http-proxy-middleware to configure how to forward the requests and we also used axios to query our api gateway from this aggregator server.

These choices resulted in the following implementation of the AggregatorProxy as introduced earlier in the features section, under 6.3.2 API gateway.



6.5.4.4.1 The result of our design decisions

6.6 Microservices Implementation Details

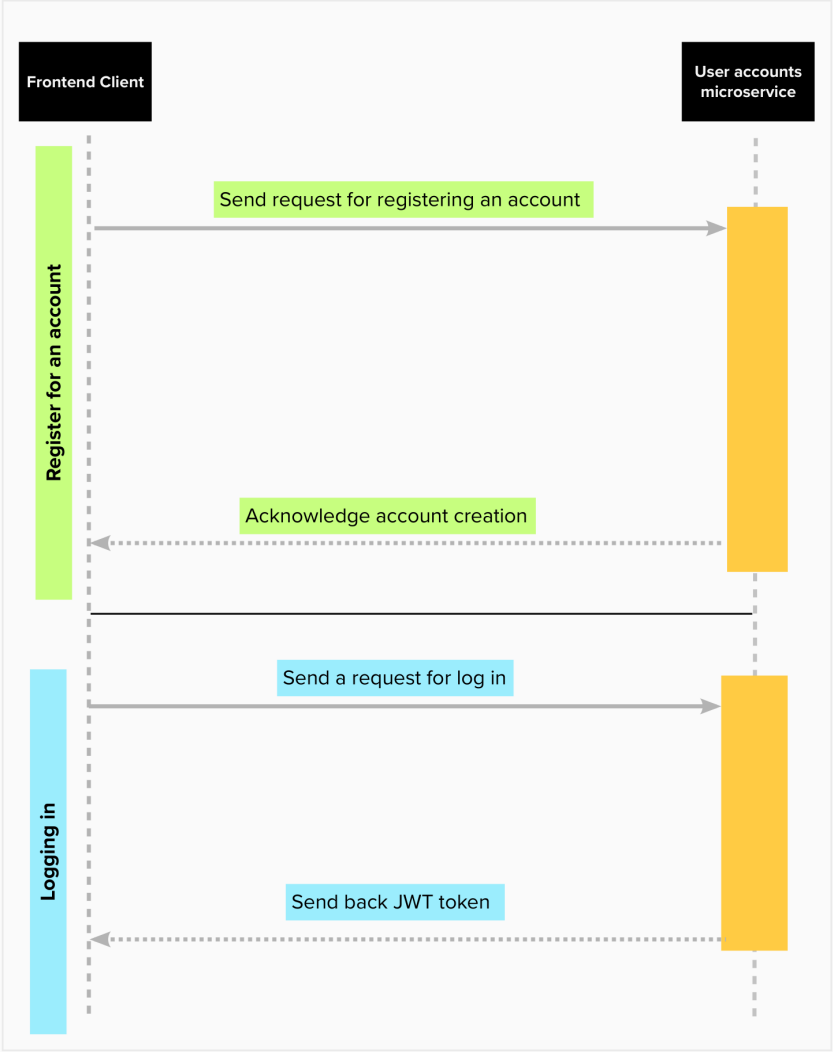
This section describes some noteworthy details on how certain features are implemented.

6.6.1 User Accounts

The user accounts microservice contains endpoints for users to sign up for an account, login to their account and for validation of JWT tokens in request objects.

JWT token

The following sequence diagram shows how the sign up and login process works:



6.6.1 Sequence diagram depicting Registration and Log in

As can be seen from the sequence diagram, once the user has successfully registered for an account and logged in, the backend will send a JWT token in the response. This JWT token is stored in the browser's Local Storage by the frontend. The JWT token must be sent along with all requests from the frontend which require authentication.

The user accounts microservice also provides an endpoint for validating a request object by checking if the JWT token is valid and whether it contains a user id. The endpoint is also used by the api gateway as the auth-url annotation in the ingress resources (more details in section Implementing The Api Gateway Using Ingress).

6.6.2 Auction room

The auction room microservice has the following duties:

- Creation of auction rooms
- Deletion of auction rooms
- Storing of bids placed by users in firebase realtime database
- Retrieval of the user with the current highest bid

Deletion of auction rooms

Before an auction room is deleted a synchronous call is made to the API exposed by the currency management microservice to facilitate the transfer of currency from the highest bidder to the auctioneer once the auction has ended. Implementation details for the communication between the two microservices have been stated under section 6.5.3 . The following sequence diagram depicts how the auction room deletion process works:

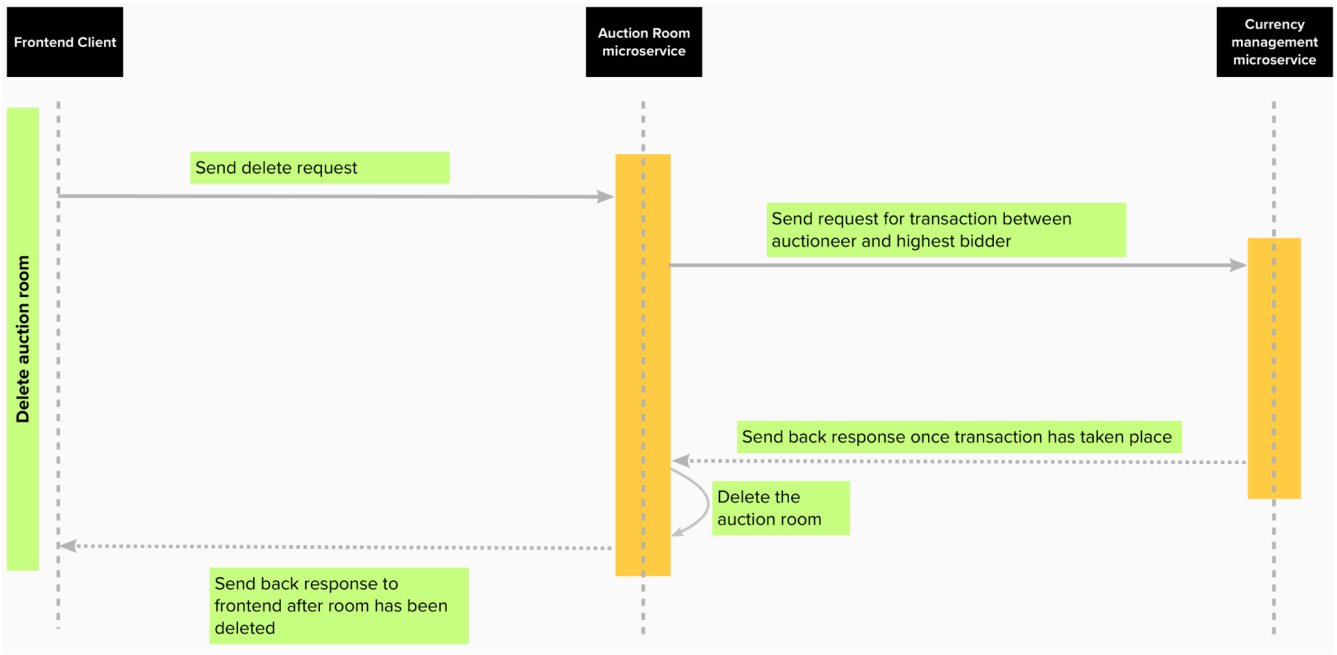


Fig 6.6.2 Sequence diagram demonstrating deletion of auction room

6.6.3 Currency Management

The currency management service handles transactions between the auctioneer and highest bidder once the auction has ended. Other features include topping up a user's account when the user requests for more currency and retrieving information about the amount of currency held by a particular user.

MongoDB Transaction API

An important feature of the currency management microservice is the transfer of currency from the highest bidder (sender) to the auctioneer (receiver). The MongoDB transaction API has been used to implement this.

In MongoDB, transactions allow for atomic updates of multiple documents. What this means is that although addition of currency to the receiver's account and deduction from the sender's account are two separate operations they can be treated as one. For example, if currency is deducted from the sender's account but the transaction fails halfway before currency has been credited to the receiver's account, the database will rollback the deduction of currency from the sender's account as well. This prevents any inconsistencies from occurring between the amount of currency held by the sender and receiver.

Another benefit of using MongoDB's transaction API is that it automatically handles transient errors that might occur during the course of the database transaction. Transient errors include difficulty in connecting to the database or temporary unavailability of a resource. These errors can usually be fixed

by retrying the database transaction. When the `withTransaction` function is used, if a transaction fails due to a transient error then MongoDB will retry the transaction automatically and update the data in the database thus ensuring durability.

A code snippet depicting the implementation of the database transaction has been given below:

```
router.post("/transaction", async (req, res) => {
        const session = await User.startSession();
        await session
                .withTransaction(async () => {
                        await User.updateOne(
                                { username: req.body.sender },
                                { $inc: { currency: -req.body.currency }}
                        );
                        await User.updateOne(
                                { username: req.body.receiver },
                                { $inc: { currency: req.body.currency }}
                        );
                })
                .then(() => {
                        res.status(200).json({ message: "Transaction completed" });
                })
                .catch((err) => {
                        res.status(500).send(err);
                });
        session.endSession();
});
```

6.6.4 Auction details

The auction details microservice has the following duties:

1. Creating, reading, updating and deleting of auctiondetail records (CRUD operations)
2. Providing endpoints that provide all the auctiondetails related to a user, for the sake of the user profile page
3. Providing a variety of endpoints from which searched and filtered auction details can be obtained, for the sake of functional requirements F2.1.2 and F2.1.3.

Although the entire list of auction details can be sent to the frontend and then sorted or filtered, we felt that it was good to keep that logic on the backend. Sending too large of a list of auction details could also

cause greater network delays or error rates. This was why we came up with the endpoints in the third bullet point.

6.6.5 Auction room manager

The auction room manager microservice enables real time communication between all users connected to an auction session.

Each auction session or instance is represented by a communication channel, which is uniquely identified by an auction id generated by the **auction details microservice**. This allows the application to safely create auction sessions without concern regarding overlapping communication channels.

Socket.io Client

On the client side, clients can join and leave channels to subscribe or unsubscribe to a given channel. By joining and subscribing to a channel, clients are notified of any bid or chat events sent by other users within the same channel. This is analogous to a real life auction where people who participate in an auction are in the same room and therefore can hear others make bids or make comments. Clients can also act as publishers and are able to broadcast bid and message events to other clients subscribed to the same channel. A code snippet of the frontend client implementation is shown below.

```
socketRef.current = socketIOClient(SOCKET_SERVER_URL, {
   extraHeaders: {
      Authorization: JSON.parse(localStorage.getItem('user'))
   },
   transport: ['websocket'],
   query: { roomid: roomId, token: JSON.parse(localStorage.getItem('user')) }

});
```

Socket.io Server

On the server side, the server maintains a *room* which is an arbitrary channel that client sockets can join and leave. This channel is used to broadcast events to a subset of clients who have joined and subscribed to that particular channel. Upon receiving a new event from a client, this event is then broadcasted or published to all relevant clients. For instance, when a chat event is sent to the server, all clients subscribed to the channel are notified and will receive the chat message to be displayed on the frontend. A code snippet of the microservice server implementation is shown below.

```
socket.on(NEW_CHAT_MESSAGE_EVENT, async (data) => {
   rateLimiterRedis.consume(socket.handshake.address)
      .then((rateLimiterRes) => {
```

```
        console.log(data);
        if (/\S/.test(data['body'])) {
          nsp.in(roomId).emit(NEW_CHAT_MESSAGE_EVENT, data);
        }
      })
      .catch((rejRes) => {
        if (rejRes instanceof Error) {
        } else {
          const secs = Math.round(rejRes.msBeforeNext / 1000) || 1;
          socket.emit(NEW_CHAT_MESSAGE_EVENT, { body: "Please do not spam the chat!", username:
"'Room Admin"' });
        }
      });
});
```

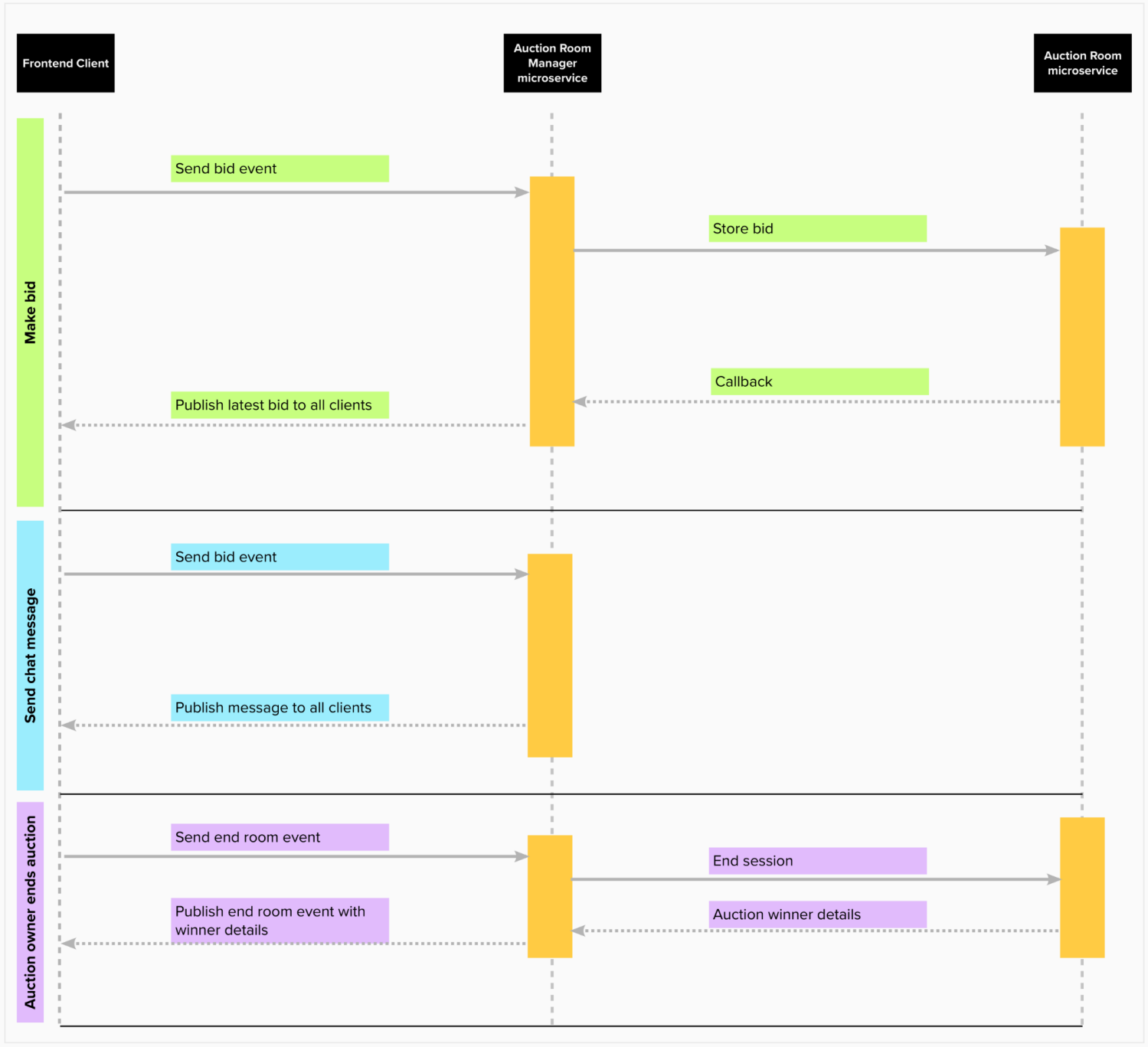


Fig 6.6.5 Sequence diagram demonstrating handling of new events

6.6.5.1 Making a bid

When the client makes a new valid bid, the client will send a new bid event to the **auction room manager** microservice. The **auction room manager** microservice then makes a call to **auction room** which will store the bid as the new highest bid. **Auction room manager** will then publish the bid event to all clients to notify each client of the highest bid.

6.6.5.2 Sending a chat message

When the client sends a chat message, the client will send a new chat event to the **auction room manager** microservice. The **auction room manager** microservice then publishes the chat event to all clients to notify each client of the new chat message.

6.6.5.3 Auction owner ends auction session

When the auction owner ends the auction, the client will send a new end auction event to the **auction room manager** microservice. The **auction room manager** microservice then makes a call to **auction room** which will in turn call the **currency management** microservice to process the transaction between the highest bidder and the auction owner which is omitted from the sequence diagram. **Auction room manager** will then publish the end auction event to all clients to notify each client of the auction winner and session ending.

6.6.5.4 Rate limiting user initiated events

We also recognized the need to prevent users from making too many bids or comments within a short period of time. As mentioned in the functional requirements, a rate limiter was implemented to limit user events within a time interval.

```
rateLimiterRedis.consume(socket.handshake.address)
  .then((rateLimiterRes) => {
    console.log(data);
    if (/\S/.test(data['body'])) {
      nsp.in(roomId).emit(NEW_CHAT_MESSAGE_EVENT, data);
    }
  })
  .catch((rejRes) => {
    if (rejRes instanceof Error) {
```

```
    } else {
      const secs = Math.round(rejRes.msBeforeNext / 1000) || 1;
      socket.emit(NEW_CHAT_MESSAGE_EVENT, { body: "Please do not spam the chat!", username:
'"Room Admin"' });
    }
  });
```

This is implemented using a point system. Users are assigned points which are refreshed every time interval. Every time a user publishes or sends an event, whether chat or bid event, a point is consumed. If all points are consumed before the time where points are refreshed, the user's IP address will be blocked from publishing events for a time interval.

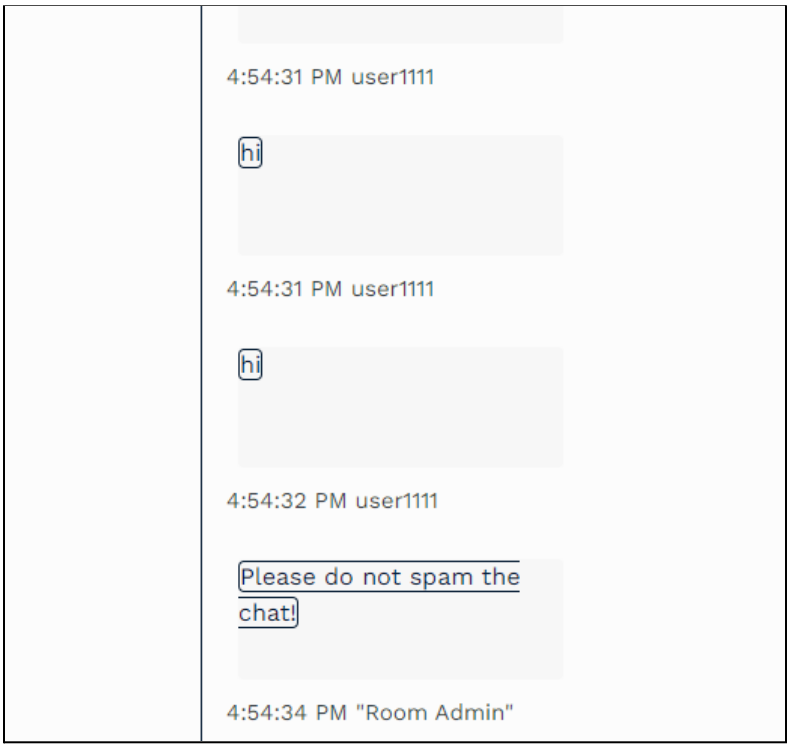


Fig 4.8.5.3 Illustration of rate limitation at work

6.6.5.5 Sticky session load balancing

Due to scalability requirements, it is a possibility that there are multiple server instances available to serve clients. However, due to the way Socket.io works using HTTP long-polling, all requests need to reach the process or server instance that originated them. The most efficient way to do this would be via sticky session load balancing based on cookies. In our case, this

could easily be done via our ingress-nginx API gateway by adding the relevant annotations as shown below.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
 name: auth-ingress
 namespace: default
 annotations:
  kubernetes.io/ingress.class: "nginx"
  nginx.ingress.kubernetes.io/auth-url: http://useraccount.default.svc.cluster.local:8080/api/user/user
  # enable cors
  nginx.ingress.kubernetes.io/enable-cors: "true"
  nginx.ingress.kubernetes.io/cors-allow-methods: "PUT, GET, POST, OPTIONS"
  nginx.ingress.kubernetes.io/cors-allow-origin: "*"
  nginx.ingress.kubernetes.io/cors-allow-credentials: "true"
  # sticky session to enable chatroom scaling
  nginx.ingress.kubernetes.io/affinity: "cookie"
  nginx.ingress.kubernetes.io/affinity-mode: "balanced"
  nginx.ingress.kubernetes.io/session-cookie-name: "route"
  nginx.ingress.kubernetes.io/session-cookie-expires: "172800"
  nginx.ingress.kubernetes.io/session-cookie-max-age: "172800"
```

7. Frontend

7.1 Architecture

React is used as the framework for the front-end development of the system. It maintains a virtual Domain Object Model (DOM) and allows for building of encapsulated components that manage their own state.

Material UI is used as the user interface library as it allows for building of the design system and faster development of the React application.
The diagram below offers an overview of the components in the front-end.

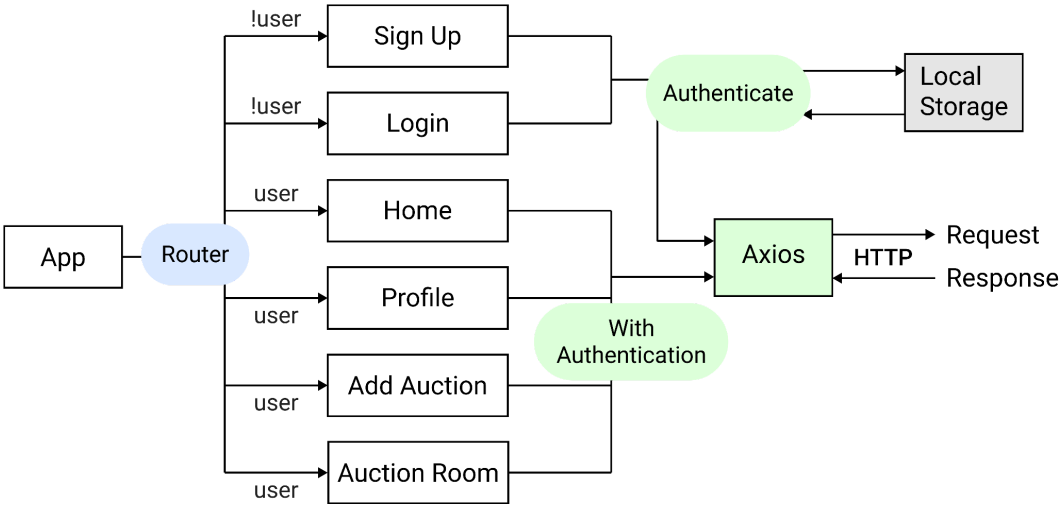


Figure 7.1.1 Overview of frontend component with React and Axios

The App route to the 6 pages where users that are not authenticated (signed up or logged in) will have to do so before being able to view and perform operations on the other pages (Home, Profile, Add Auction and Auction Room). Upon signing up or logging in, users will be directed to the Home page and their JWT token will be stored in the local storage. The JWT token is then used when making calls to the backend through the use of axios.

The following sections 7.2 to 7.7 shows screenshots of potential actions by users on the various pages of the platform.

7.2 User Authentication

Users without an account can sign up with a username, email and password.

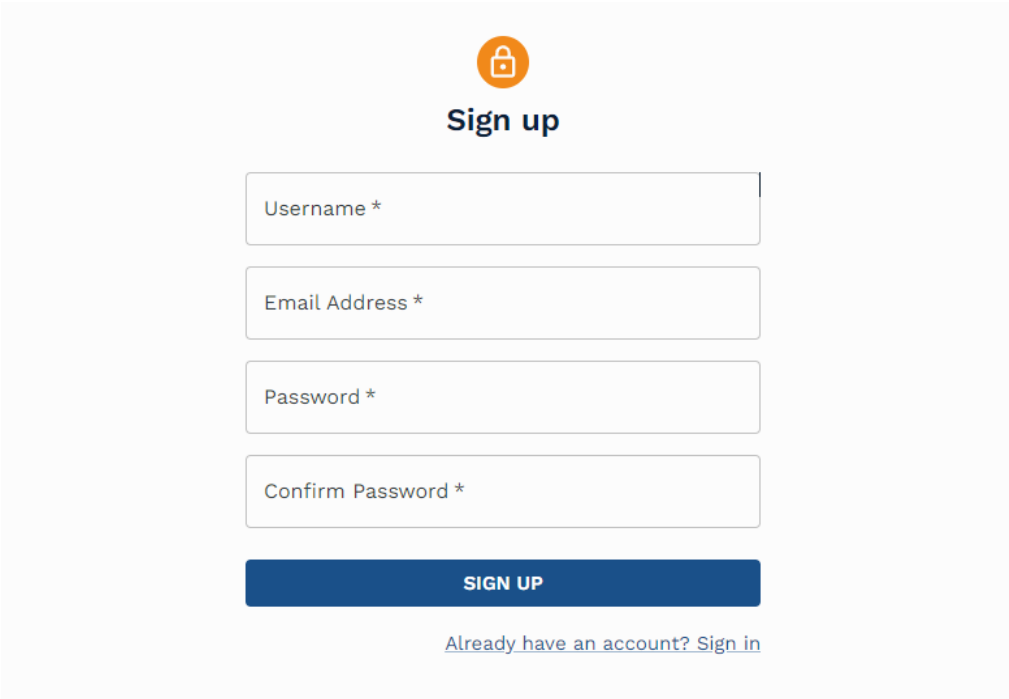


Fig 7.2.1 Signup Page

The front-end performs another layer of validation checks before sending the request to the backend microservice, including the checking of the email address to ensure that it ends with “u.nus.edu”, and the matching of passwords.

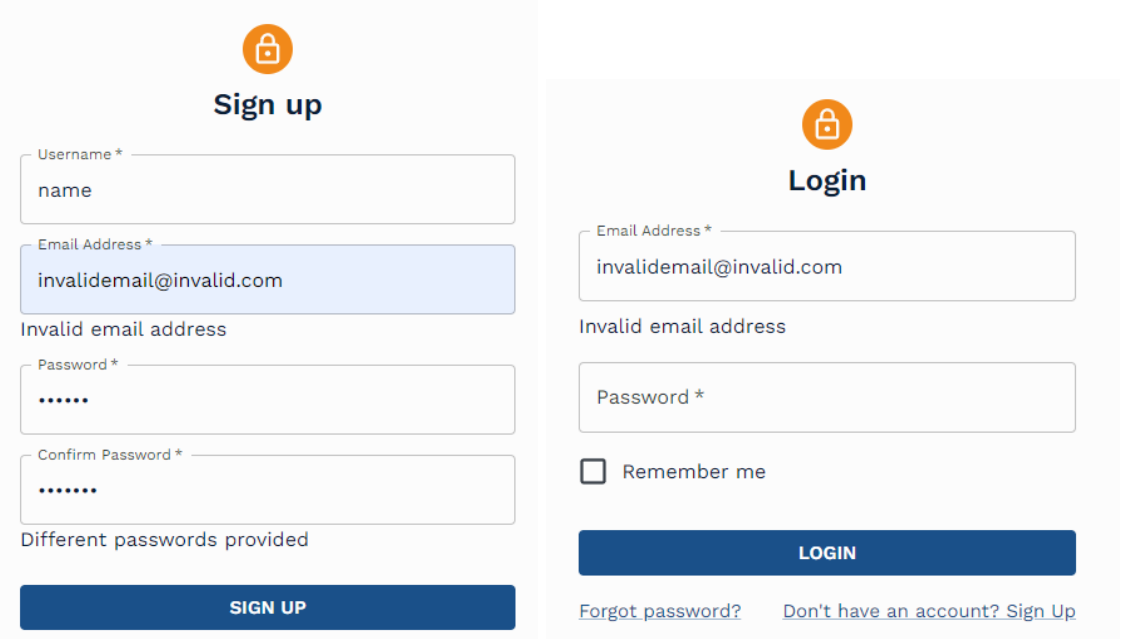


Fig 7.2.2 Validation Examples on Sign up and Login Pages

Users with an account can log in using their email address and password.

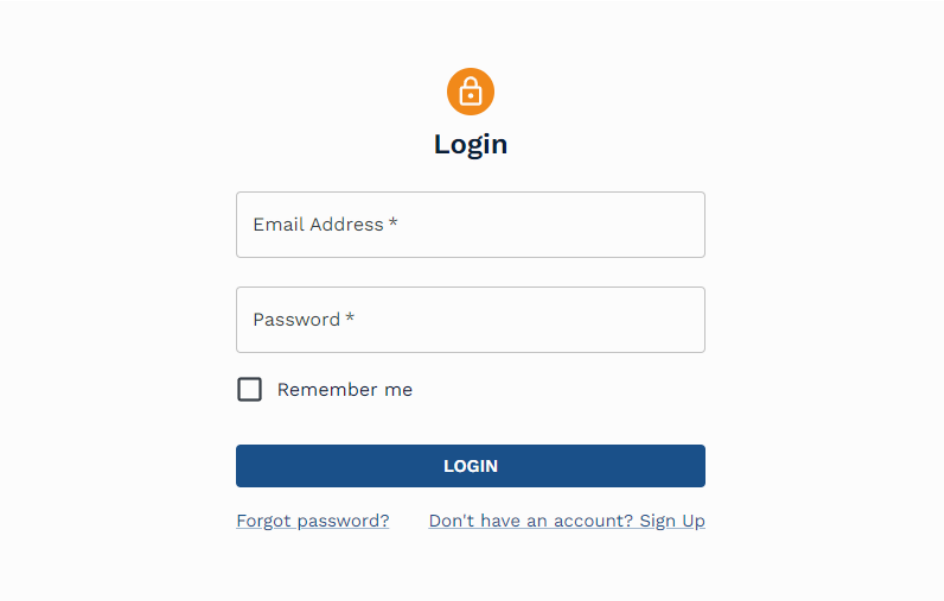


Fig 7.2.3 Login Page

7.3 Auction Viewing

Upon logging in, users will be redirected to the home page where they can view all the ongoing auctions.

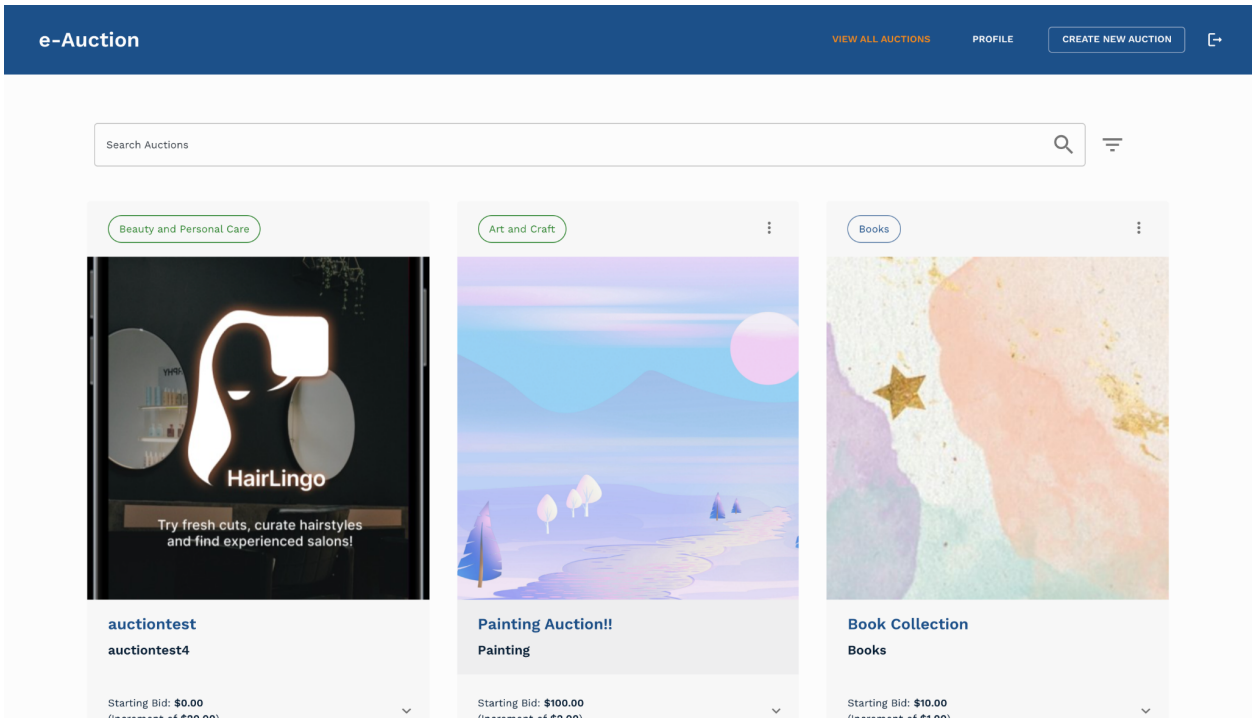


Fig 7.3.1 Example of homepage auction viewing (With ongoing or future auctions)

Users can search for auctions by auction name, as well as filter for auctions based on their item name, starting bid range and category.

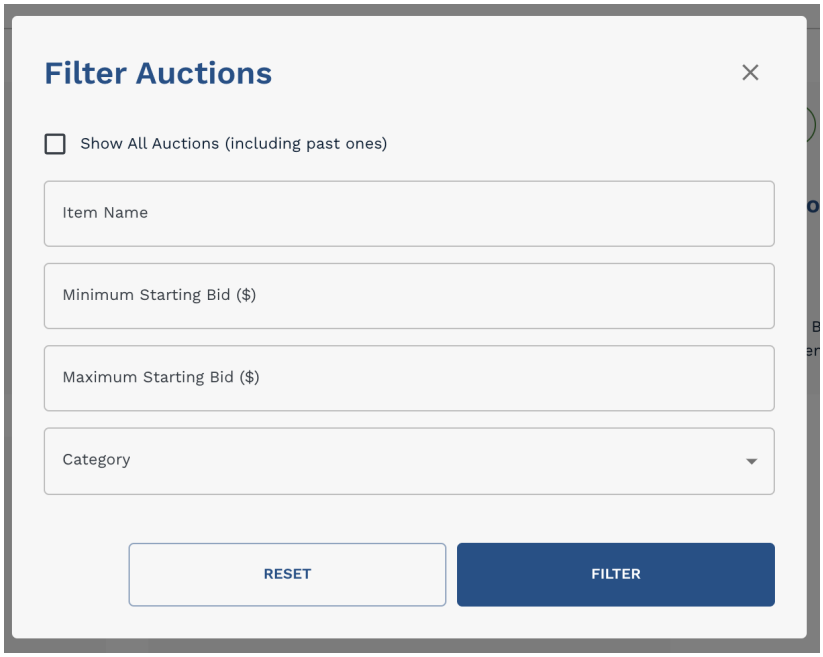


Fig 7.3.2(a) Demonstration of auction filtering by category and price (Filter Dialog)

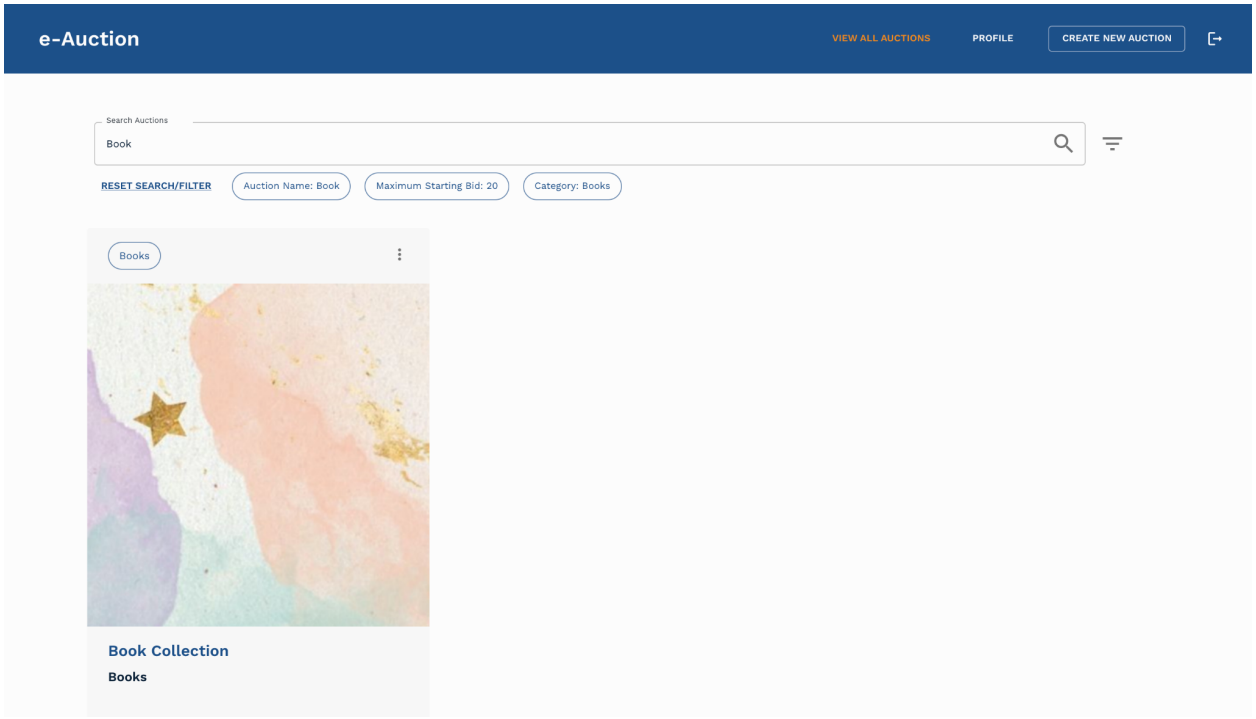


Fig 7.3.2(b) Demonstration of auction filtering by category and price (After Filtering)

Users can be directed to the auction room by clicking on the auction listed. If the auction has yet to start or has ended, users will be prompted accordingly.

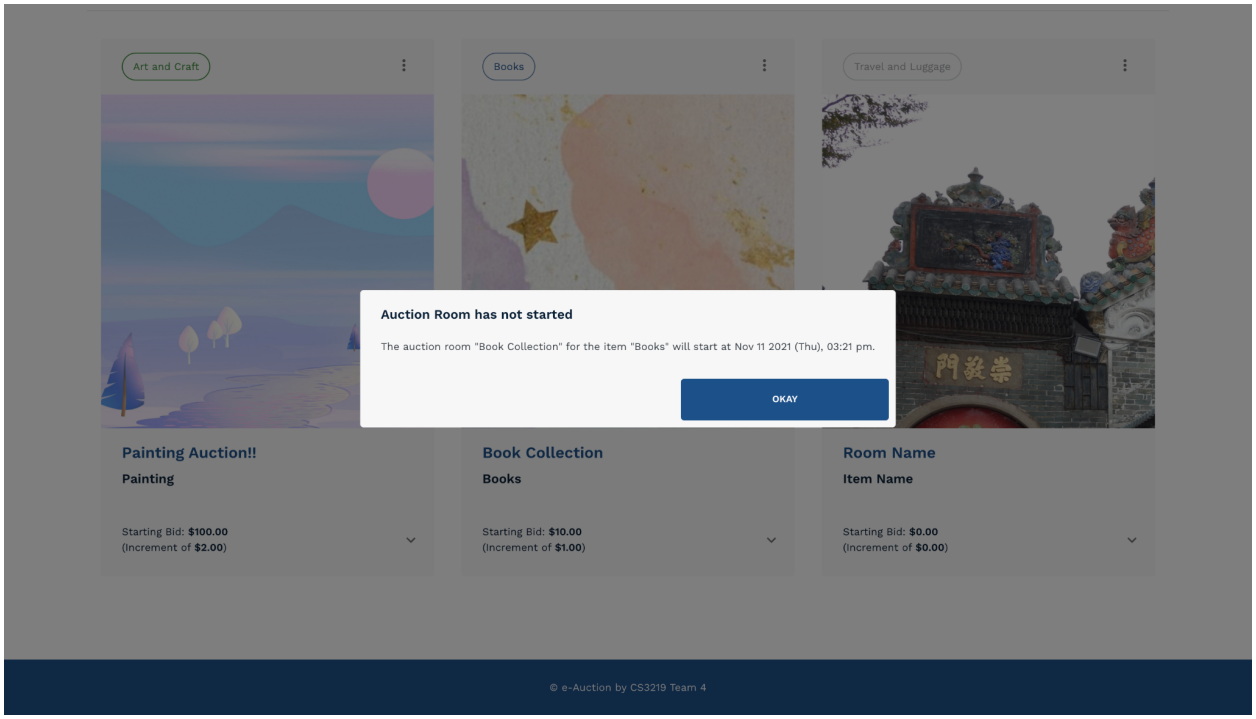


Fig 7.3.3(a) Demonstration of attempt to click on auction has yet to start

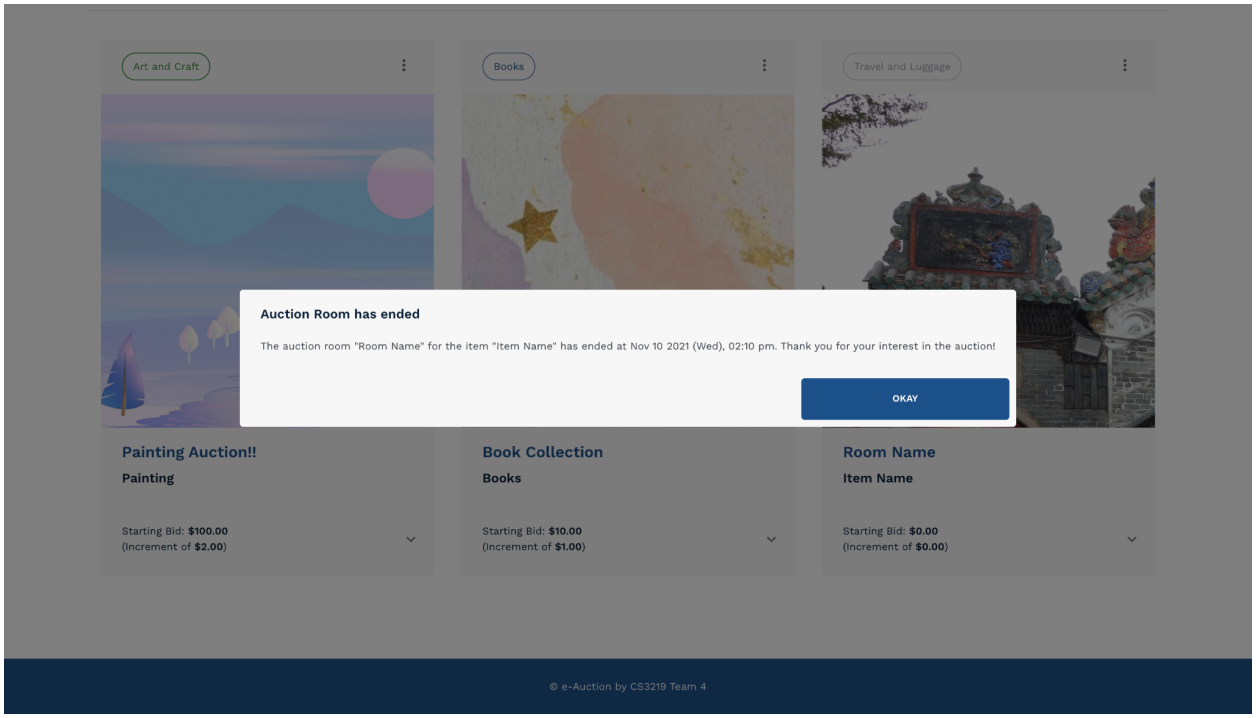


Fig 7.3.3(b) Demonstration of attempt to click on auction that has ended

7.4 Account and Currency Management

From the Profile page, users can view their username, email and currency.

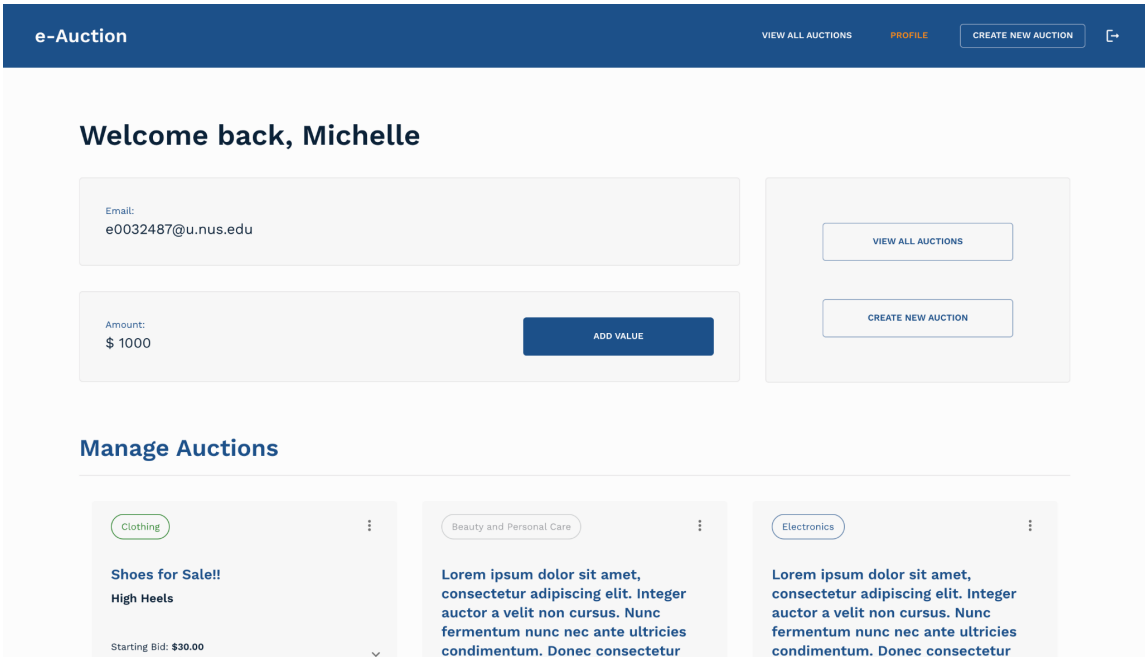


Fig 7.4.1 Example of Profile account viewing

From the Profile page, users can click to top up the amount of currency in their account.

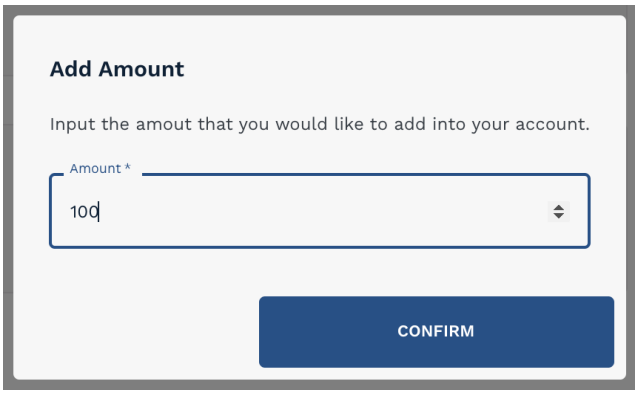


Fig 7.4.2(a) Demonstration of currency top-up (Add Amount Dialog)

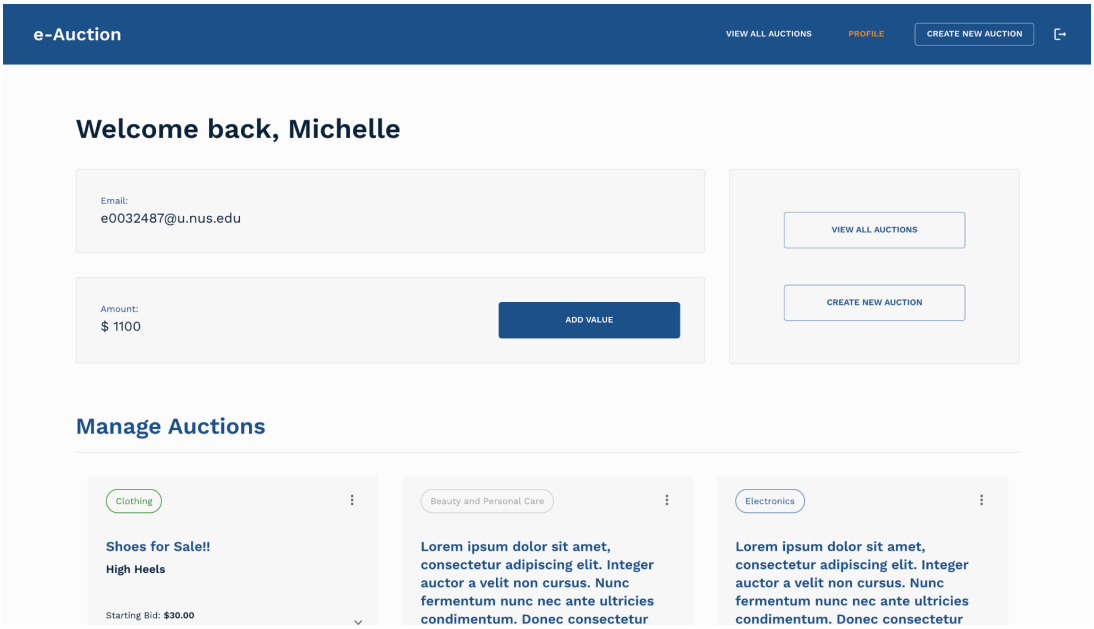


Fig 7.4.2(b) Demonstration of currency top-up (After Top Up)

7.5 Personal Auction Management

Users can create a new auction from the Create Auction Page, by providing the compulsory fields, as indicated by the asterisk (*). Validation of the required fields are performed.

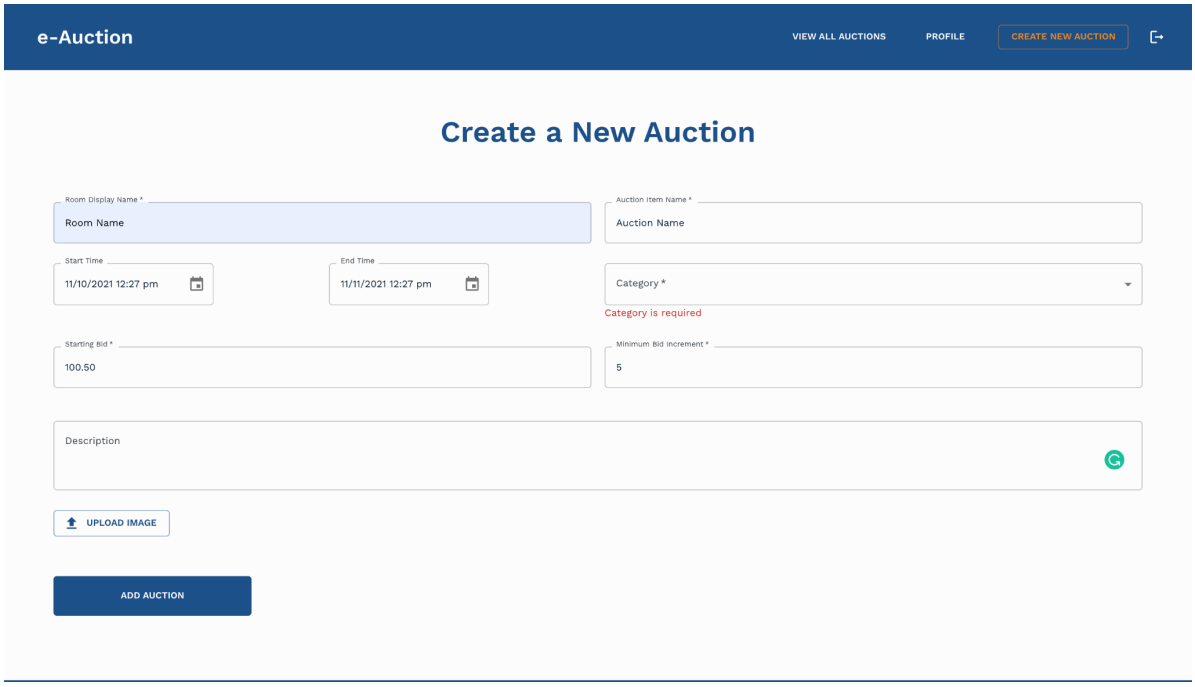


Fig 7.5.1 Demonstration of auction creation (with validation)

From the Profile page, users can view the list of auctions created, including all past auctions.

From the Home page or Profile page, users can delete an auction that they had created if the auction is not ongoing. If an auction is ongoing, users can end the auction in the Auction Room Page, shown in Section 7.6.

Fig 7.5.2(a) Demonstration of auction deletion (Not yet Started/Ended vs Ongoing)

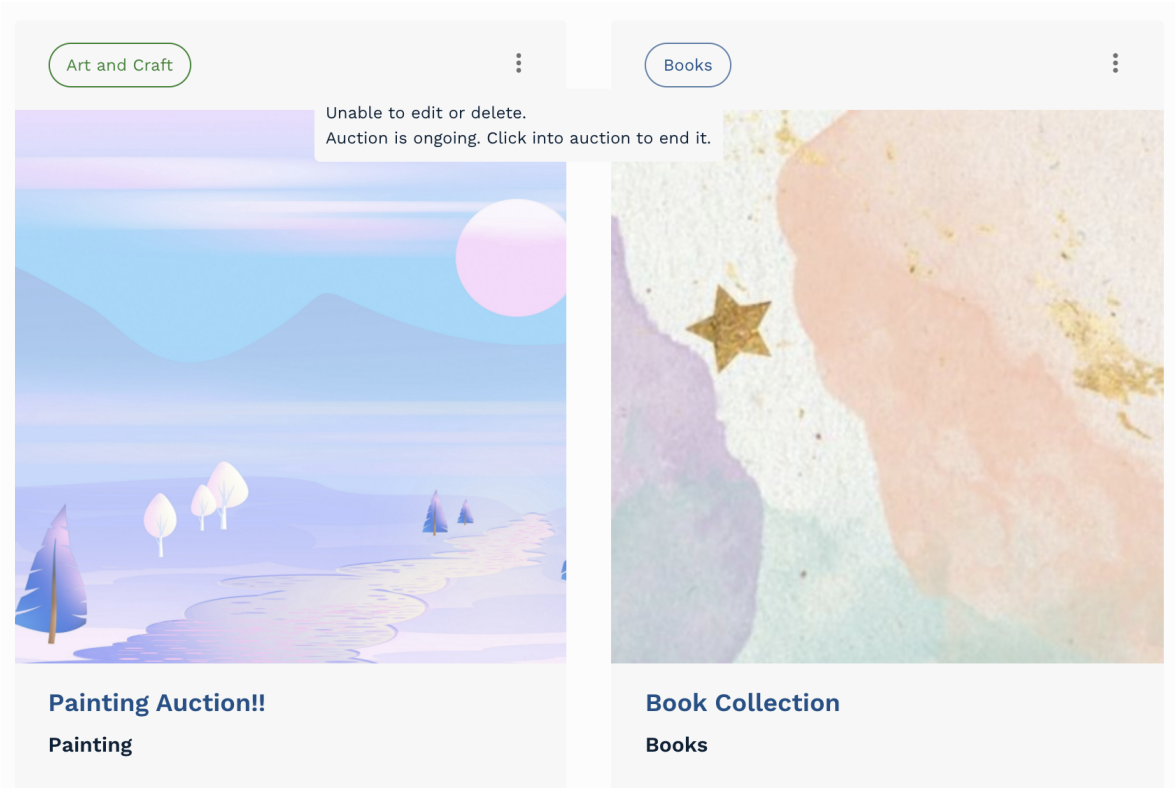


Fig 7.5.2(b) Demonstration of auction deletion (Prompt unable to delete for Ongoing auctions)

From the Home page or Profile page, users can edit the auction that they had created if the auction is not ongoing.

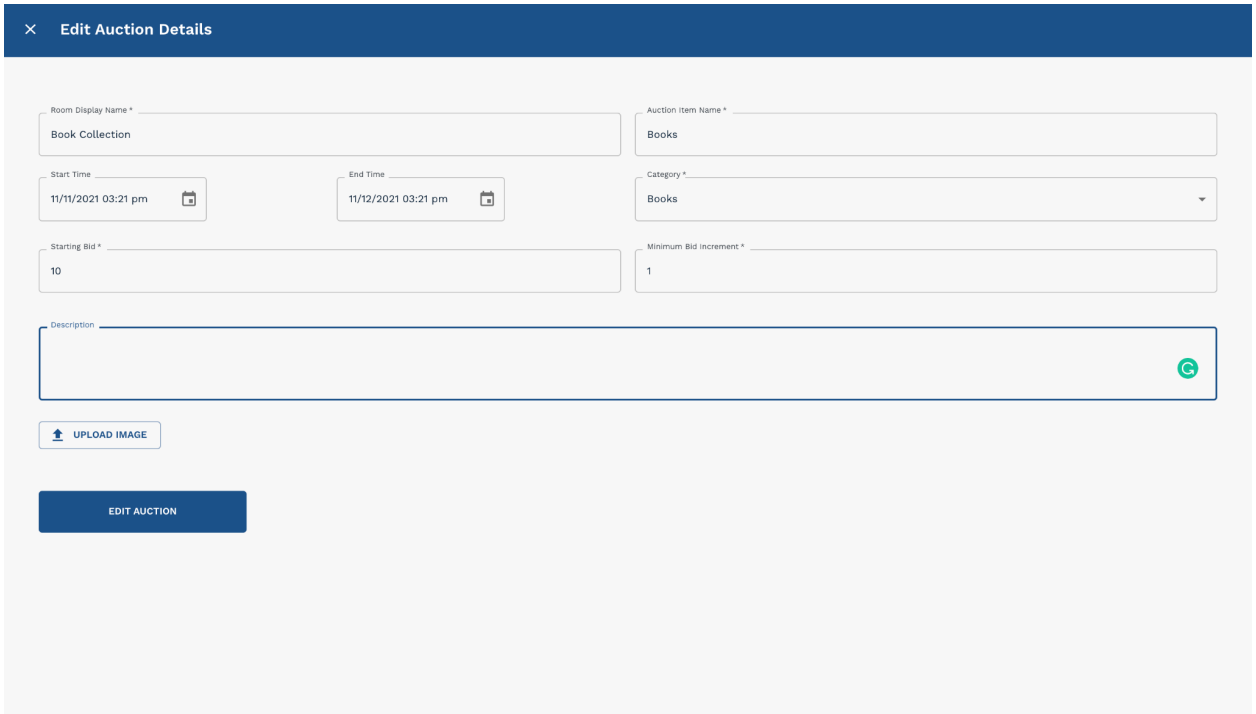


Fig 7.5.3 Demonstration of auction editing (Edit Dialog)

7.6 Auction Session

Users can access the auction room once it is open. The auction room displays the relevant information about the auction, and the minimum bid that users have to bid for.

In the Auction Room page, users can bid with a higher price than the current highest bid or starting bid.

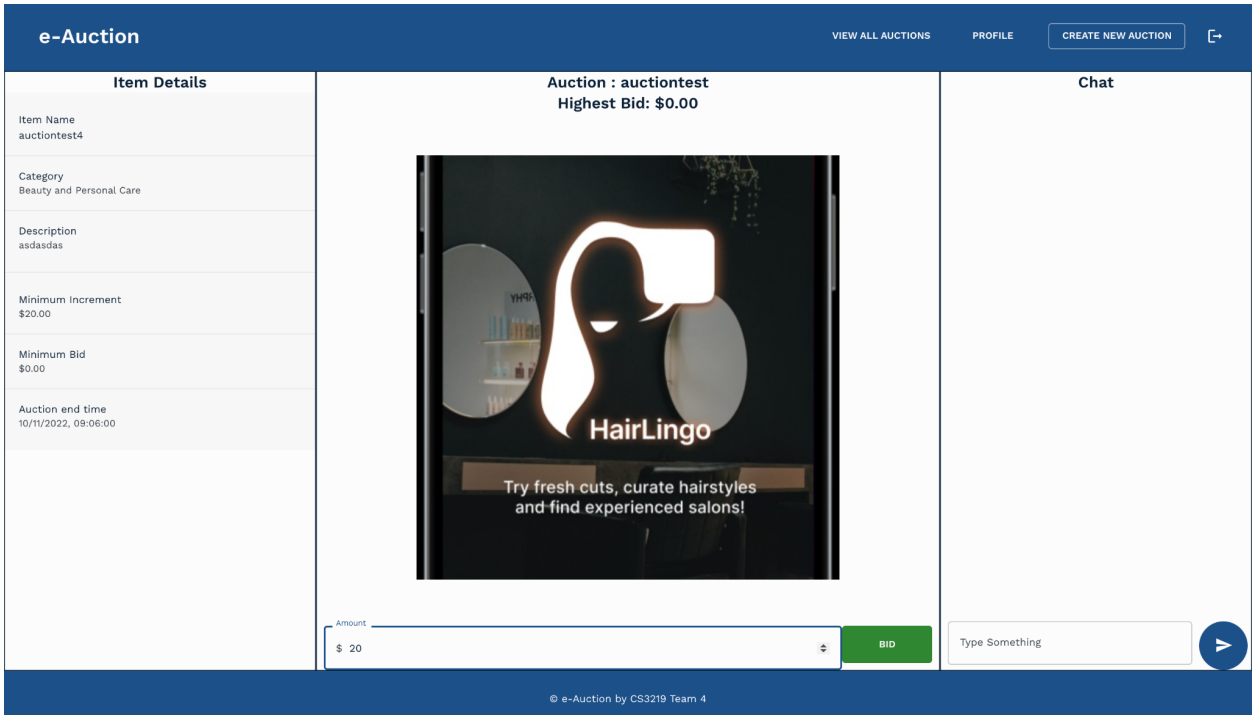


Fig 7.6.1(a) Demonstration of user making a bid

Fig 7.6.1(b) Demonstration of user making a bid (Handling of invalid bid)

The owner of the auction can opt to end an auction by clicking on the button.

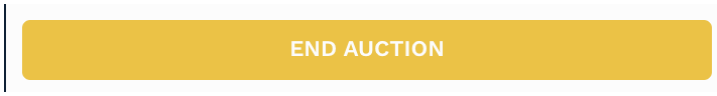


Fig 7.6.2(a) Demonstration of owner ending an auction (End Auction Button)

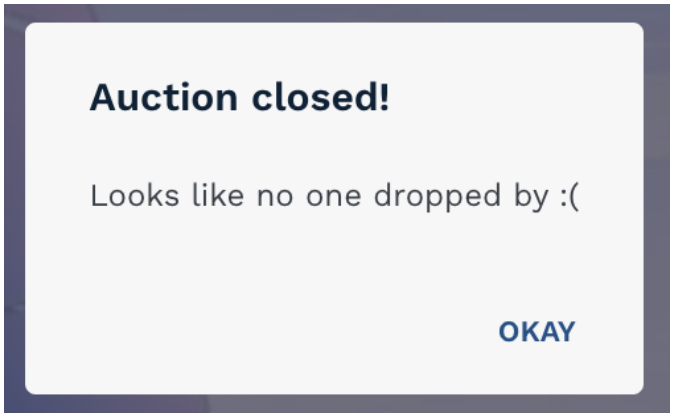


Fig 7.6.2(b) Demonstration of owner ending an auction (Prompt upon end auction when no one bidded)

Users will then be informed of the auction winner.

7.7 Auction Chat

The Auction Room allows chatting amongst users.

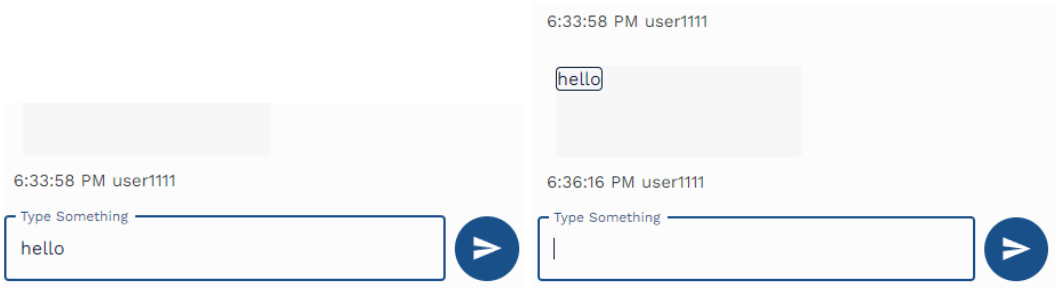


Fig 7.7.1 Demonstration of chat feature

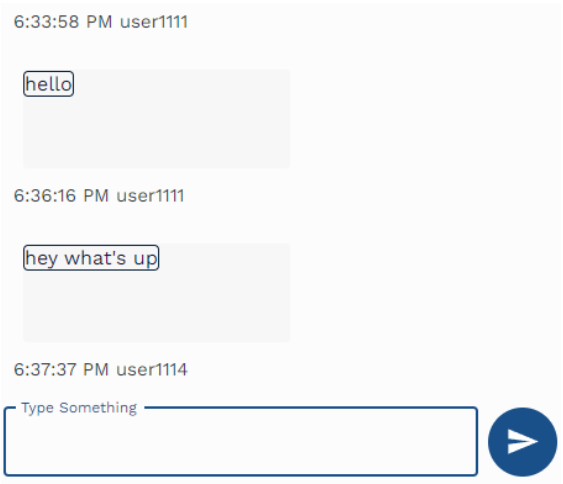


Fig 7.7.2 Demonstration of real time chat interaction with other users

8. Practical assumptions made

1. The application does not currently have a legitimate payment system implemented. All users are given 1000 currency credits upon registration to simulate depositing currency. Users are able to top-up an arbitrary amount of currency to simulate top-ups for system testing purposes.Withdrawals are also not possible and we assume the application is a closed system.

2. We assume items are preemptively delivered to a trusted middleman. Upon the end of an auction, a transaction is automatically made between the auction winner(highest bidder) and the auctioneer of the item.

3. We assume that auctions that are not manually ended by the auctioneer are considered inactive after the end date and time has elapsed. This happens when the auctioneer does not manually end the auction before the end date and time of the auction. The auctioneer will then have to delete the inactive listing if he wishes to.

9. Challenges faced and other remarks

1. We had trouble deciding on how to perform and ensure real time interactions for chat and bidding capabilities.
2. Communication between frontend and backend team members was not the most efficient due to division and specialization of labour. However, the expertise gained in each specific component more than made up for this.
3. Kubernetes and Google Cloud Platform services presented a steep learning curve. Thankfully, the abundance of online resources helped us to pull through.
4. Deployment required a lot of reconfiguration and debugging to get right.

10. Future improvements and enhancements

10.1 New Feature: Live Streaming

To better replicate real-life auctions, live streaming will be a good feature to add. A new microservice architecture can be added to handle the live streaming services.

Alternatively, voice and video chats can be added to assist the communication in real-life situations, while ensuring the efficiency of the platform.

10.2 New Feature: Auto relisting of auctions

In the case where an auction has ended but no bids have been placed, the user should have the option to automatically relist the auction. When creating an auction and entering the details, the user can simply check a box which allows for auto relisting if no bids are placed by the end time of the auction and specify the duration for which the relisted auction should run.

Appendix A. Tasks Completed by Sprint

Sprint	**Task**	**Completed By**
1	Update Jira with tasks in the backlog	All
1	Backend - User Account API	Simran
1	Backend - Auction Details API	Simran
1	Backend - Auction Details Testing (CI)	Priscilla
1	Frontend - Create Auction Page	Michelle
1	Frontend - Login and Sign Up page	David
1	Deployment - Auction Details Microservice	David
2	Backend - Auction Room API	Simran
2	Backend - Auction Room Manager API	David
2	Frontend - Auction Room page	David
2	Frontend - Homepage page & Backend - Search/Filter API	Michelle
2	API Gateway	Priscilla
3	Deployment - Auction Room Manager Microservice	David
3	Deployment - User Account Microservice	David
3	Backend - Currency Management API	Simran
3	Deployment - Currency Management Microservice	Simran
3	Deployment - Auction Room Microservice	Simran
3	Frontend - Profile Page	Michelle

3	Fix Bugs	All
3	Documentation	All