Team: Strawhats (G12)

S/N	Name	Email	Matriculation Number
1	Ritesh Kumar	ritesh.k@u.nus.edu	A0201829H
2	Kor Ming Soon	e0415784@u.nus.edu	A0201975A
3	Fang Junwei, Samuel	e0406144@u.nus.edu	A0199163U
4	Ong Si Min	e0426303@u.nus.edu	A0206380M

Github Code Repository Link:
https://github.com/CS3219-SE-Principles-and-Patterns/cs3219-project-ay2122-2122-s1-g12

Working Repo:
https://github.com/StrawHats12/CS3219_strawhats

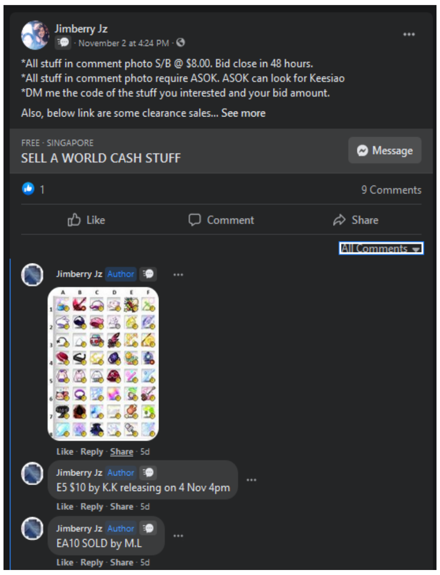
Deployment URL:
https://deploy.d35y0d6u0ucr1n.amplifyapp.com/

Table of Contents

1. Background and Purpose

1.1. Background

In niche markets such as the toys collectible market, there are no established platforms for collectors to market and sell their products. Collectors are forced to use Facebook marketplace or similar sites which are not well suited to selling these niche items.


Jimberry Jz
November 2 at 4:24 PM
*All stuff in comment photo S/B @ $8.00. Bid close in 48 hours.
*All stuff in comment photo require ASOK. ASOK can look for Keesiao
*DM me the code of the stuff you interested and your bid amount.
Also, below link are some clearance sales... See more

FREE · SINGAPORE
SELL A WORLD CASH STUFF
Message

1 · 9 Comments
Like · Comment · Share
All Comments

Jimberry Jz Author
Like · Reply · Share · 5d

Jimberry Jz Author
E5 $10 by K.K releasing on 4 Nov 4pm
Like · Reply · Share · 5d

Jimberry Jz Author
EA10 SOLD by M.L
Like · Reply · Share · 5d

Figure 1-1: Example of auctions in Facebook for Maplestory groups

Furthermore, toy collectors often have difficulty determining the valuation of their collectibles. This asymmetry of information has resulted in new collectors either paying too much or inexperienced sellers receiving too little. Hence, possibly causing an unfair valuation for prized possessions of figurines.

1.2. Purpose

We aim to provide a platform for commerce within the toy collectors community. However, the concept of auctioning products applies to all niche and luxury products, and we hope to also expand our platform to allow these auctioning mechanisms to be available to the layperson. On our platform, bidders and sellers will be able to view the ongoing valuation of the collectibles transparently through our real time bidding mechanism.

We also aim to level the playing field for both experienced and novice collectors so that they are able to purchase and sell their collectibles at a fair price. Allowing sellers to showcase their collectibles for easy viewing. The viewing of the product comes in the form of livestreaming or photo uploads.

Auctioning helps the seller to better value their product, even for passion projects like self-moulded figurines. This can help artists / passion projects become a viable income source as well.

2. Project Management

2.1. Team Contributions

Name	Technical Contributions	Non-Technical Contributions
Fang Junwei, Samuel	Frontend Accounts Service Listings Service CI/CD Pipelines	Scrum Master
Kor Ming Soon	Bidding service	Documentation Quality
Ong Si Min	Messaging service	Code Quality
Ritesh Kumar	Livestream service	Setup of PM Tools

2.2. Development Process

We understand the importance of planning and project management for software engineering projects, so we followed the agile SDLC in our implementation, where we repeated the cycle upon each iteration of the application.

2.2.1. Stages of Development

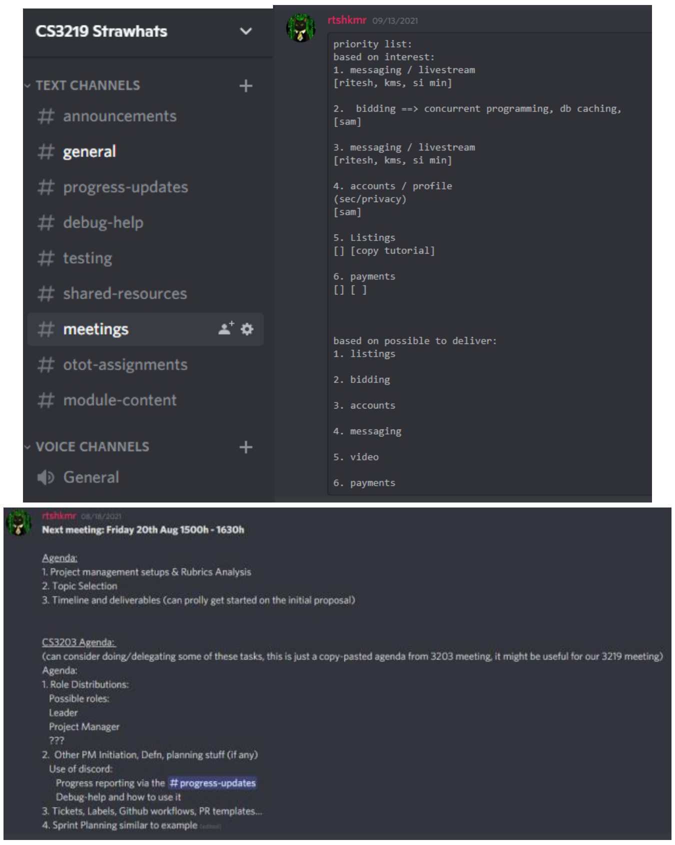
This project had multiple novel areas for the team. Other than being new to implementing microservices and deploying them, we were new to Socket.IO and its uses as well as the domain of media streaming. To plan for such novelty, we set aside the 3 broad iterations. We allocated 2-3 weeks for Proofs of Concept (POCs) where we sought existing tutorials for the technologies that were new to us. Subsequently, we allocated 4 weeks for implementation of a MVP followed by deployment and integration. Conducting basic user testing amongst ourselves allowed us to have the necessary feedback loops to make adjustments along the way. The rest of the time was spent on UX improvements.

2.2.2. Communication Workflows

Having structured communication platforms was important to keep communication as asynchronous as possible. Discord was a primary source of communication, and we structured our communication as follows:

<u>Weekly Scrums on Discord</u>

In our own discord channel, where we held our weekly meetings, we also created various text channels where we posted the agenda of our weekly scrums as well as the brief minutes of our meeting.



rtshkmr 09/13/2021

```
priority list:
based on interest:
1. messaging / livestream
[ritesh, kms, si min]

2.  bidding ==> concurrent programming, db caching,
[sam]

3. messaging / livestream
[ritesh, kms, si min]

4. accounts / profile
(sec/privacy)
[sam]

5. Listings
[] [copy tutorial]

6. payments
[] [ ]



based on possible to deliver:
1. listings

2. bidding

3. accounts

4. messaging

5. video

6. payments
```

rtshkmr 08/18/2021

Next meeting: Friday 20th Aug 1500h - 1630h

<u>Agenda:</u>
1. Project management setups & Rubrics Analysis
2. Topic Selection
3. Timeline and deliverables (can prolly get started on the initial proposal)

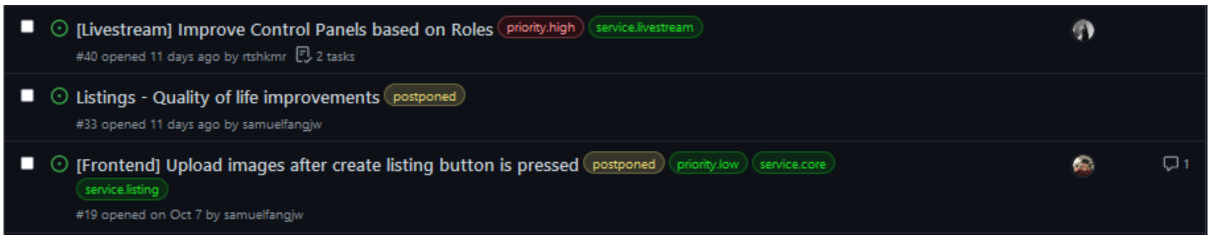
<u>CS3203 Agenda:</u>
(can consider doing/delegating some of these tasks, this is just a copy-pasted agenda from 3203 meeting, it might be useful for our 3219 meeting)
Agenda:
1. Role Distributions:
  Possible roles:
  Leader
  Project Manager
  ???
2. Other PM Initiation, Defn, planning stuff (if any)
  Use of discord:
    Progress reporting via the #progress-updates
    Debug-help and how to use it
3. Tickets, Labels, Github workflows, PR templates...
4. Sprint Planning similar to example (edited)

<u>Figure 2-1: Examples of our work process flow in Discord</u>

<u>Issue Tracker & Labels</u>

After our meetings on Discord, we would then proceed to formally delegate our taskings by creating allocated issues on Github. In addition to the issues created, we also created a variety of labels to determine the priority as well as urgency of the issue itself. The labels considered our management workflow, and we segmented the notable labels in the following few categories:

1. service: service, accounts, service, listings etc.
2. urgency: high, medium, low
3. priority: high, medium low


[Livestream] Improve Control Panels based on Roles priority.high service.livestream
#40 opened 11 days ago by rtshkmr 2 tasks

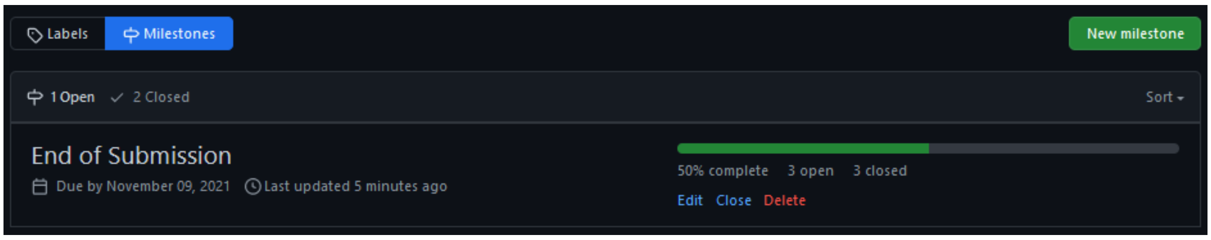
Listings - Quality of life improvements postponed
#33 opened 11 days ago by samuelfangjw

[Frontend] Upload images after create listing button is pressed postponed priority.low service.core service.listing
#19 opened on Oct 7 by samuelfangjw

Figure 2-2: Snapshot of our Github Issues

Milestones

We created some Github milestones to track the overall progress of our application to ensure that we were in accordance with the deadline of the project submission. Our first milestone was the completion of the basic microservices, followed by the "Big Feature POC" where we integrated and refined the microservices with the front end. Lastly, the final milestone denotes the final tweaks, hosting of our application as well as the documentation of our project.

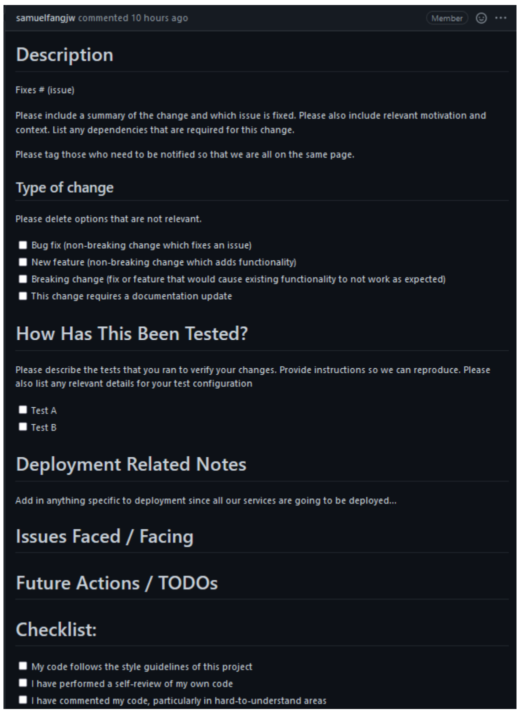


End of Submission
Due by November 09, 2021 Last updated 5 minutes ago
50% complete 3 open 3 closed

Figure 2-3: Snapshot of our Github Milestones

Pull Request Template

In our pull request, we decided to have a templated pull request description embedded, it served as a reminder for us to ensure that we had completed our series of checks prior to merging the pull request.



samuelfangjw commented 10 hours ago

Member

Description

Fixes # (issue)

Please include a summary of the change and which issue is fixed. Please also include relevant motivation and context. List any dependencies that are required for this change.

Please tag those who need to be notified so that we are all on the same page.

Type of change

Please delete options that are not relevant.

- [ ] Bug fix (non-breaking change which fixes an issue)
- [ ] New feature (non-breaking change which adds functionality)
- [ ] Breaking change (fix or feature that would cause existing functionality to not work as expected)
- [ ] This change requires a documentation update

How Has This Been Tested?

Please describe the tests that you ran to verify your changes. Provide instructions so we can reproduce. Please also list any relevant details for your test configuration

- [ ] Test A
- [ ] Test B

Deployment Related Notes

Add in anything specific to deployment since all our services are going to be deployed...

Issues Faced / Facing

Future Actions / TODOs

Checklist:

- [ ] My code follows the style guidelines of this project
- [ ] I have performed a self-review of my own code
- [ ] I have commented my code, particularly in hard-to-understand areas

Figure 2-4: Snapshot of our Github Pull Request Description

3. Requirements

3.1. Functional Requirements

3.1.1. Accounts & Profiles Services

Functional Requirements		Priority
A1	Allow users to register an account	H
A2	Users have to be authenticated being given access to the platform	H
A3	Allow users to edit their own profile	H
A4	Allow users to view a profile	H
A5	Allow users to delete their account	M
A6	Allow users to customize their profiles (e.g profile picture, biography etc)	L
A7	Display reviews of the seller from past or existing buyers	L

3.1.2. Listing Service

Functional Requirements		Priority
B1	Allow users to create a listing with title, description and other details	H
B2	Allow users to delete a listing	H
B3	Allow users to edit a listing	H
B4	Allow users to retrieve a listing to view	H

3.1.3. Bidding Service

Functional Requirements		Priority
C1	Allow users to make a bid for an item and update the highest bid	H

C2	Store the history of bids for a particular item	M

3.1.4. Livestream Service

Functional Requirements		Priority
D1	Allow sellers to start a livestream	H
D2	Allow sellers to stop a livestream	H
D3	Allow the integration of messaging service within the livestream	M

3.1.5. Messaging Service

Functional Requirements		Priority
E1	Bidders can start a one-on-one chat function with sellers.	M
E2	Messaging service can be integrated as a UI element to value-add to livestreams.	M
E3	Bidders and sellers can send messages to chat with one another (one-to-many chat function) (integrated inside livestream service)	M

3.2. Non-Functional Requirements (NFRs)

Our NFRs are listed as follows (in order of priority):

1. Scalability
2. Availability
3. Security
4. Performance

We prioritized scalability first as we anticipate high traffic especially during the last few hours before a listing's deadline. This is followed by availability to ensure uptime of the service. Security is in third place because the nature of our application is a platform that enables financial transactions and exchanging of personal details. Hence, there is a need to ensure that security is not compromised, and bids are securely placed. Lastly, we have performance as we feel that the user experience will be smooth even without deliberate enhancements.

3.2.1. Scalability

Key features:

1. The application should support more than 1000 concurrent users.

Feature	Implementation
1	Use of Application Load Balancer to direct incoming traffic to different tasks[1], allowing us to scale horizontally during peak traffic periods. All microservices are assigned an autoscaling group and can be scaled horizontally to meet demand. Use of Mux Video as a Infrastructure-as-a-service solution ensures that the delivery of streams is auto scaled by Mux's infrastructure.

3.2.2. Availability

Key features:

1. The application should be up even in the event of an outage

Feature	Implementation
1	Use of two different subnets within two different availability zones. In the event of an outage, traffic can be directed to containers in other subnets. Use of redundant read only replica nodes in redis cluster that can take over the master node in the event it fails. Use of Mux video for infrastructure outsources the content distribution to Mux and Mux assures a guarantee against service disruptions. Furthermore, using Adaptive Bitrate Streaming accounts for varying network speeds and screen sizes, hence is available to a variety of clients.

3.2.3. Performance

Key features:

1. Bidding, messaging and livestreaming services should provide real-time communication and updates
2. The application should ensure at most 2 seconds response delay for any data retrieval or update.

[1] Tasks is an ECS specific definition similar to Pods in Kubernetes

Feature	Implementation
1	Use of Socket.IO to enable bidirectional and real-time communication between clients.
2	Use of CloudFront to serve a cached version of the content from edge locations, enabling faster response time Use of low-latency DynamoDb for a central store of records.

3.2.4. Security

Key features:

1. All microservices should only allow relevant IP addresses (e.g front end) to access it.
2. Private data cannot be accessed or modified by users who are unauthorized and unauthenticated.

Feature	Implementation
1	Use of security groups to only allow access from authorized IP addresses
2	Use of JWT and Cognito Identity Pools and IAM (identity and access management) policies to ensure that only authorized users can access and modify application resources.

4. Application User Interface

Strawhats has been designed to be an easy-to-use auction site with an intuitive UI. This section outlines how the various microservices have been integrated into a cohesive frontend application.

4.1. Home Page

At a glance, the home page provides the view of a set of featured listings on Strawhats.

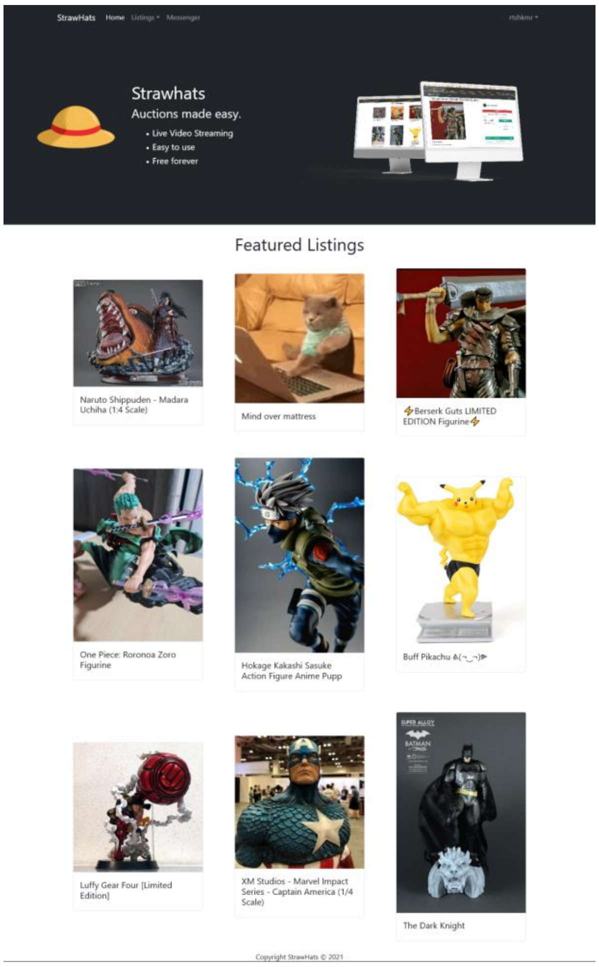

StrawHats
Home
Listings ▾
Messenger
rtshkmr ▾
Strawhats
Auctions made easy.
• Live Video Streaming
• Easy to use
• Free forever
Featured Listings
Naruto Shippuden - Madara Uchiha (1:4 Scale)
Mind over mattress
⚡Berserk Guts LIMITED EDITION Figurine⚡
One Piece: Roronoa Zoro Figurine
Hokage Kakashi Sasuke Action Figure Anime Pupp
Buff Pikachu ᕙ(⇀‸↼‶)ᕗ
Luffy Gear Four [Limited Edition]
XM Studios - Marvel Impact Series - Captain America (1/4 Scale)
The Dark Knight
Copyright StrawHats © 2021

Figure 4-1: Snapshot of Homepage

4.2. Site Navigation

Site navigation is intuitively done via the top navbar. In the screenshot below, we see a compact view of this navbar that shows the navigation options presented to the user.

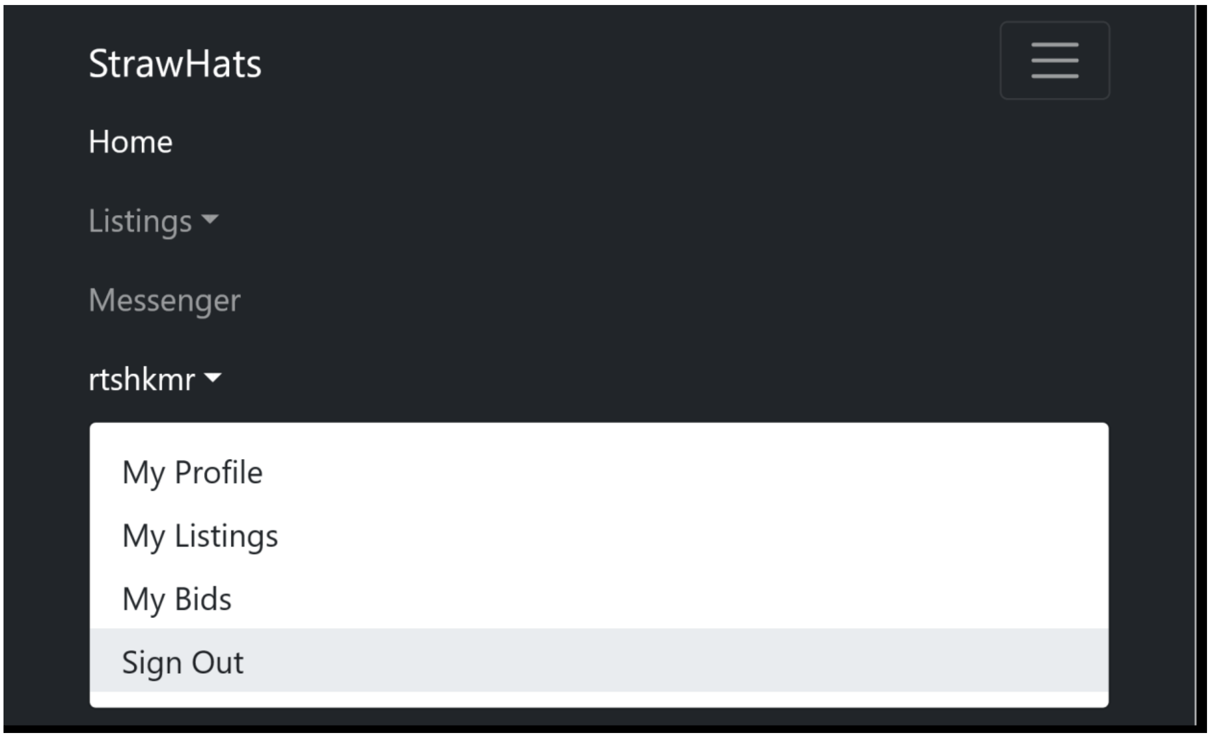
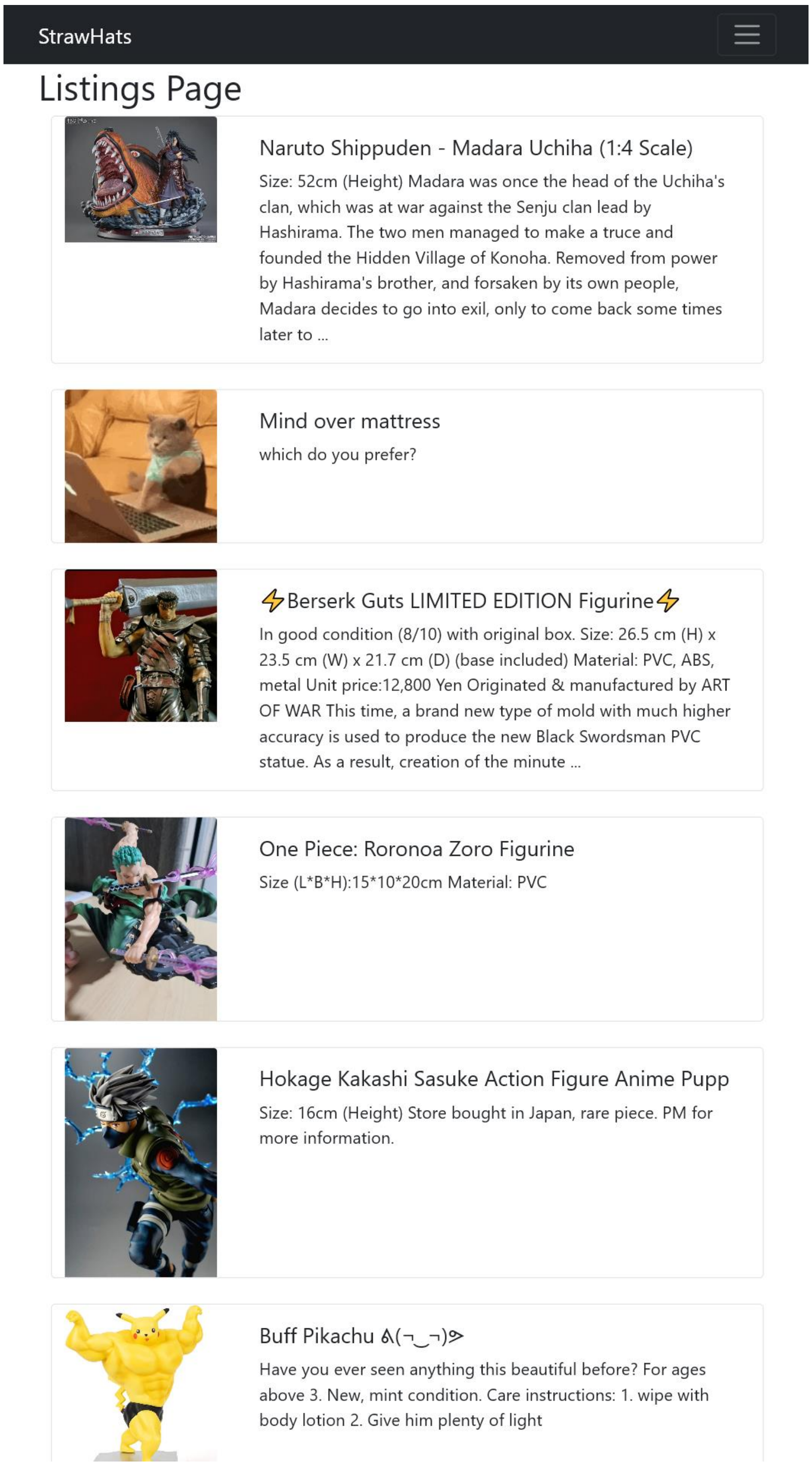


Figure 4-2: Snapshot of Navigation Bar

4.3. Detailed Listings Page

A detailed view of all the current listings and their descriptions provides a bigger, catalogue-view of the listings currently being auctioned.



StrawHats

Listings Page

Naruto Shippuden - Madara Uchiha (1:4 Scale)

Size: 52cm (Height) Madara was once the head of the Uchiha's clan, which was at war against the Senju clan lead by Hashirama. The two men managed to make a truce and founded the Hidden Village of Konoha. Removed from power by Hashirama's brother, and forsaken by its own people, Madara decides to go into exil, only to come back some times later to ...

Mind over mattress

which do you prefer?

⚡Berserk Guts LIMITED EDITION Figurine⚡

In good condition (8/10) with original box. Size: 26.5 cm (H) x 23.5 cm (W) x 21.7 cm (D) (base included) Material: PVC, ABS, metal Unit price:12,800 Yen Originated & manufactured by ART OF WAR This time, a brand new type of mold with much higher accuracy is used to produce the new Black Swordsman PVC statue. As a result, creation of the minute ...

One Piece: Roronoa Zoro Figurine

Size (L*B*H):15*10*20cm Material: PVC

Hokage Kakashi Sasuke Action Figure Anime Pupp

Size: 16cm (Height) Store bought in Japan, rare piece. PM for more information.

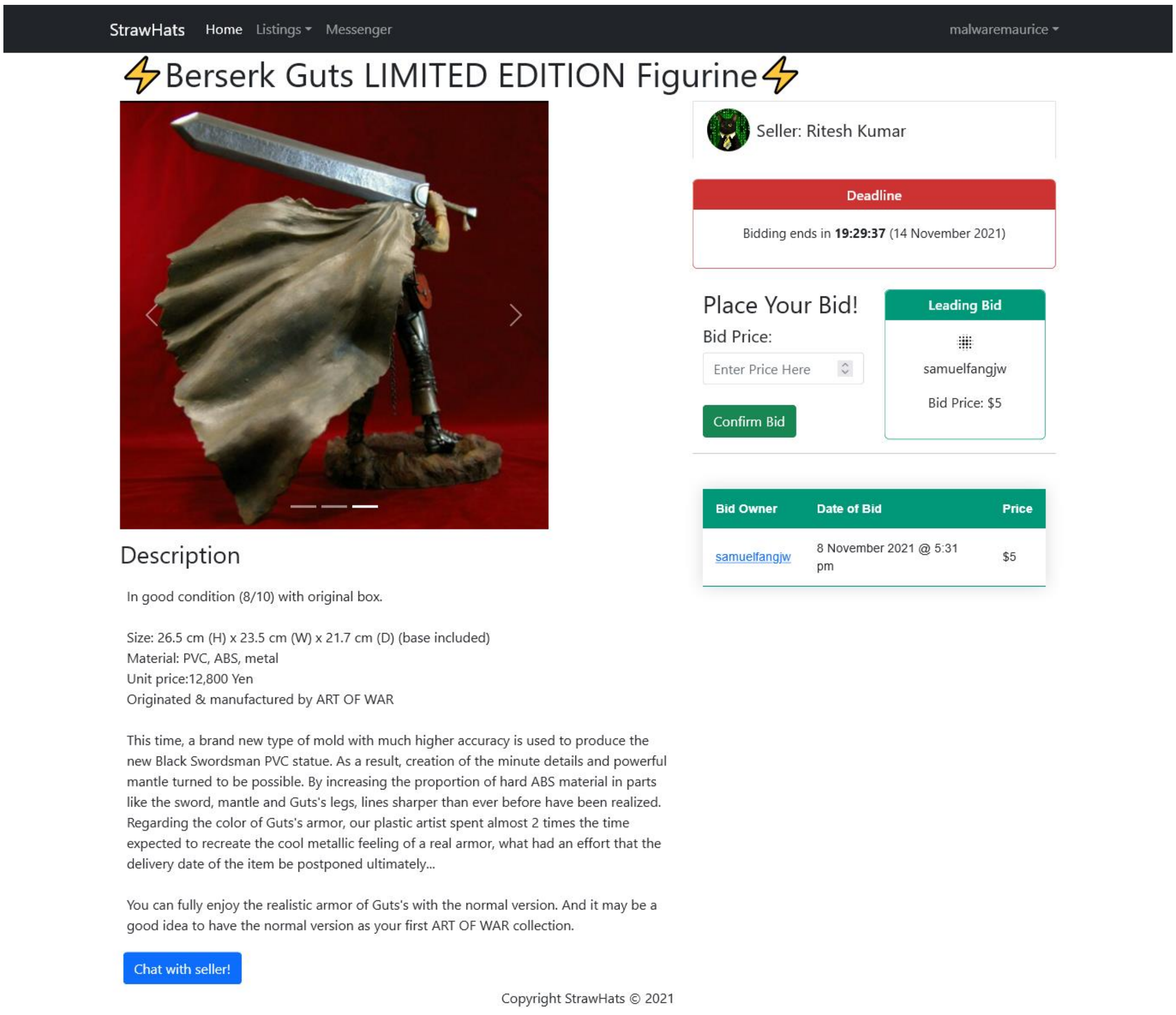
Buff Pikachu ᕦ(¬‿¬)ᕤ

Have you ever seen anything this beautiful before? For ages above 3. New, mint condition. Care instructions: 1. wipe with body lotion 2. Give him plenty of light

Figure 4-3: Snapshot of Detailed Listings

4.4. Generic Listing Page

Viewing a single listing displays the main interface for the auction. The listings service's content has been integrated by displaying the image objects from our S3 store as well as listings description.



⚡Berserk Guts LIMITED EDITION Figurine⚡

Seller: Ritesh Kumar

Deadline

Bidding ends in **19:29:37** (14 November 2021)

Place Your Bid!

Bid Price:

Leading Bid

samuelfangjw

Bid Price: $5

Bid Owner	Date of Bid	Price
samuelfangjw	8 November 2021 @ 5:31 pm	$5

Description

In good condition (8/10) with original box.

Size: 26.5 cm (H) x 23.5 cm (W) x 21.7 cm (D) (base included)
Material: PVC, ABS, metal
Unit price:12,800 Yen
Originated & manufactured by ART OF WAR

This time, a brand new type of mold with much higher accuracy is used to produce the new Black Swordsman PVC statue. As a result, creation of the minute details and powerful mantle turned to be possible. By increasing the proportion of hard ABS material in parts like the sword, mantle and Guts's legs, lines sharper than ever before have been realized. Regarding the color of Guts's armor, our plastic artist spent almost 2 times the time expected to recreate the cool metallic feeling of a real armor, what had an effort that the delivery date of the item be postponed ultimately...

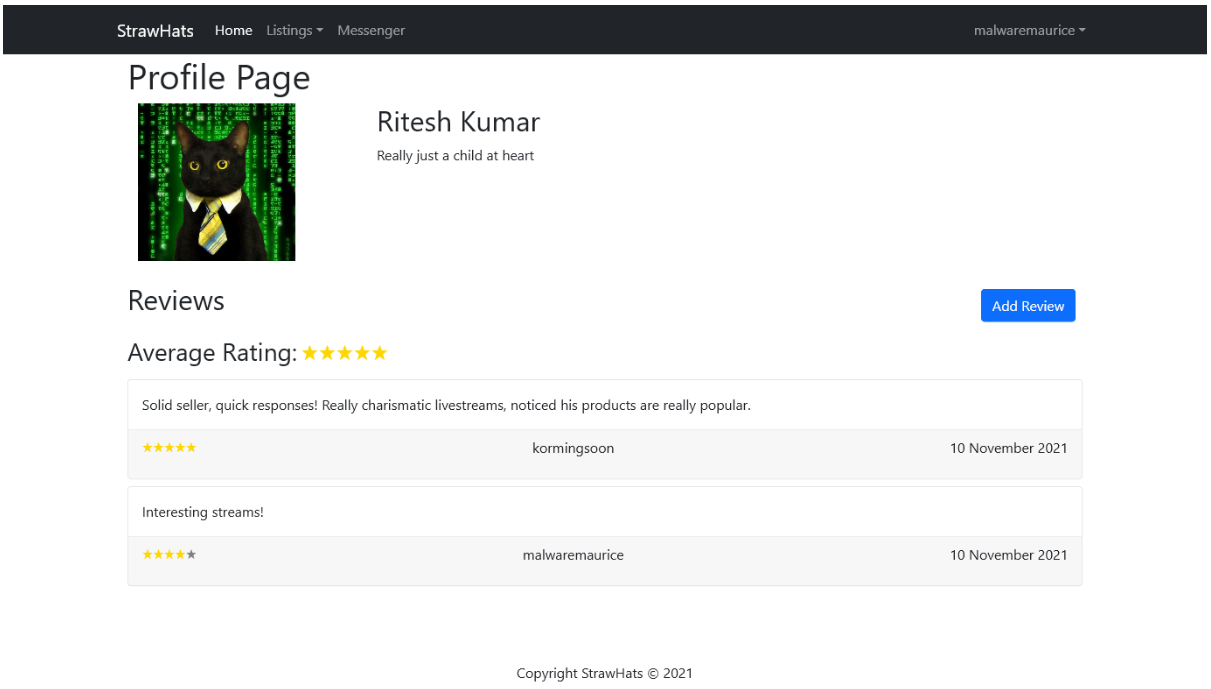
You can fully enjoy the realistic armor of Guts's with the normal version. And it may be a good idea to have the normal version as your first ART OF WAR collection.

Copyright StrawHats © 2021

Figure 4-4: Snapshot of Listing Page

4.5. Viewing Profiles

Seeing the seller profile and adding reviews to them is a quick process.



StrawHats Home Listings Messenger malwaremaurice

Profile Page

Ritesh Kumar

Really just a child at heart

Reviews

Add Review

Average Rating: ★★★★★

Solid seller, quick responses! Really charismatic livestreams, noticed his products are really popular.

★★★★★	kormingsoon	10 November 2021

Interesting streams!

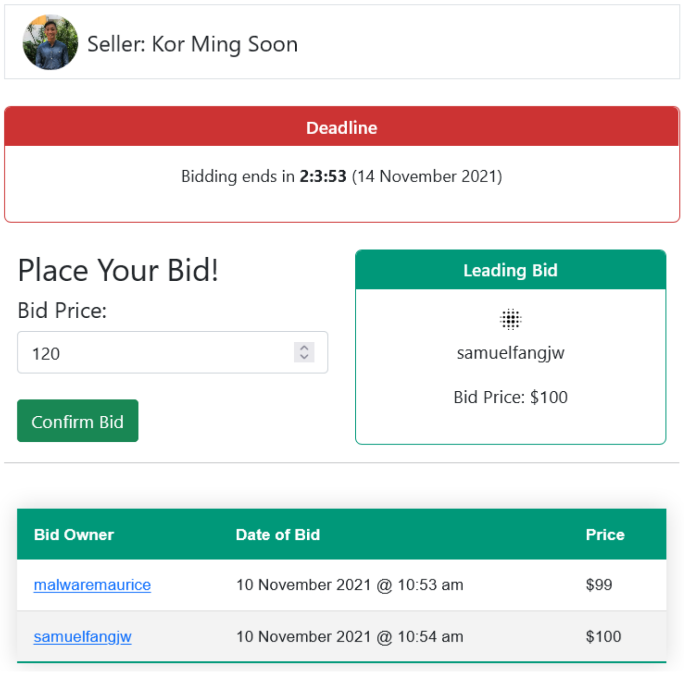
★★★★★	malwaremaurice	10 November 2021

Copyright StrawHats © 2021

Figure 4-5: Snapshot of Profile Page

4.6. Bidding

Bidding service integrations shows the leading bid. It encourages the bidding of a price higher than the leading bid.



Seller: Kor Ming Soon

Deadline

Bidding ends in **2:3:53** (14 November 2021)

Place Your Bid!

Bid Price:

120

Confirm Bid

Leading Bid

samuelfangjw

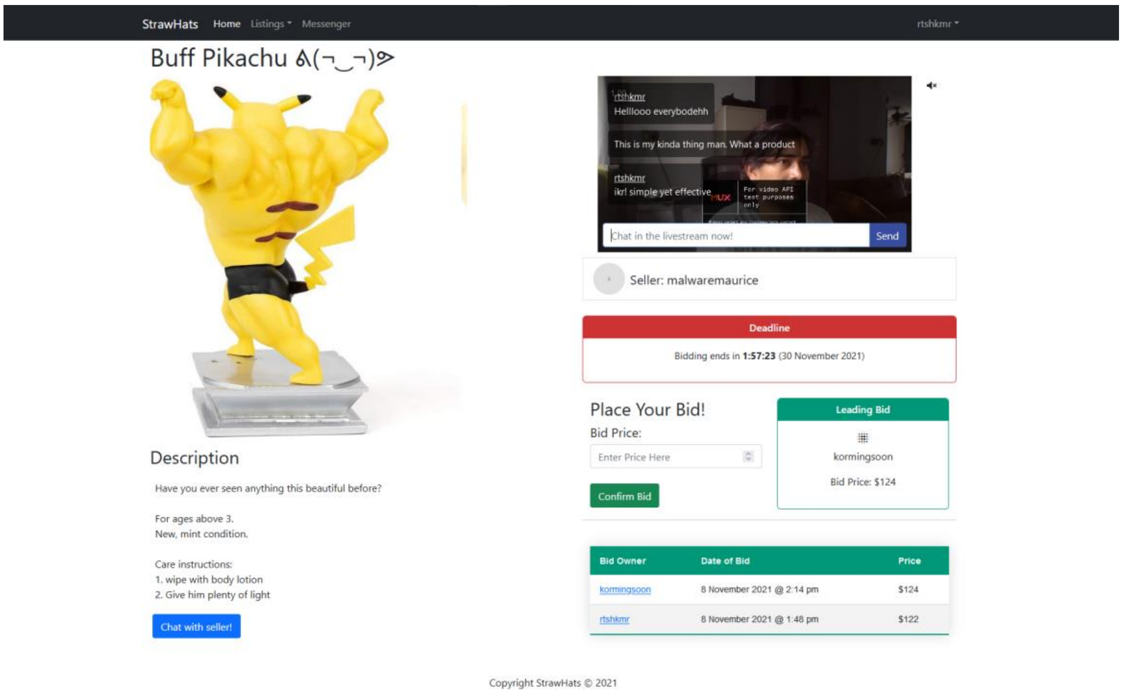
Bid Price: $100

Bid Owner	Date of Bid	Price
malwaremaurice	10 November 2021 @ 10:53 am	$99
samuelfangjw	10 November 2021 @ 10:54 am	$100

Figure 4-6: Snapshot of Bidding Section

4.7. Listing with Active Live Stream

When a livestream is active, a video player pops up. This stream gives sellers an opportunity to market their product. The player has an ephemeral chat function, a messenger api integration, that allows viewers to continually send messages during the livestream.



Buff Pikachu ᕦ(¬‿¬)ᕤ

rtshkmr
Hellllooo everybodehh

This is my kinda thing man. What a product

rtshkmr
ikr! simple yet effective

Seller: malwaremaurice

Deadline

Bidding ends in **1:57:23** (30 November 2021)

Place Your Bid!

Bid Price:

Leading Bid

kormingsoon

Bid Price: $124

Bid Owner	Date of Bid	Price
kormingsoon	8 November 2021 @ 2:14 pm	$124
rtshkmr	8 November 2021 @ 1:48 pm	$122

Description

Have you ever seen anything this beautiful before?

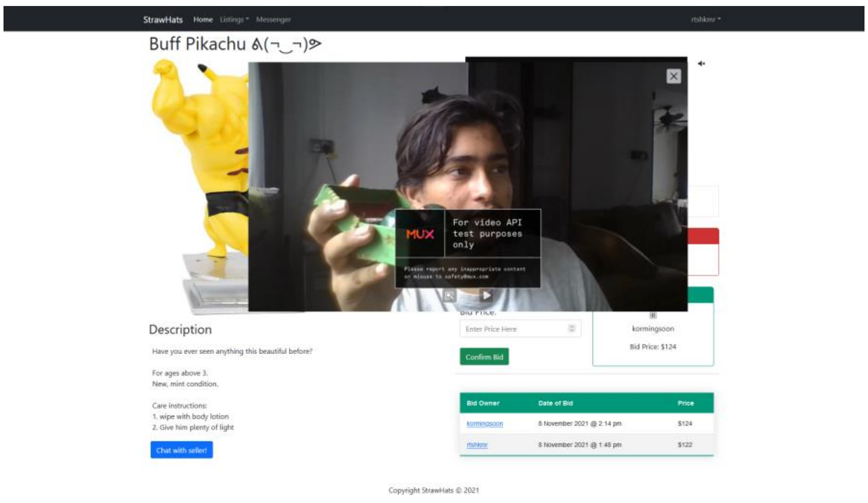
For ages above 3.
New, mint condition.

Care instructions:
1. wipe with body lotion
2. Give him plenty of light

Copyright StrawHats © 2021

Figure 4-7: Snapshot of Livestream in Listing Page

Additionally, picture-in-picture mode is enabled so that the viewer may keep the video on at the corner of the screen while multitasking on other tasks[2].


StrawHats
Home
Listings
Messenger
rtshkmr
MUX
For video API test purposes only
Please report any inappropriate content or misuse to safety@mux.com
Enter Price Here
Confirm Bid
Chat with seller!

Buff Pikachu ᕦ(¬‿¬)ᕤ

kormingsoon

Bid Price: $124

Bid Owner	Date of Bid	Price
kormingsoon	8 November 2021 @ 2:14 pm	$124
rtshkmr	8 November 2021 @ 1:48 pm	$122

Description

Have you ever seen anything this beautiful before?

For ages above 3.
New, mint condition.

Care instructions:
1. wipe with body lotion
2. Give him plenty of light

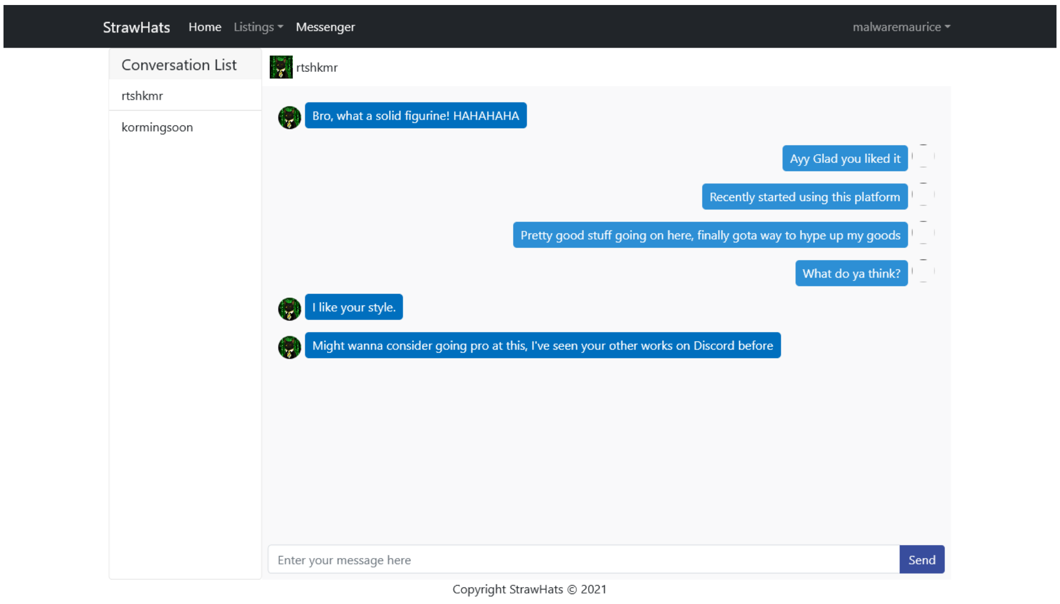
Copyright StrawHats © 2021

Figure 4-8: Snapshot of Livestream Picture-In-Picture View

[2] : note the current livestream has a watermark because we utilized a free testing account that allows for unlimited api testing to be done.

4.8. One-to-one Chat

In the listing page, a buyer can contact the seller by clicking on the chat button, which redirects the user to the messenger page. If this is the first time the buyer is starting a 1-on-1 conversation with the seller, a new conversation object will be created in DynamoDB. This chat is persisted across multiple sessions, unlike the ephemeral livestream chat.



Bro, what a solid figurine! HAHAHAHA

Ayy Glad you liked it

Recently started using this platform

Pretty good stuff going on here, finally gota way to hype up my goods

What do ya think?

I like your style.

Might wanna consider going pro at this, I've seen your other works on Discord before

Figure 4-9: Snapshot of Messaging

4.9. Forms and Control Panels

Form interactions are necessary for updating profiles and creating listings, amongst others. Simple form interactions allow for quick setting up of listings by sellers.

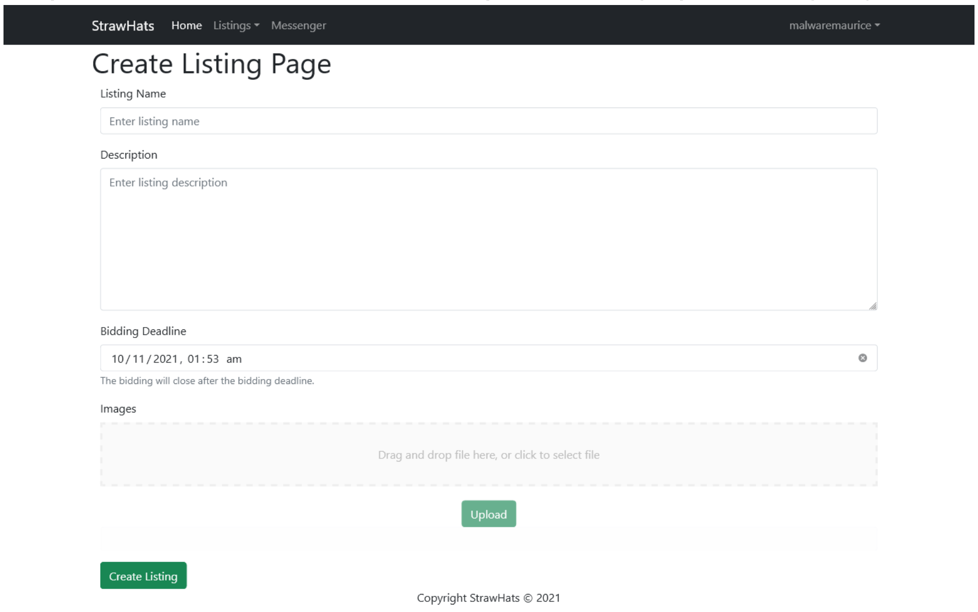
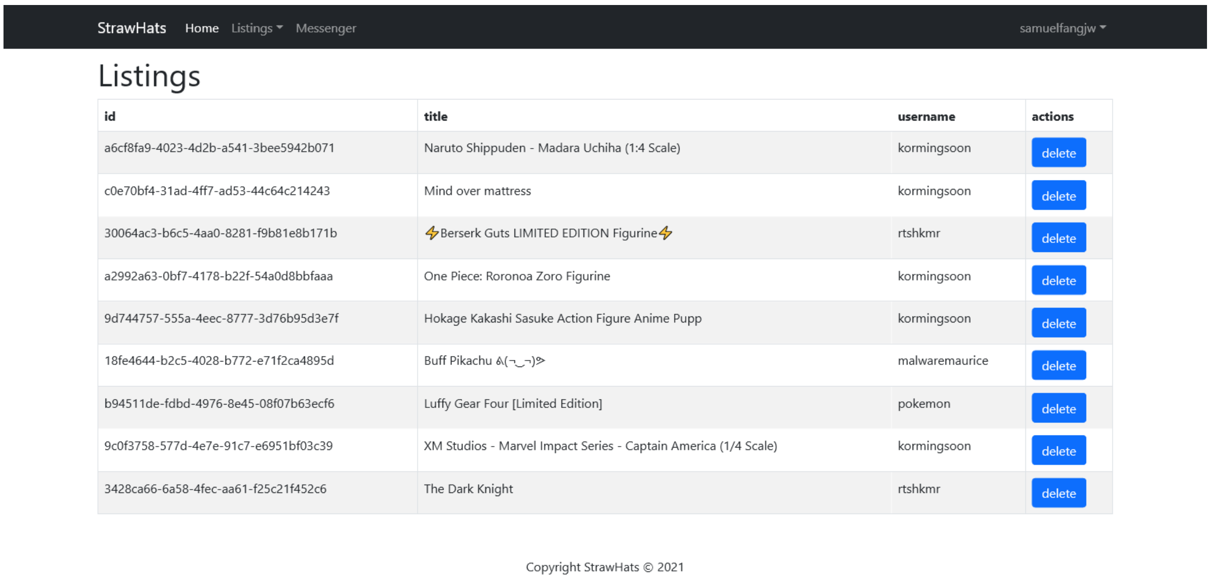


Figure 4-10: Snapshot of Listing Creation Panel

4.9.1. Admin Control Panel

StrawHats includes a handy admin panel that allows administrators to perform common administrator-only actions such as deleting listings.



Listings

id	title	username	actions
a6cf8fa9-4023-4d2b-a541-3bee5942b071	Naruto Shippuden - Madara Uchiha (1:4 Scale)	kormingsoon	delete
c0e70bf4-31ad-4ff7-ad53-44c64c214243	Mind over mattress	kormingsoon	delete
30064ac3-b6c5-4aa0-8281-f9b81e8b171b	⚡Berserk Guts LIMITED EDITION Figurine⚡	rtshkmr	delete
a2992a63-0bf7-4178-b22f-54a0d8bbfaaa	One Piece: Roronoa Zoro Figurine	kormingsoon	delete
9d744757-555a-4eec-8777-3d76b95d3e7f	Hokage Kakashi Sasuke Action Figure Anime Pupp	kormingsoon	delete
18fe4644-b2c5-4028-b772-e71f2ca4895d	Buff Pikachu ᕙ(¬‿¬)ᕗ	malwaremaurice	delete
b94511de-fdbd-4976-8e45-08f07b63ecf6	Luffy Gear Four [Limited Edition]	pokemon	delete
9c0f3758-577d-4e7e-91c7-e6951bf03c39	XM Studios - Marvel Impact Series - Captain America (1/4 Scale)	kormingsoon	delete
3428ca66-6a58-4fec-aa61-f25c21f452c6	The Dark Knight	rtshkmr	delete

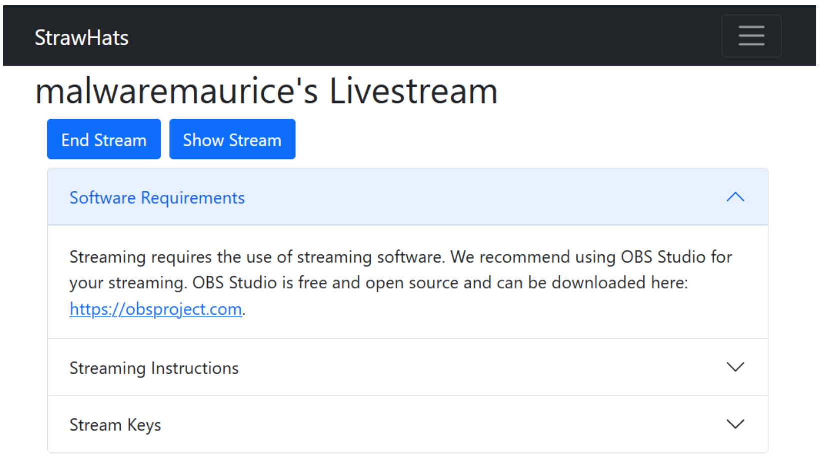
Copyright StrawHats © 2021

Figure 4-11: Snapshot of Admin Panel

This makes it convenient for administrators to moderate content (such as inappropriate listings) or help users who have difficulty performing certain actions.

4.9.2. Livestream Control Panel

Since Livestream creation is a more nuanced process, we found the use of an accordion to be a useful step-by-step guide on how a seller can set up a livestream. Interacting with the accordion in a step-by-step manner intuitively allows users to successfully input their stream keys into their choice transcoders and push their stream to our application.



malwaremaurice's Livestream

Software Requirements

Streaming requires the use of streaming software. We recommend using OBS Studio for your streaming. OBS Studio is free and open source and can be downloaded here: https://obsproject.com.

Streaming Instructions

Stream Keys

Figure 4-12: Snapshot of Livestream Instruction Panel

5. Developer Documentation

In the following section, we refer to various AWS (Amazon Web Services) cloud services by their acronyms. These services have been abbreviated, and a glossary has been provided for clarification of terms.

5.1. Design

5.1.1. Overall Architecture

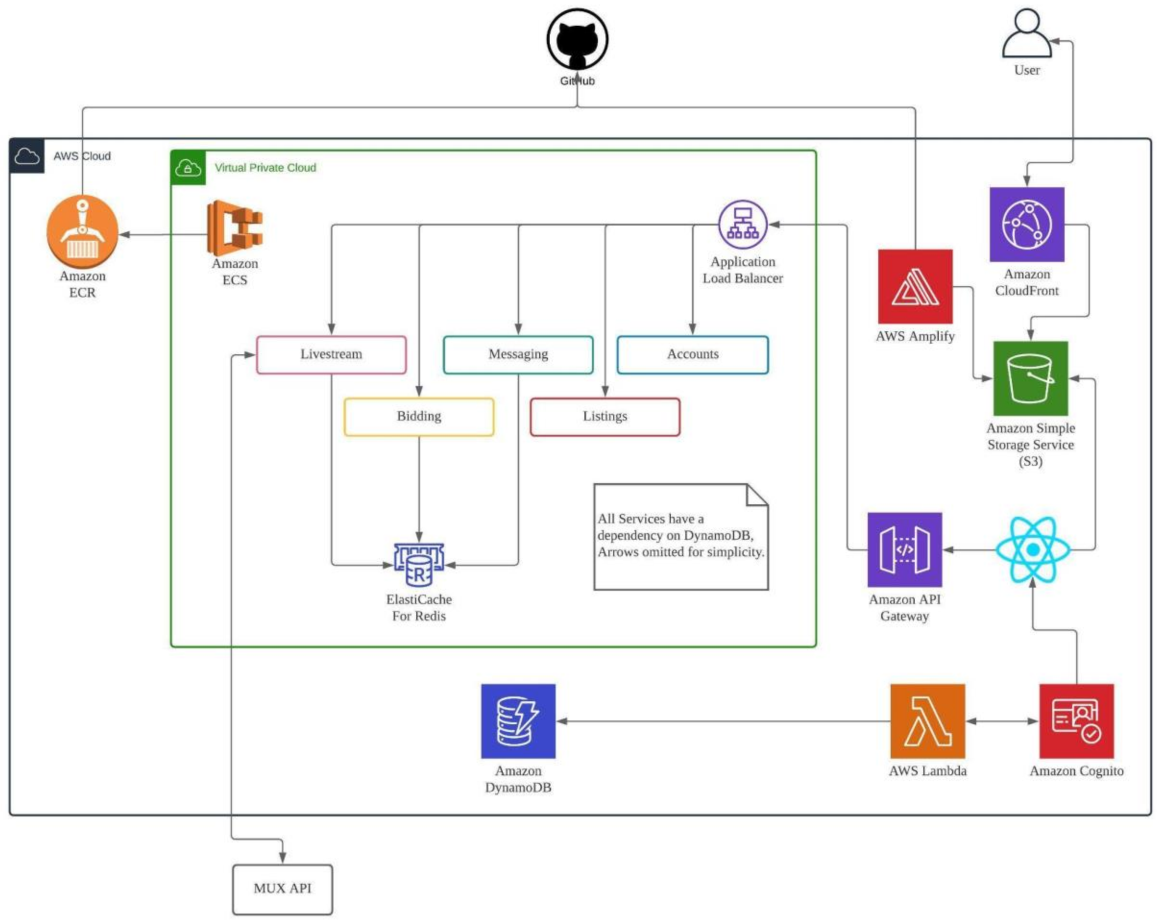


Figure 5-1: Overall Architecture Diagram

Strawhats is a Web Application built primarily using AWS. When designing the architecture for Strawhats, we prioritised scalability, availability and performance, while keeping in mind security concerns.

5.1.2. Microservice Architecture

Our application is divided into the following loosely coupled microservices:

1. Listings Service
2. Accounts Service
3. Messaging Service
4. Livestreaming Service
5. Bidding Service

The decision to use Microservice Architecture (vs Monolithic Architecture) was made based on a value judgement. The factors we considered are listed below.

Factor	Monolithic Architecture	Microservice Architecture
Scalability	Easy to scale vertically (more resources), but might be difficult to scale horizontally, especially since modules within a monolithic application tend to be tightly coupled.	Easy to scale each individual service horizontally since they are independent loosely coupled services.
Availability	Difficulty scaling the application horizontally means increased downtime if the application goes down.	Easy to scale horizontally so if one instance of the service goes down, traffic can be routed to other instances of the service minimising downtime.
Performance	Possibly higher performance since fewer API calls needed for complex operations involving multiple modules within the monolith.	Possibly worse performance if multiple API calls are needed to perform a complex action. Can be mitigated by smart design decisions which decrease coupling between services.
Security	Unlikely to differ between the two since traffic to API endpoints is encrypted (HTTPS) in both cases. The use of JWT also makes it simple to authenticate users across different microservices.	
Deployment	Easy to deploy, only need to deploy one application.	Orchestrating the deployment of multiple services is more complicated.
Testing	May be harder to test and deploy since modules within	Easier testing since we can test individual services independently

	monolith are tightly coupled.	of others (except integration tests).
Developer Workflow	More coordination required between developers.	Easier for developers to work on separate services independently.

Given the factors above, we chose to give higher weightage to Scalability and Availability. We believe any performance hit from using a microservice architecture will be minimal at worst and can be minimised by careful planning and smart design decisions such as using Data Transfer Objects to minimise API calls, designing services to be less dependent on each other to minimize API calls needed to perform complex actions. Additionally, we understand the importance of good UX design in face of slower load times, for example by adding in loading indicators to make waiting more tolerable for the user, lazy loading sections of the page etc.

5.1.3. S3 + Cloudfront Deployment

The Strawhats frontend is built using React and stored in an S3 bucket. We also utilise the CloudFront content delivery network to serve a cached version of the application from edge locations, ensuring a faster response for users.

5.1.4. API Gateway and Application Load Balancer

The API Gateway serves as an ingress to direct traffic to the correct listener on the Application Load Balancer. It also acts as a proxy for HTTPS traffic to HTTP endpoints.

The Application Load Balancer directs incoming traffic from the API gateway to respective tasks evenly using a round-robin algorithm. This ensures we can create more tasks to scale the application horizontally when needed.

The Application Load Balancer security group is configured to only allow inbound traffic on the port we use from a limited range of IP addresses. Allowing both whitelists and blacklists for inbound traffic is a low-hanging fruit when working towards a more secure application. ECS Cluster

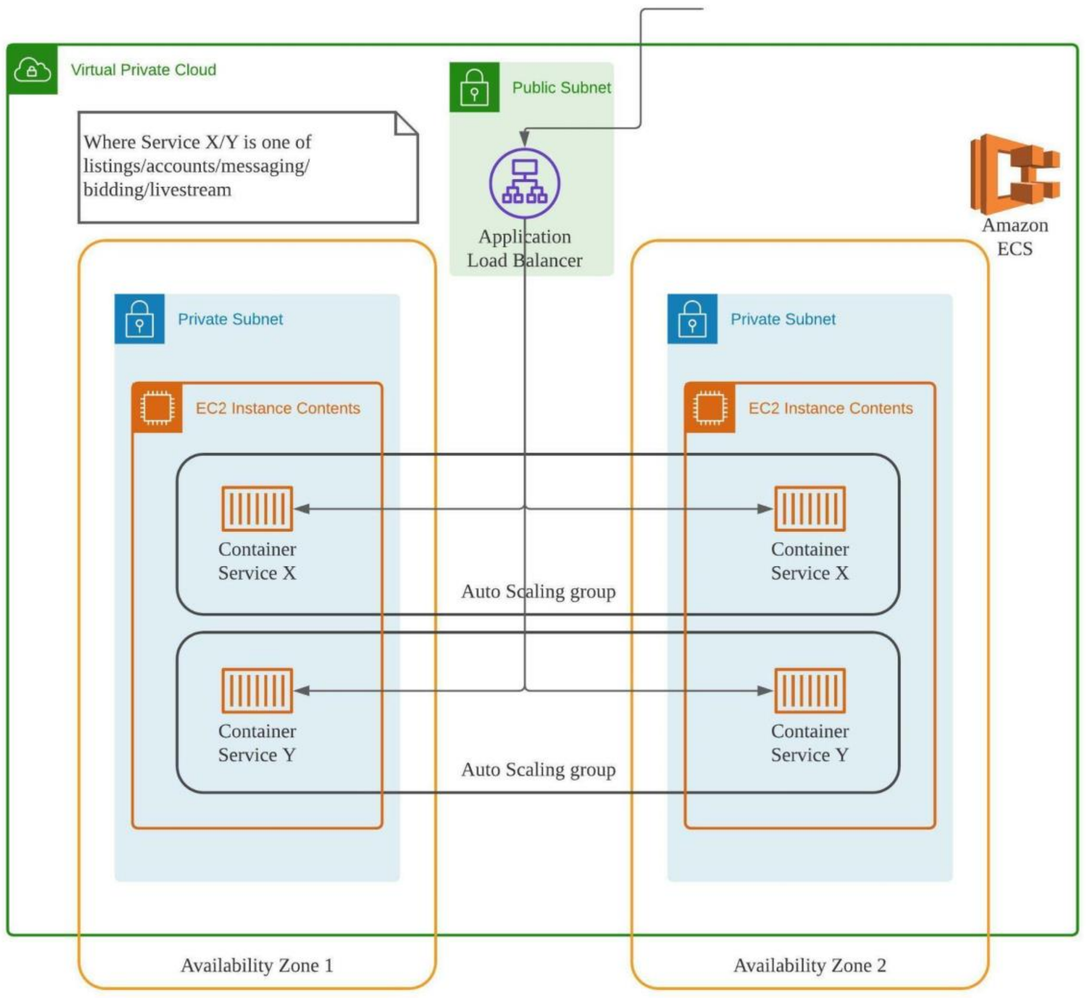


Figure 5-2: ECS Cluster Diagram

Amazon ECS is used as a container orchestration service to deploy, manage and scale the microservices.

To ensure availability, we have set up two different subnets within two different availability zones. This ensures that in the event of an outage, traffic can be directed to tasks in other subnets.

Each service is assigned to an EC2 Auto Scaling Group, with an auto-scaling policy that tracks CPU utilisation (90%). ECS manages the creation and deployment of new tasks when the current CPU utilisation rises above the target. Each task is also set up with its own health checks so that it can be automatically replaced should the container stop functioning.

The EC2 instances are also in their own Auto Scaling Group and can scale up or down when new tasks are created.

5.1.5. Socket.IO and ElastiCache for Redis

Strawhats makes heavy use of Socket.IO to enable bi-directional and event based communication between clients. We currently use Socket.IO in these services:

1) Messaging: To send and receive messages in real-time
2) Livestreaming: To load UI elements and inform user when a streamer has started livestreaming
3) Bidding: To update bids in real-time

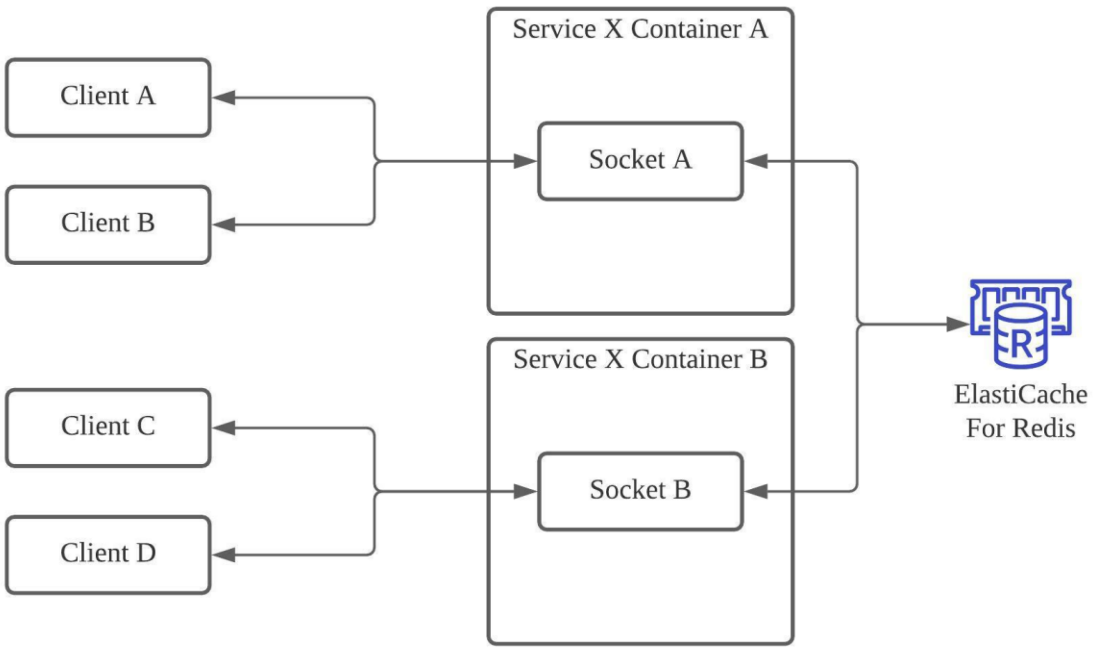


Figure 5-3: Socket Diagram

To allow Socket.IO to work with multiple containers (so that we can scale the service horizontally), we have implemented a Redis Adapter (using Elasticache for Redis) to allow packets sent by one container to be received by all other containers.

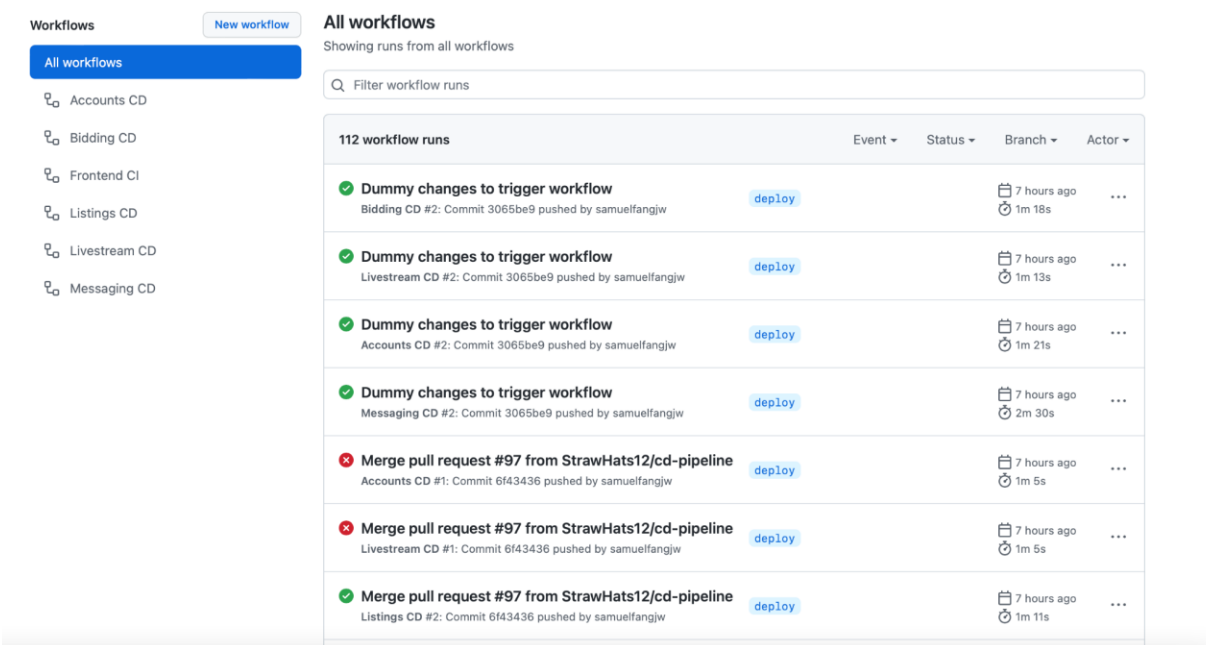
ElastiCache for Redis is configured with 2 read-only replica nodes, to ensure scalability and high availability.

5.2. DevOps

Strawhats has an integrated CI/CD pipeline to streamline the software delivery process. Both the frontend and backend (microservices) are deployed automatically.

5.2.1. GitHub Actions

GitHub Actions is used to automate our CI/CD.



112 workflow runs

Run	Branch	Time
✓ Dummy changes to trigger workflow Bidding CD #2: Commit 3065be9 pushed by samuelfangjw	deploy	7 hours ago 1m 18s
✓ Dummy changes to trigger workflow Livestream CD #2: Commit 3065be9 pushed by samuelfangjw	deploy	7 hours ago 1m 13s
✓ Dummy changes to trigger workflow Accounts CD #2: Commit 3065be9 pushed by samuelfangjw	deploy	7 hours ago 1m 21s
✓ Dummy changes to trigger workflow Messaging CD #2: Commit 3065be9 pushed by samuelfangjw	deploy	7 hours ago 2m 30s
✗ Merge pull request #97 from StrawHats12/cd-pipeline Accounts CD #1: Commit 6f43436 pushed by samuelfangjw	deploy	7 hours ago 1m 5s
✗ Merge pull request #97 from StrawHats12/cd-pipeline Livestream CD #1: Commit 6f43436 pushed by samuelfangjw	deploy	7 hours ago 1m 5s
✓ Merge pull request #97 from StrawHats12/cd-pipeline Listings CD #2: Commit 6f43436 pushed by samuelfangjw	deploy	7 hours ago 1m 11s

Figure 5-4: Github Actions Workflow

5.2.2. Branches and Workflow

We employed a feature branch workflow.

After being assigned an issue, developers will work on the feature/bugfix on a separate feature branch, named `<service>/<feature>`.

Once done, the developer submits a pull request. The pull request is automatically formatted using Prettier, an opinionated code formatter. GitHub Actions then runs automated tests to check for regressions. Once the pull request is reviewed (or if the change is minor and the changes are made known to others in the group), it is merged with the main branch, which acts as the staging branch.

The staging branch is deployed locally, and the team lead checks if there are any regressions or integration bugs. Once satisfied, the team lead fast-forwards the head of the deploy branch to the head of the main branch. This triggers the deployment of both the backend and frontend if necessary.

5.2.3. Backend Deployment

GitHub Actions are used to automate continuous deployment of the backend. The 'paths' variable is set to the appropriate path so that the deployment workflow only runs when there are changes to the corresponding service.

The workflow builds the docker container and uploads it to ECR. Once the ECS service has detected an update to the ECR repository, it will trigger a rolling update. This ensures that the service is always available, even while it is being updated.

5.2.4. Frontend Deployment

AWS Amplify detects a new commit in the deploy branch. If there are changes to the frontend, it clones the repository and builds the React Application. The application is stored in an S3 bucket and distributed via the CloudFront CDN edge network.

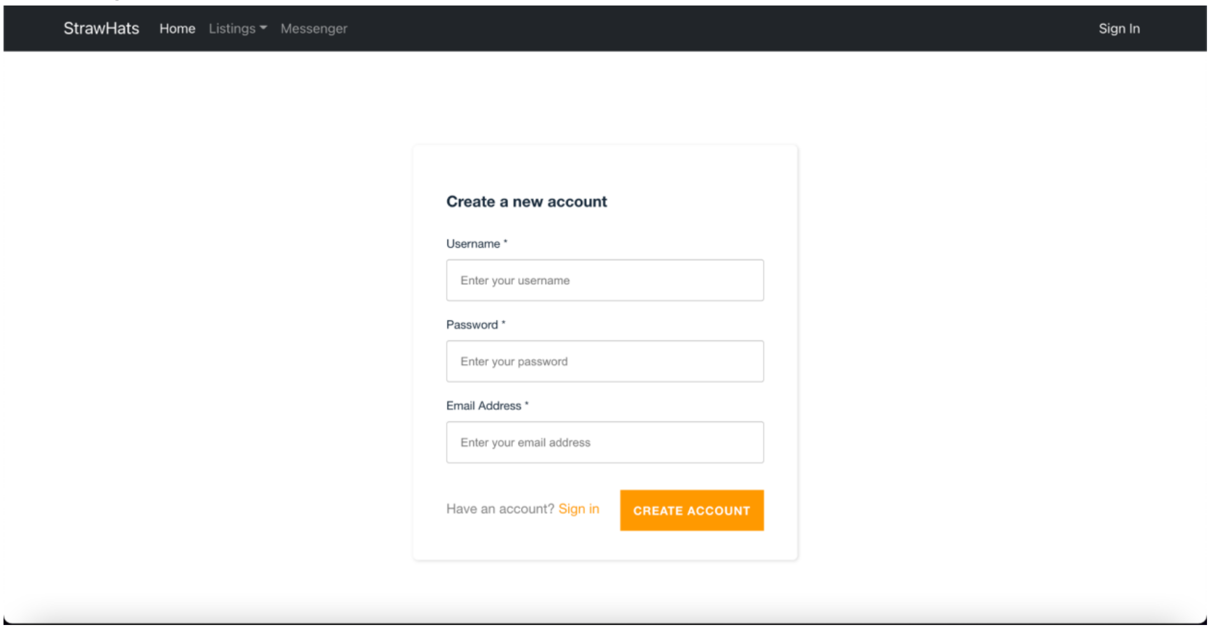
5.3. Authentication and Authorization

StrawHats uses a flexible and extensible system built upon AWS Cognito to handle user authentication and authorization. AWS Cognito is made out of two services, User Pools (for Authentication and Authorization) and Identity Pools (for Authorization).

5.3.1. Authentication

Step 1: User Registration

User registers for an account with the User Pool



Create a new account

Username *

Enter your username

Password *

Enter your password

Email Address *

Enter your email address

Have an account? Sign in

Figure 5-5: Snapshot of Account Creation Page

Step 2: User Verification
Users verify their account with the User Pool using a verification code sent by email to the users email account.

Step 3: User Authentication
User logs in to the application by providing the correct username and password. Upon successful login, the User Pool creates a Session and returns an ID token and access token (JSON Web Token) containing claims about the user's identity (as well as the corresponding refresh token).

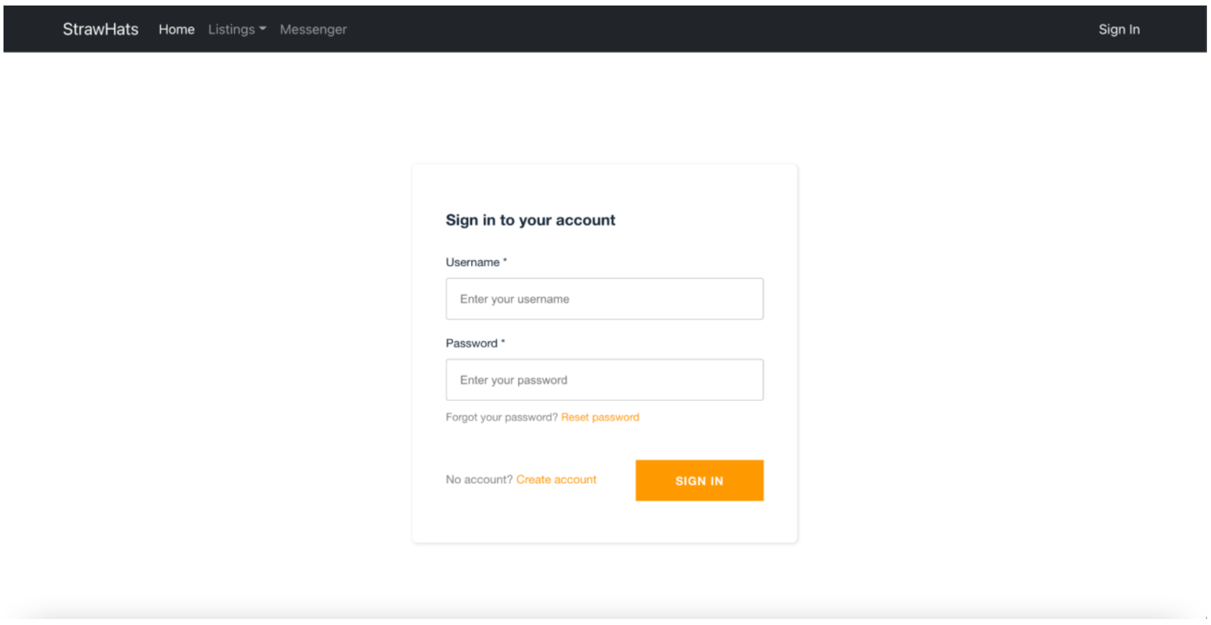


Figure 5-6: Snapshot of Login Page

Using these services allows us to automatically handle password recovery and other common account-related use-cases.

5.3.2. Authorization

A user can belong to one of two groups: user or admin. In general, users are only able to modify resources they created, while admins can modify any resource. There are two main methods StrawHats uses to authorize users to access/modify resources.

Option 1: Exchanging for Temporary Credentials

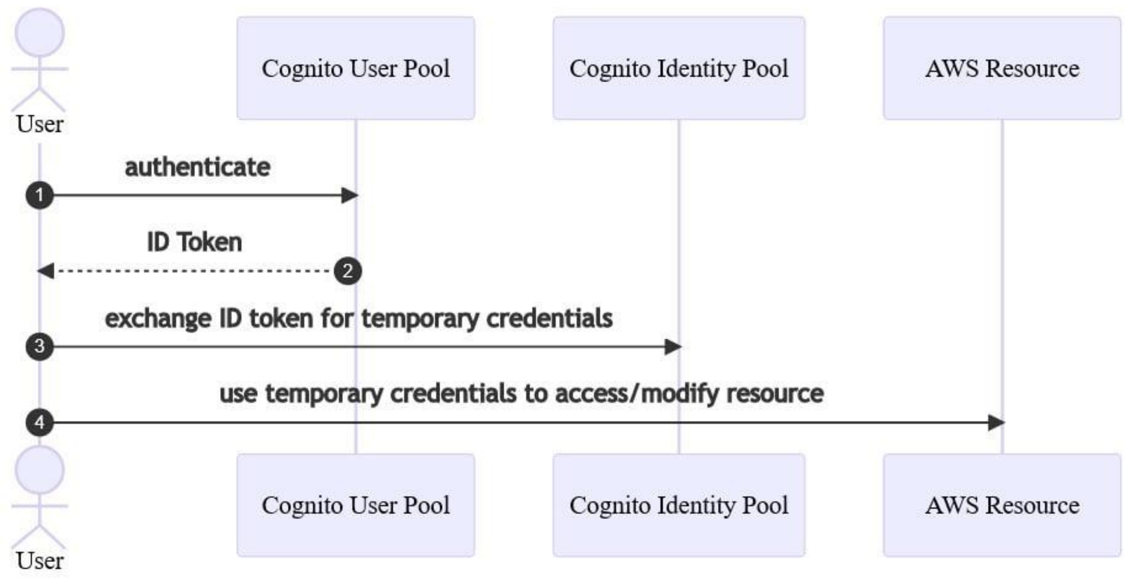

User
Cognito User Pool
Cognito Identity Pool
AWS Resource
1 authenticate
2 ID Token
3 exchange ID token for temporary credentials
4 use temporary credentials to access/modify resource

Figure 5-7: Sequence Diagram of Authorization Service

For access to certain resources (such as viewing images stored in the S3 bucket), the access token is used to get temporary credentials from the Identity Pool. The resources that can be accessed by the user depends on the IAM role assigned with the user group the user belongs to.

Unauthenticated users are also able to gain temporary read only credentials from the Identity Pool without an access token.

Option 2: Verifying JSON Web Token

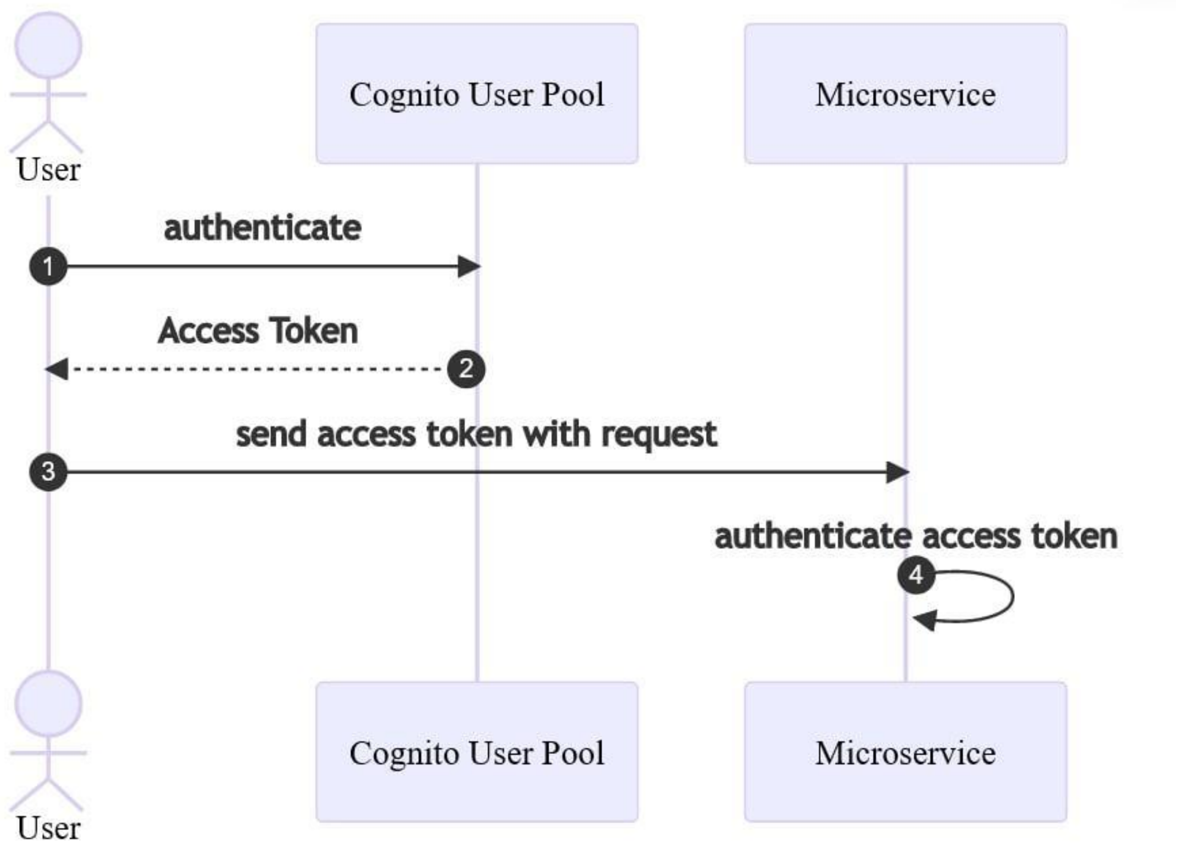


Figure 5-8: Sequence Diagram of JWT Verification

The access token is passed together with the header of REST API calls. The JWT signature is verified using the public key to verify that the token has not been tampered with. Logic within the application determines if the user is authorised to access the resource (based on unique user id as well as the user's role).

5.3.3. User Roles

StrawHats currently supports two user roles, 'user' and 'admin'. All users are automatically added to the 'user' user group. Admins are added to the admin user group manually via the AWS console.

If required, more roles can be easily added in the future with minor tweaks to each microservice.

5.4. Features

Our project's features can be categorised by the following distinct microservices, each with special design and architecture considerations:

5.4.1. Accounts

The accounts service handles all user account related requests. This can roughly be split into two categories:

1) User Profile - the users name, username, profile description and profile image
2) User Reviews - reviews that other users have left for a user.

The service handles CRUD operations for the above-mentioned categories. Users can only modify their content (except admins). Authentication is handled via JSON Web Tokens, explained above.

Creating an account object

Once a user has successfully registered for an account and verified his email, an AWS Lambda function is called, which creates a new Account object in DynamoDb as well as adds the user to the correct user group in AWS Cognito.

Editing a profile

When a user edits a profile in the frontend, a new accounts object is created. Once a user saves the object two events happen sequentially.

1. If the user has uploaded a new profile image, the image is saved to an S3 bucket tied to the user's ID. Images in this bucket can only be modified by the user and administrators. Access to this bucket is controlled by IAM policies (see section on Authorization above). A unique URI is returned, and this URI is saved in the accounts object. If required, previous images are deleted.

2. A POST request is sent to the account's endpoint with the new object. At this point, the application authorizes the update of the existing accounts object. Even though the user has been authenticated in the frontend already, we still have to authenticate again as a malicious actor might call the API directly. The object is also validated once more, to ensure that there are no errors.

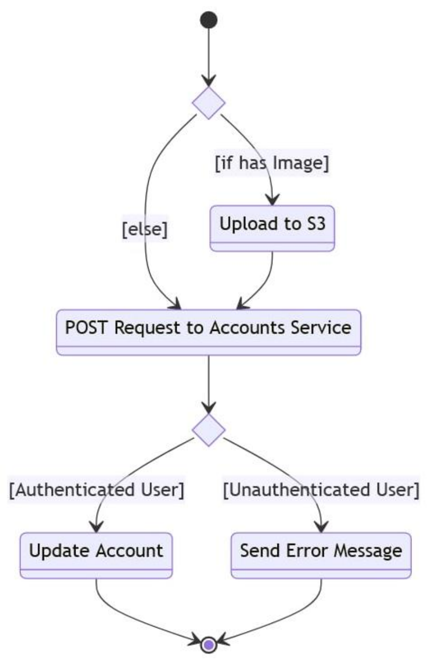


Figure 5-9: Activity Diagram of JWT Verification

Retrieving/Deleting an account

Frontend makes a REST API call to the accounts microservice to retrieve or delete an account, given that the user has the correct permissions.

Design Decisions

Handling login requests on the frontend	
Issue	We could have handled login requests in the accounts microservice instead. This would result in better separation of concerns, as the frontend will not have to concern itself with authentication anymore (will only need to render related UI elements).
Our Design	We chose to authenticate users on the frontend as it provides an extra layer of security (even though traffic to the api endpoint is encrypted). Since Cognito authenticates using the Secure Remote Password Protocol, which does not require the user's password to be sent with the API call, we can ensure that the password never leaves the frontend. This helps mitigate any possible man in the middle attacks. In addition, this reduces the dependency of other services on the frontend. Users can still authenticate (and use other services) even if the accounts service is down.

Username cannot be edited	
Issue	By default, Cognito username cannot be edited.
Our Design	We implemented a name field to our accounts. Users can modify the name field if needed. This name field can be displayed instead of the username on the frontend. This way, users can change the name that is displayed to other users without changing their actual username.

5.4.2. Listings

The Listings service handles all listing related requests. A listing contains the product information displayed on the listings page, such as product name, description, image URI and seller information.

CRUD Operations

The CRUD operations are like accounts service and omitted here for simplicity. The only difference is that to create a listings object, users send a POST request to the API instead. The behaviour of this is like the behaviour when editing an account/listing.

Filtered Get Requests

We have implemented endpoints that fetch listings by certain criteria (e.g. all listings whose seller is a certain user). The behaviour of these endpoints is similar to the behaviour of the endpoint that returns all listings.

Design Decisions

Separating Biddings Service from Listing's service	
Issue	Biddings object is dependent on listings object (as they share the same primary key), so it would make sense to combine them.
Our Design	We chose to separate these two services because we expect bids to be modified (and accessed) more frequently than listings. Users also expect bids to be updated on the UI in real time, whereas it is usually acceptable for changes to listings to only be shown when the page is refreshed. If we were to combine them, we would have to fetch a fairly large listing everytime the bid updates. By separating them, we can update just the bids and fetch only the required information when there are updates. In addition, we could potentially add a redis cache for the bidding service to reduce the number of read requests to DynamoDB.

Using primary key of account object as primary key of bidding object	
Issue	The alternative would have been to add a foreign key that identifies the bidding object. This could be better, as the two objects will no longer be as tightly coupled. We could change the primary key of the bidding object without changing the primary key

	of the accounts object.
Our Design	We chose to use the primary key for both, as it will speed up loading time and potentially reduce the number of calls to the backend. With just the listings_id, we can make calls to get both the listings service and biddings service in parallel. In addition, if the listings object is not needed (such as in the My Bids page), we do not have to make an additional call to the listings API.

5.4.3. Livestream

Background

Modern media streaming is complex and relies on multiple protocols and livestream adds extra challenges of managing latency to provide a real-time experience.

A rudimentary series of steps on how video is streamed by a streamer and viewed is as follows:

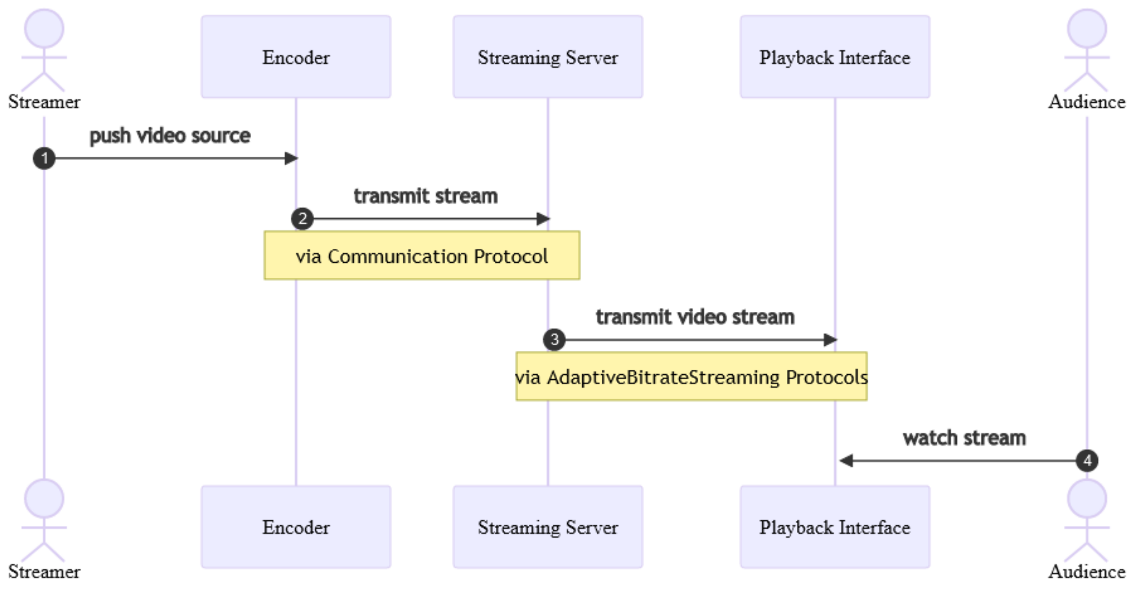


Figure 5-10: Sequence Diagram for High Level Flow of Livestream

In short, a stream is produced via an encoder and is ingested by a Transcoding system, i.e. a Streaming Server. The server can then produce video streams via a HTTP-based Adaptive Bitrate Protocol (ABR). This means that the streaming server can support a variety of video sizes, resolutions, network-speeds and other related factors to give a consistent video stream to the audience. The Streaming Server as per this diagram also has the responsibility of distributing the streams. The playback interface may be anything that supports the decoding of the streaming protocol (e.g. VLC media player, or browser-embedded media players).

For our use case, the communication protocol we chose was the Real Time Messaging Protocol (RTMP) and HLS was the Streaming Protocol chosen. A discussion on the choice of protocols and other alternatives can be found here.

Current Implementation

Currently, the sequence of interactions between our various components for livestreaming can be summarized via the followed sequence diagram:

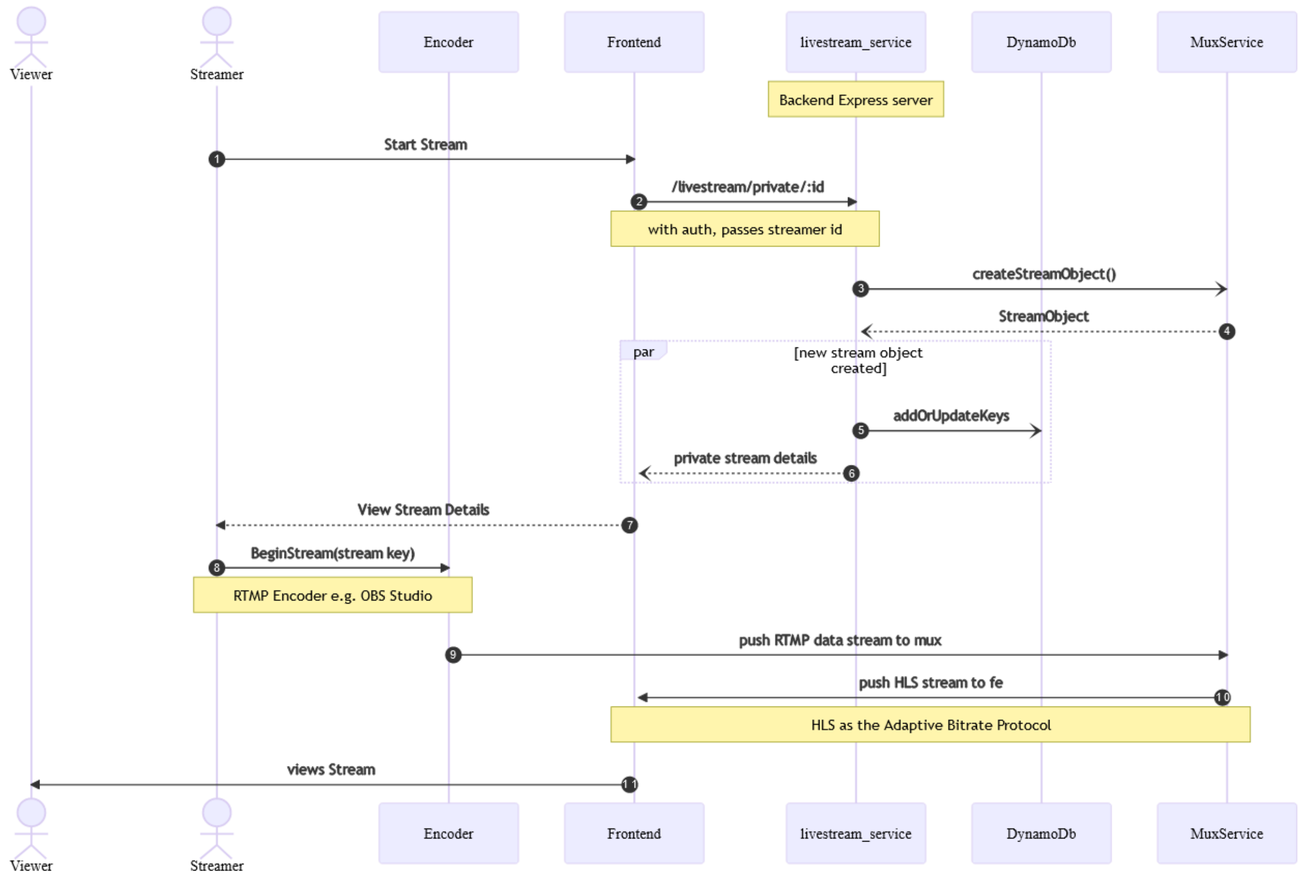


Figure 5-11: Sequence Diagram for Livestream

Summary of Steps in Creating Streams:

1. Streamer initiates the creation of a new stream by interacting with the front end streamer control panel:
2. Front end calls the livestream_service APIs (our backend server running on Express) and passes the required streamer id
3. Livestream_service then calls Mux's APIs to create a new stream asset
4. Upon successful creation, a stream object with assets such as playback ids (which can be used to fetch the video stream for playback) will be passed to the livestream_service.

5. The livestream_service then updates these keys on the DynamoDb database. This low-latency persistence of stream-details for every streamer is necessary to provide a common source of information across multiple containers in our deployment.
6. In parallel with step 5, private stream details are passed to the frontend as a response to the request made in step 2.
7. The streamer can now view private stream details. Most importantly, the viewer is given a private stream key that allows the pushing of video streams to MUX.
8. The streamer then selects their choice encoder (ffmpeg and OBS studio are some of the options) and provides their private stream key to begin streaming. In our case, we recommended using OBS Studio since that is the de-facto solution for streaming (e.g. on twitch).
9. The Encoder directly produces RTMP packets for the streaming server to consume. Upon consumption, the streaming server then does post-processing (e.g. establishing multiple streamlets to support Adaptive Bitrate Streams).
10. The livestream server, when requested by the front end application, can then provide HLS video streams required for playback. Here, our front end application relies on HLS.js, an NPM package that embeds a HLS video player within our StreamViewer component.
11. The viewer now can watch the livestream.

Some noteworthy points:

- Creating a Stream (initiated by step 1) and making that stream live (step 8) are two separate processes. A stream lifecycle, for simplicity, may be described to be the following: create, idle, live, deleted/destroyed.
- In step 2, only an authenticated user is authorized to create a new livestream corresponding to them and may not configure any other users' livestreams.
- We have used MUX Video as our streaming server solution. A discussion on why it was used and other alternatives can be found here.
- livestream_service is our backend application that manages the business logic for streams and interfaces with MUX. This express server acts as an adapter to our own APIs required and the functionality that MUX provides.

Choice of Application/Transport Protocols

The choice of a livestream solution depends on two aspects: producing a livestream (the Communication Protocol) and how that stream is distributed to viewers (the Streaming Protocol). One may implement WebRTC based streaming or RTMP based streaming and these solutions and their deployments vary greatly.

Choice of Communication Protocol

Comparing WebRTC vs RTMP as our choice solution for producing streams involves selecting protocols from different layers of the OSI model.

Consideration	WebRTC-based Streaming	RTMP-based Streaming
OSI Layer	Application Layer	Transport Layer
Supported Platforms	WebRTC is an application layer protocol that relies on the browser. Should the project require the development of other applications (e.g. mobile apps) in the future, such extensibility will not be there.	RTMP is a communication protocol that sits on TCP as part of the Transport Layer. This makes it application-agnostic since this lower level of abstraction allows us to broadcast streams in our Application Layer protocol of choice.
Coupling between Transcoding and Edge Distribution	Choosing this means that the delivery of streams and the production of streams will be tightly coupled and be handled by the same protocol.	There's no coupling between the two. On the flipside, a lower abstraction level requires more configuration when implementing and adds to development complexity.
Traditional Use Cases	As a real-time communication protocol developed for low-latency experiences. Involves direct application-to-application communication without the involvement of a server as a service registry. Google Meets and such bi-directional real time communications are the traditional use cases for WebRTC.	Developed specifically for streaming over the internet, this protocol is intended for one-way broadcasting of video and audio streams and is created specifically for livestreaming. Twitch, the current gold-standard of live-stream implementations, uses RTMP under the hood.
Network Patterns	WebRTC is traditionally peer-to-peer.	RTMP is used for a one-way of video and audio to be ingested by an RTMP server.

Scalability	Since we require a unidirectional transmission of video, WebRTC connections would be wasteful and unnecessarily taxing on network bandwidth. This has scalability concerns. For example, we tested the use of platforms such as BigBlueButton to implement webinar-style streaming and beyond 100 connections, the service becomes unusable.	Since stream producer and delivery will be handled by two different entities instead of a single source. This would be more bandwidth-friendly since a stream producer only needs to push a stream to a single stream server.
Available Frameworks	General purpose WebRTC server frameworks like Janus are viable options to implement a gateway for viewers to connect to and support broadcasting of information.	Multiple options exist. We have tested implementing a Nginx RTMP streaming server as well as utilizing MuxVideo as an infrastructure-as-a-service solution.

Eventually, the push factors against WebRTC were the tight coupling between producing streams and delivering them. Scalability is a huge concern for WebRTC implementations and for creating an auction-site that includes livestreams, delivering low-latency streams to at least 1000 viewers is a reasonable requirement to have.

Choice of Streaming Protocol

This choice deals with how the stream is distributed to clients and hence is only required to make if we choose RTMP-based streaming from the previous section. HTTP Adaptive Bitrate Streaming has two main camps of solutions: HLS or MPEG-DASH. While both are equally reliable, the key difference is that DASH is newer and non-proprietary than HLS. We chose HLS because being older, it has more support by video players. Compatibility is the biggest pull factor towards HLS because of in-browser players as well as native support for iOS should we aim to extend Strawhats' support to iOS apps.

Choice of Implementation

Now that we have articulated the use of RTMP Streaming for our communication protocol and use of HLS as our streaming protocol, here were the two main solutions we had at hand:

1. Implementing an Nginx RTMP Server. The implementation relies on containerizing 3 servers: RTMP server that ingests the RTMP stream from the viewer (or a Broadcast Server for retained Video on Demand) followed by a Streaming Server that is responsible for fetching fragments from a persistence volume when a Viewing client requests for a stream. The following architecture diagram describes the necessary components for such an implementation.

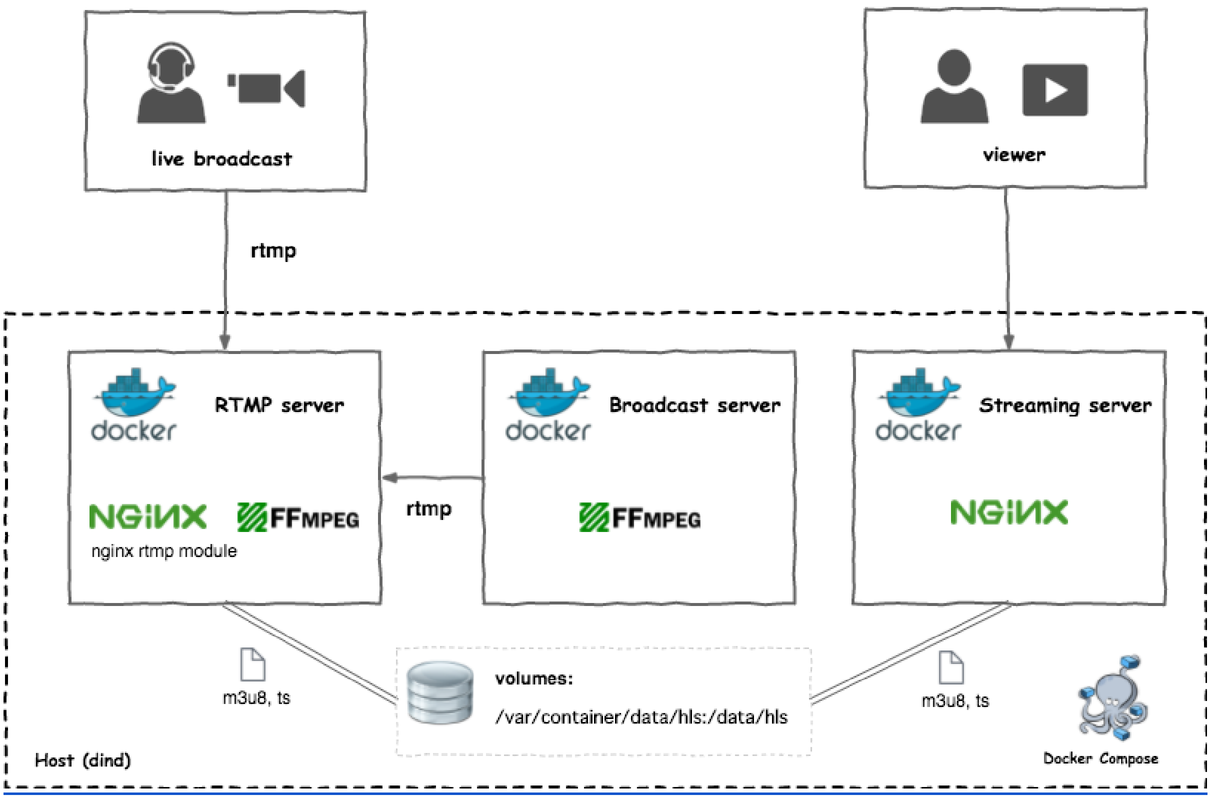


Figure 5-12: Sample Diagram of Livestreaming from Nginx

2. Utilizing Mux Video as the infrastructure-as-a-service tool, as shown in our sequence diagram earlier.

Consideration	NGINX RTMP Server	Utilizing Mux Video APIs
Supported Formats	Nginx has a wider variety of supported file types, so both HLS and DASH can be supported by the streaming server.	Mux primarily supports HLS only.
Implementation nuances	● If we stored stream fragments in the same container as the rtmp server, then a global service registry that would have been required to map the stream to the correct container instance ● Otherwise, it would have added another instance of transmitting data fragments from the RTMP server to the central store (e.g. S3 bucket), adding another point of failure or source of bottleneck	● Generally hassle free because it's an infrastructure-as-a-service so they handle the nuanced implementations under the hood. The only thing to implement would be the adaptors that utilize Mux APIs and the only persistence required would be to have the DynamoDb instance that stores stream details ● Replaying of old streams comes out of the box
Availability & Points of Failure	With more self-configured components, each component would have to be tweaked to minimise latency, jitter and packet loss	Relies on trust in the mux platform. Use of DynamoDb for persistence ensures that
Practical Deployment Considerations	● Cheaper to host in a similar fashion to the main application. ● Familiarity with this problem domain: developers are new to this problem domain and this novelty mean little guarantee of meeting	● While more expensive, it's a pay-per-minute pricing model that makes it a viable option for an actual production-level auction site. Furthermore, content streaming platforms like Vimeo and Udemy utilize Mux so it gives added

	project deliverables	confidence.

Essentially, the decision boils down to implementing the RTMP server ourselves or adopting an Infrastructure-as-a-service. While we successfully managed to set up the server, along with custom Nginx binaries and nginx config (relevant Proof-of-concept PR), unfamiliarity with deployment considerations motivated us to use Mux.

5.4.4. Messaging

The messaging service handles all communication related actions between users. It consists of conversation objects, where each conversation is shared between two parties. It also consists of message objects, whereby each message object is linked to a conversation and a sender.

Sending messages

The messaging service uses Socket.IO to enable real time communication between two users. The message data is also persisted and stored inside DynamoDB for future retrieval. Each conversation channel is identified by a unique identifier, `convo_id`. Once a user opens a conversation, it will listen to the socket with this `convo_id`.

The messaging service uses the publisher/subscriber design pattern to send messages between the involved parties.

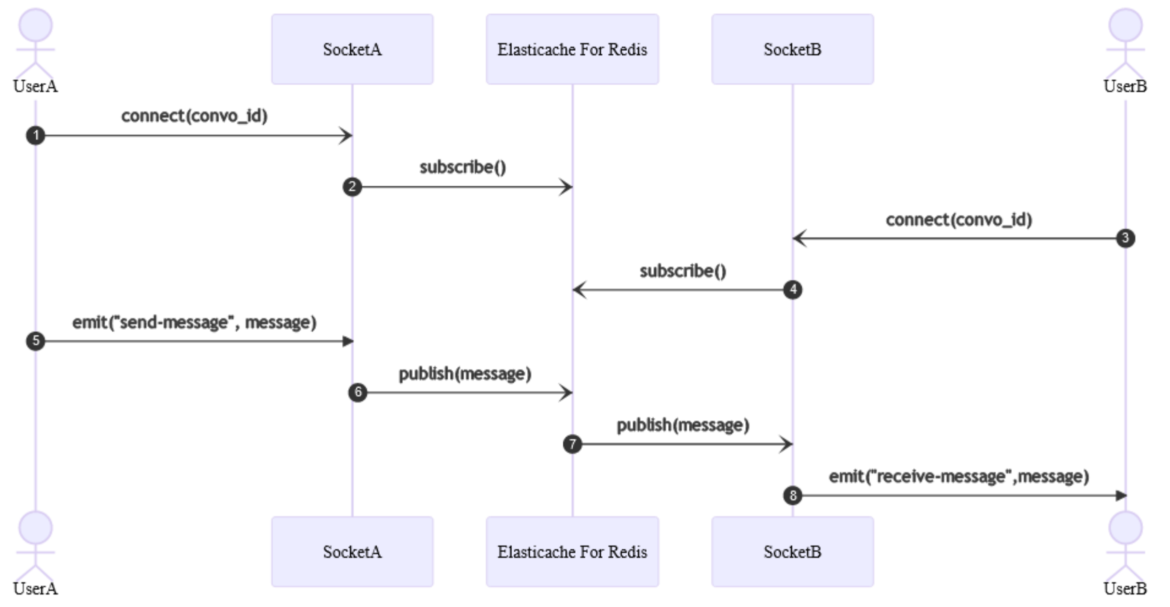


Figure 5-13: Sequence Diagram for displaying interactions between users, sockets and Elasticache

When client A opens a conversation, it will fetch the preexisting messages of this conversation from DynamoDB. Additionally, it will connect to socket A and listen to events belonging to this particular `convo_id` as seen in the figure above. The same is done for client B to socket B when he opens the same conversation.

As mentioned before, the sockets connect to the Redis adapter, so that for every message that is being broadcasted or emitted, it is sent to all matching clients connected to the current server and also published in the Redis channel and received by the other Socket.IO servers of the cluster.

When client A sends a message, it will emit a “send-message” event, which sends an object containing the message and the `convo_id` to socket A. Socket A will also publish it to the Redis channel. Since socket B is subscribed to the same Redis channel, the message is emitted back to client B via a “receive-message” event. Hence, client B can receive the message from client A in real time.

Security is ensured such that even if there was another client C, he is not able to eavesdrop on the messages exchanged since a Socket.IO connection would not have been established.

<u>Design Decisions</u>

Calling backend API and emitting events to socket separately from frontend	
Issue	Since sending a message requires it to be persisted in the database and also emitted to the socket, an alternative would be to have the frontend emitting event to the socket and as a result , this triggers the side-effect where the socket calls the backend API. Currently, the frontend calls both emit event and backend API upon a message send as two separate actions instead of designing it as a side-effect.
Our design	We decided to let the frontend handle both as this would allow messages to be persisted in the event of a socket outage. Hence, even if the socket were to be down, it would ensure that the message would still be sent and received by the other party.

5.4.5. Bidding

The bidding services, like the messaging service, leverages on Socket.IO to ensure that the biddings made by any users are communicated real time. The sockets are identified by the `listingId`, hence, all users who are viewing the same particular listing would be listening to one another through the socket.

In the listing socket, whenever a user places a bid, the socket then listens to "add-bid" and receives the new bidding object with all the information of the new bid. Thereafter, it broadcasts to all the listening sockets "receive-new-bid" alongside the new bidding object. Hence, all other users receive the new bidding made by another bidder in real time.

Users who receive the emitting newBid from the server's socket will then render the newBid in the existing array of Bids. Hence, allowing them to view the newest bid. The process flow as described above is illustrated in the sequence diagram Figure 4.13 below.

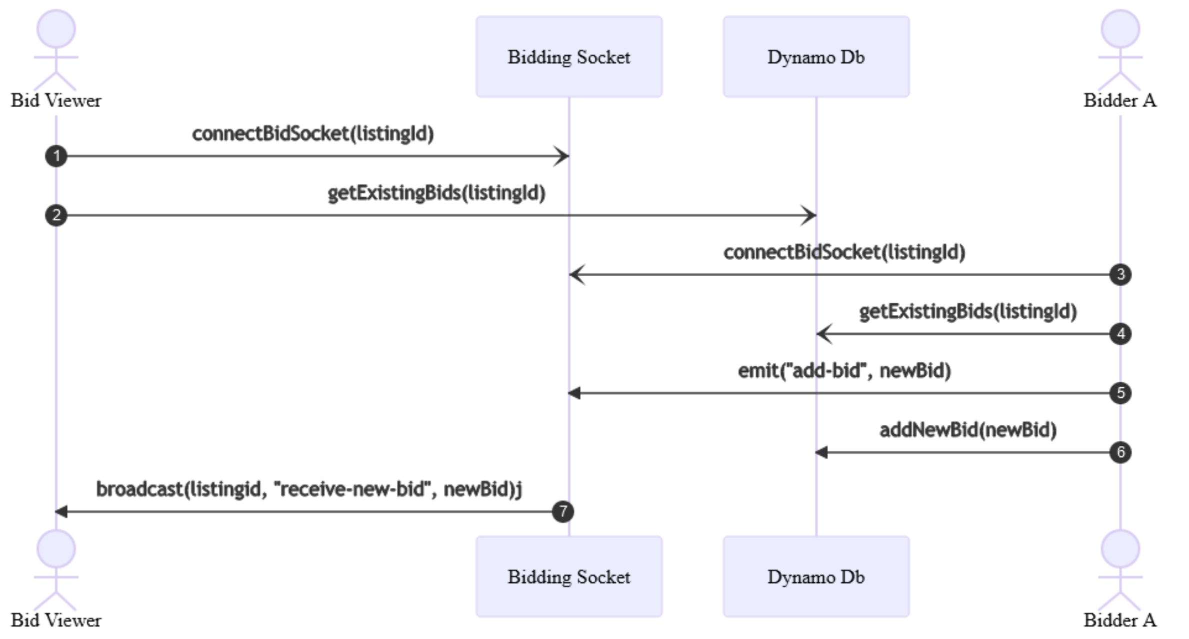

Bid Viewer
Bidding Socket
Dynamo Db
Bidder A
connectBidSocket(listingId)
getExistingBids(listingId)
connectBidSocket(listingId)
getExistingBids(listingId)
emit("add-bid", newBid)
addNewBid(newBid)
broadcast(listingid, "receive-new-bid", newBid)j

Figure 5-14: Sequence Diagram for showing interaction between frontend and sockets

For unauthorized users who do not have an account with Strawhats, they are still able to view the listings available as well as the bids made for the particular listing. However, they would not be able to place a bid. Only authorized users who have a registered account would be able to do so.

The following diagram shows the architecture design for the Bidding Service. Apart from the usage of Socket.IO, the bidding microservice also stores the bids in Amazon DynamoDB for the bidding data to persist. This allows any new user viewing any particular listing to be able to view the ongoing or past bids. The microservice's activity diagram is as shown below Figure 4.14.

The security for the addition of bids is ensured through the backend service where the user is authenticated before being able to successfully bid for a particular listing.

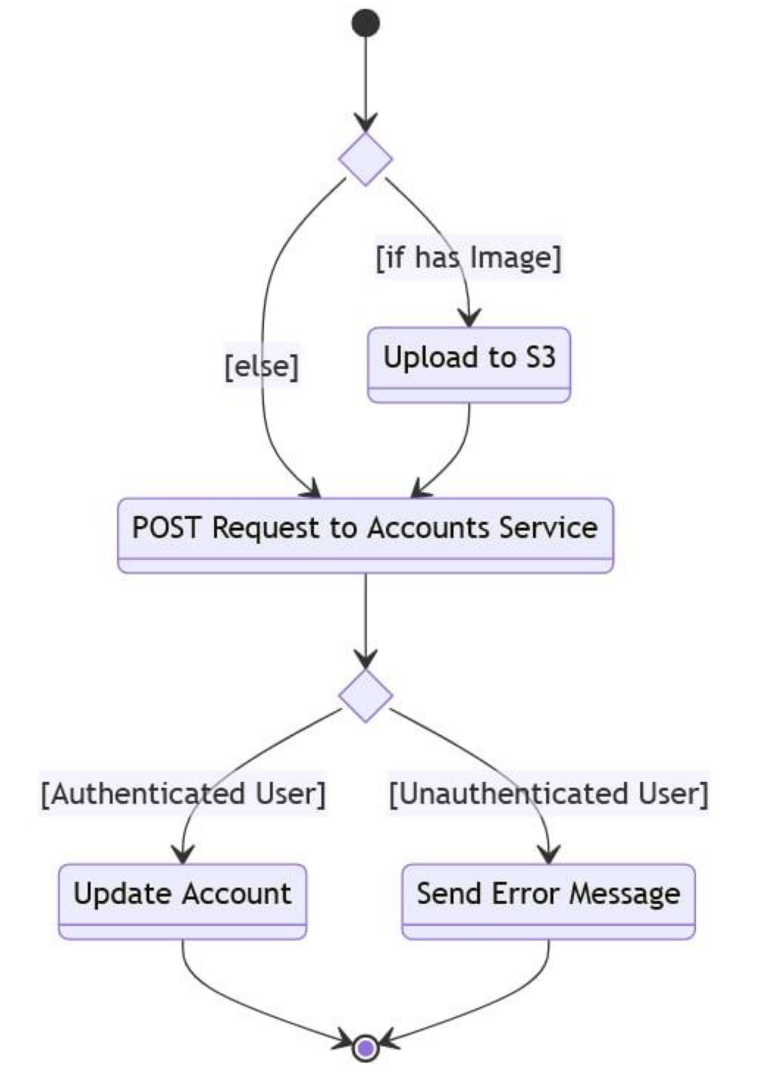


Figure 5-15: Activity Diagram for Bidding Microservice

5.5. Frontend

The frontend is built on the React framework. React is a free and open-source component-based JavaScript front-end framework that allows developers to write code in a declaratively manner.

We used the React-Bootstrap to style and design our UI components. This allows Strawhats to achieve a unified look, even though different pages were designed by different developers. We strictly enforce the use of React-Bootstrap classes and only allow other packages when no suitable React-Bootstrap alternative exists.

5.5.1. Design Decisions

Not using Redux or similar libraries	
Issue	Redux is an extremely popular library commonly used with React to manage and centralise application state. We did not use Redux in our application.
Our design	We decided not to use Redux as it makes it harder to reuse components. Since state that is passed between components is stored in the Redux store, components are tightly coupled with the Redux store (which acts as a singleton). Components that are dependent on a global state also means that they are not properly encapsulated, which makes them harder to reuse and test. Often, the Redux store has to be modified if we want to add another instance of the component. We believe that not using the Redux store will make us more mindful about how we design our components to be properly encapsulated and reusable, which will pay off in the long run. For state that has to be shared across different components, we decided that custom Hooks and where necessary, the Context API were sufficient for our needs.

6. Future Improvements and Enhancements

6.1.1. Frontend

Real-time incoming bid notification

Whenever a bid is added, the table is re-rendered. We find that a stronger sense of urgency, as well as user experience can be offered if an alert is rendered upon addition of a bid. The alert notification can also reflect how popular the listing is and thus urges bidders to place their bidding.

Winning Bidder

At the end of the auction the winning bidder is denoted in the winner bid section in the listing page. Thereafter, the seller would be able to personally drop a message to the winner by clicking on the “Chat with Winner” button. However, we feel that the winning bidding process can be further refined by having the seller send an templatized automated message to the winner.

Messaging notification

As a messaging feature, having push notifications would be an ideal add-on that allows users to be instantly updated. Modern browser-based push notifications api are a viable way for us to add such notification features to alert when there’s an incoming message.

Listing FAQ

To simplify the auction process for the seller, we thought an improvement that could be incorporated would be a FAQ section in the listing page. This could reduce enquiry between bidders and sellers via the messaging function.

Regulating Shared Content

The content of messages in our messaging services needs to have some auto-sanitization. Currently, a malicious user may use our messenger as a vector to send malicious urls via chat. At some levels, chat content has to be regulated to keep users safe. Url checking apis such as Google’s safe browsing API are some options to consider.

6.1.2. Additional Services

Payment Service

Currently the winning bidder is denoted at the deadline of the auction. The seller then connects with the winner via the messaging functionality. We thought that we could have expanded further by introducing possible payment options, to simplify the payment process between the winner and sellers.

7. Reflections and Learning Points

7.1. Challenges Faced

Cloud Service Integrations

Understanding of AWS Services and documentation posed a challenge as we had no prior experience working with AWS. Furthermore, examples given on the documentation as well as on discussion forums (e.g StackOverflow) were at times outdated. The implementation took various trials and error efforts to get it functioning.

Our learning point here was that despite leveraging on AWS capabilities to support some functionalities of our application, we should not assume that the integration would be smooth.

Integration of services

As we all started working in our own individual services, there was some difficulty in integrating our own microservices into the front end as well as between the microservices. For example, the conversations held on the livestream are a result of the integration between the livestream and messaging microservices. The integration proved to be a challenge as we were not familiar with the functions and API calls in the other microservice. We managed to overcome this hurdle by filling each other in on our weekly scrums.

Our learning point here was that despite aiming to be as loosely coupled as possible with the usage of microservice architecture, we had to be mindful of communicating our implementation with one another.

Proofs of Concept and Analysis Paralysis

In face of the novel aspects of this project, we rightly allocated time to try out various existing solutions and technologies. This was useful to find reference projects to learn from and get acquainted with the problem domain. However, in instances like the livestream service, there were multiple solutions which differed greatly. To give an example, developing the WebRTC based solution would have been valid and there were references to follow but so would using Nginx. However, the expected development time taken was long and it would not have been easy to switch between the two solutions. The research phase took too long and development for livestream was backlogged the most due to such analysis paralysis. Thankfully, we had Mux as an infrastructure-as-a-service solution as the eventual backup solution that we resorted to. The takeaway for this is that we should always have multiple backup solutions to utilize a project backlog but at the same time utilize decision matrices to make judgements in face of analysis paralysis.

8. Appendix

8.1. Glossary

This section serves to provide a summary of definitions and descriptions of terms used in this report.

8.2. AWS Specific

Term	Meaning or Definition
AWS (Amazon Web Services)	Infrastructure as a service (IAAS) provider for scalable cloud computing services.
EC2 (Elastic Compute)	A web service that provides secure, resizable compute capacity in the cloud.
ECS (Elastic Container Service)	A fully managed container orchestration service that makes it easy for deployment, management, and scaling containerized applications. Similar to Kubernetes.
ECS Task Definition	The blueprint which describes how a docker container should launch.
ECS Task	The instantiation of a *task* definition within a cluster.
ECS Service	Allows you to run and maintain a specified number of instances of a task definition simultaneously in an Amazon *ECS* cluster.
S3	S3 is an object storage service that stores data as objects within buckets.
CloudFront	A global content delivery network (CDN) service built for high-speed, low-latency performance, security, and developer ease-of-use.
ALB (Application Load Balancer)	Automatically distributes your incoming traffic across multiple targets, such as EC2 instances, containers, and IP addresses, in one or more Availability Zones.
API Gateway	An AWS service for creating, publishing, maintaining, monitoring, and securing REST, HTTP, and WebSocket APIs at any scale.

Cognito	Provides authentication, authorization and user management services.
Elasticache	A managed caching service for Redis.
DynamoDB	A fully managed, serverless, key-value NoSQL database designed to run high-performance applications at any scale.
Amplify	A set of purpose-built tools and services such as ready-to-use components and code libraries that makes it quick and easy for developers to create apps in AWS.
ECR (Elastic Code Repository)	A fully managed Docker container registry that makes it easy to store, share, and deploy container images. Similar to Docker Hub.
AWS Lambda	AWS Lambda is a serverless, event-driven compute service that lets you run code in response to events.

8.3. Others

Term	Meaning or Definition
Socket.IO	A library that enables real-time, bidirectional and event-based communication between the browser and the server.
React	A free and open-source front-end JavaScript library for building user interfaces or UI components.
Prettier	An opinionated code formatter.
Redis	An open source (BSD licensed), in-memory data structure store, used as a database, cache, and message broker.
OSI Model	Open Systems Interconnection Model
ABR	Adaptive Bitrate Streaming, a technique that allows streaming of multimedia of different qualities simultaneously in anticipation of client-side connections.
WebRTC	Web Real-Time Communication**,** a real-time peer-to-peer communication protocol.
RTMP	Real Time Messaging Protocol, a transport layer protocol for real

	time audio and video communications.
HLS	HTTP-Livestreaming, an ABR protocol made by Apple Inc.
MPEG-DASH	Dynamic Adaptive Streaming over HTTP, another ABR protocol.