c18n

A Community Discussion Platform

S. No.	Contributor	Comments	Date	Reviewed By	Review Date
1	Jitender Singh	Introduction, Architecture dagram	Jul/25/2021		
2	Jitender Singh	User Registration, Authorisation, Discovery	Jul/25/2021		
3	Jitender Singh	Users APIs	Jul/25/2021		
4	Jitender Singh	Api details, web-socket implementation, lambda architecture, kafka producers consumers	Jul/26/2021		

Video Links

HQ videos version added here.

https://drive.google.com/drive/folders/1xDqRkB3oxdKKnD1HL7OuGxx9dSIQZ7LQ

Low quality versions also available here.

https://youtu.be/MLSZCdkTQ0M

https://youtu.be/sPMRsJIlfN8

Project code added at:

https://github.com/paljsingh/community-discussion

Introduction

The community discussion platform code named **c18n** (naming convention borrowed from other numeronyms like k8s, i18n, l10n etc.) is a community platform to support user discussions and direct messaging between users or user-groups.

The primary goal is provide a social network service where users across the globe can connect to each other, form communities based on their interest areas, create user-group with people they know directly or indirectly and engage in discussions, or send direct messages.

The **c18n** platform intends to build a strong user base by providing them with a rich feature set including (but not limited to) community creation, rich media posts containing text, images, audio, video and other interactive media, user discovery, likes, comments and shares, direct and group messaging, user discussion threads, custom profiles and building of user circles / friends networks.

As the user base is expected to grow to millions of users worldwide, the c18n platform aims to make use of a highly scalable, fault tolerant and low latency architecture by using various streaming and big data solutions.

For the proof of concept app for **c18n**, the focus area will be limited to a system that provides an end to end implementation of the community posts and direct user (or user-group) messaging. We also intend to provide a set of custom scripts to make it easy to generate the required data and a functional UI to visualise the user experience.
The PoC app is expected to have all the primary components running in their simplest configuration such that the entire **c18n** PoC app can be set up and run on a single node (or a single node cluster.)

High Level design

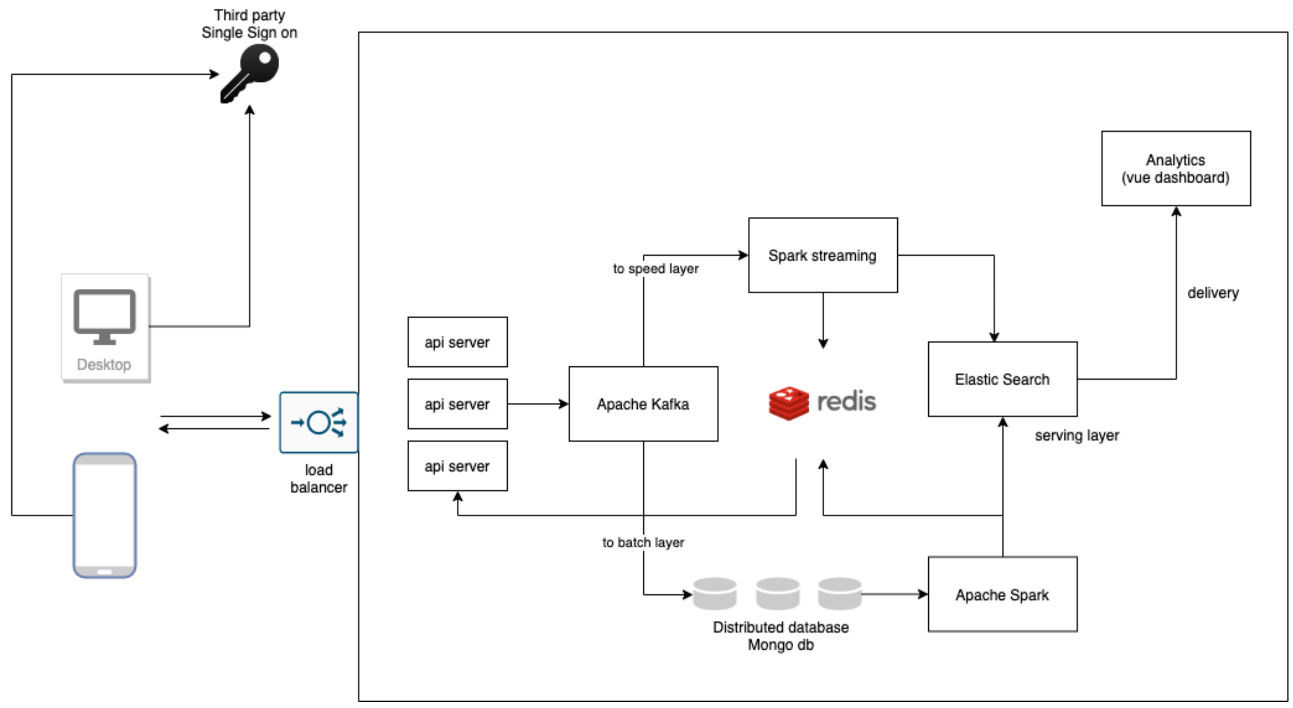

Fig. 1 - High Level Design

The figure above shows the proposed high level design diagram for the **c18n** app.
The design diagram primarily caters to the following use cases, and is subjected to changes with the introduction of new requirements / features.

- User registration
- User login
- User discovery

- Community creation
- Community discovery
- Joining a community
- Inviting users to join a community
- Accept / decline an invitation

- User-Group creation
- User-Group discovery
- Joining a user-group
- Inviting users to join a user-group
- Accept / Decline an invitation

- Community posts containing
  - text
  - images
  - videos
- Share a community post
- Comment on a community post
- Like a community post
- Direct message to a user
- Reply to a user message
- Message (broadcast) to a user-group

- Comment / Reply on a user-group message
- Like a user-group message

- Reporting / Analytics
- Ad hoc queries

The below section provides a brief introduction to the various components of the system -

Api Servers

The application is supposed to have multiple scalable api server components, each of components catering to one particular type of resource and its associated use cases.

The api servers for Users, Communities, User-Groups, Messaging etc will be deployed via kubernetes deployments and each one of the deployments can be scaled up / down independently, depending upon the incoming traffic volume.

We intend to use HTTP based apis for use-cases related to user authentication, creation of user-groups, communities and other non-heavy workflows, while use-cases requiring heavy media flow, longer connections and instant user notifications (e.g. messages, posts, likes, shares) shall be implemented via web-sockets.

The user facing servers shall be further split into application backend servers and frontend servers to make it easy to develop the two independently and separate the presentation layer from the backend apis. The api (backend) server thus will only accept requests from the frontend servers.

We intend to use python/flask for implementing the backend api servers, while the frontend shall be using Vue.js .

Message Queue - Apache Kafka

Apache Kafka shall be used to decouple the user facing servers from the core application components. In practice, some of the low volume apis may (for example community creation, user creation) may bypass Kafka and directly save the data to a database. This shall simplify the workflows that require events to be processed in specific order (for example, a community must be created first before posting a message to it.)
Further, the components responsible for creating a resource may emit an event like - “a new user has been created”, or “a new community has been created”, enabling the Kafka consumers to pick up and take appropriate action on it (e.g. we may want to build a list of users that may be interested in joining this community, and send them a **suggested community** notification.)

The api (backend) servers shall act as Kafka producers and shall push incoming event streams such as new messages / posts, likes, share to the Kafka server.

Append only distributed db - Mongo db

We shall make use of a distributed database for storing the event metadata (for example, text content / urls of posts and messages, likes, shares and other information related to creation and subscriptions of various communities, user-groups and their mappings to the users of the c18n platform.
We intend to use an append only database in accordance with the guidelines for implementing the lambda architecture, which shall accept data from Kafka to serve as the batch layer.

For the PoC, we plan to make use of a mongo db server owing to its ease of setup. For a real production case we may want to explore Hadoop hdfs/hbase or Cassandra.

Batch Layer - Kafka / Custom scripts

TBD - Not yet explored.
For generating the batch views, we may use custom queries triggered via external scripts or through Elastic search, or use a batch processing system like Apache Kafka.

Distributed File System - S3 / HDFS / GridFS

The media files posted by the users shall be saved into a distributed storage like S3 or Hadoop hdfs.
However, for the PoC, we may use mongo db with GridFS support to save on the cost and effort of setting up a dedicated distributed file storage.

Speed Layer - Kafka Streams / Spark Streams

We shall be using Spark streams or Kafka stream for the speed layer of the lambda architecture. This component shall be responsible for generating the speed view and save the view to the serving layer, (to be implemented via Elastic Search)
TBD - Yet to explore both the options and need to figure out which one is more suitable for the PoC.

Serving Layer - Elastic Search

We plan to use elastic search for combining the speed and batch views and serve the results to both the api servers and in turn the end users, and to the analytics dashboard.

Cache - Redis

In order to speed up the query and reduce the load on the database, we plan to use a redis cache server. Redis shall be used to hold the recent events data, for example currently logged in user's info, recently created communities, recent posts belonging to large user communities, or recents posts by users having large following.

RDBMS / Graph databases

A relational database may be used to hold user information, roles and access permissions. As we do not expect the total number of users to cross a few billions, a traditional RDBMS may work fine for this, however if there is a need to build a user - user relationship networks, followers / fans or a friends of friends network, we may want to explore a graph database for such cases.

For the PoC, any information related to dummy users shall be stored in the Mongo db itself.

Analytics Dashboard

We shall be using ElasticSearch for various reports and other ad-hoc queries. The Python / Flask api servers shall expose api endpoints to the frontend, and use search queries to fetch data from elastic search and serve the same to the frontend app.
The dashboard shall be implemented with Vue.js and accessible to the users having required roles / permissions (e.g. users with **admin** / **analytics** role).

In a production environment, the access to the analytics dashboard may further be restricted by the network / firewall policies, and by allowing access from the corporate VPNs only.

Design Components

The PoC app intends to use the following components **exposed via the api servers** -

Users (Registration, Authentication, Authorisation and discovery)

User registration can be done via a hosted user authentication service, which provides a form based interface for new users to supply user information (email / password / phone / username etc) and validate them via email or a secondary channel (eg. Sms / voice call)
In addition to this, c18n platform will also provide the option to use a third party authentication, so that users can use their existing accounts (Google / Facebook / okta etc) to join the c18n platform.

Once signed up and validated via email / phone, the users can authenticate themselves to the c18n platform and start accessing its services. Both the hosted and third party auth services will make use of JWT token to validate the authenticity of the users. The token may have an expiry period set to 7 days by default, and may need to re-login once the JWT token is expired.

All the users will be given a default **regular user** role. In future, the c18n platform may also support additional roles (say, **premium users**, **api developers** etc).
In addition to the above, there will be one of more users assigned with **administrator** roles, having access to various subsystems (e.g. analytics dashboard) and control permissions (user or content deletion/deactivation etc). A separate authorisation table shall be used to decide the user roles and permissions.

For the **c18n** PoC app, we are going to make use of okta third party authentication for signing up new users. In addition to this, the PoC app will also allow batch creation of dummy users via a user creation api.
For the PoC app, users signing up via okta are going to be treated as admin users and will have the permissions to use the analytics dashboard, batch create dummy users or impersonate other users.

In addition to this, a user discover api will allow users to search other users by name and any other public info.

Communities

The communities module will handle use cases related to creation, discovery and user mappings to a community.
Future scope of this module may also include features like assigning tags to communities and extend the community discovery / search by associated tags. Further, we may introduce the concept of transferring the community ownership to other users, providing one or more users with a moderator role and management of various community specific workflows (e.g. limiting the number of posts per user, block / kick out certain users, moderation of posts etc.)

Posts

The Posts module handles publishing / update / search of community posts. The posts returned for a community shall be default sorted by date, recent posts first. The application may provide any additional sorting and filtering methods (e.g. relevancy to user interests, unread posts only, posts from people you have interacted first etc.) to make for a better user experience.

Comments

The comments module shall handle the user comments / replies on a community postEach comment will have a parent that can be another comment, or a community post.
This shall provide a nested thread where multiple users can engage in discussion over a community post or a user-group message.

Likes

The likes module shall associate user likes / dislikes to a community post, user-group message or a comment. The events generated by this module may be used to associate a user with their interest areas.

Shares

The shares module will handle sharing of community posts to other communities. The application may want to maintain a lists communities to which a post has been shared.

User-Groups

The user-groups module shall handle creation, update and addition / removal of users to a user-group. Users in a user group may send messages that will be broadcasted to all the members of the user-group.
At the backend, the api servers shall create a Kafka topic partition for each of the User-Groups, users joining a user-group shall receive all the notifications / messages intended for that Kafka topic partition / user-group.

Messages

The messages sent in a user-group or directly to another user shall be handled by the Messages api. Direct user messages or broadcast messages to a user-group shall be implemented via web sockets. The frontend application / browser is expected to open a web socket with the api servers upon logging in a user, the web socket session shall be maintained till the user explicitly logs out, or the JWT token expires.

The api servers should check for the token expiry and must ensure it is not expired, before sending any communications over web socket channels to the end users.

As each of the user-groups is maintained as a Kafka topic partition, messages sent to a user-group shall be broadcasted to all its subscribed users. On the other hand direct user to user messages will be sent only to the intended recipient user.

We may also persist the messages to the database and provide a http/rest api to obtain the previous message history. The browser / frontend app is expected to use this api upon loading the user-group message or user - user message interface.

As there can be multiple api servers for messaging, each connecting to N number of end users via websockets, it becomes necessary to efficiently handle the message routing and ensure that the direct / broadcast notifications are sent to intended end users, and servers do not have to subscribe to each and every Kafka topic partitions.

We suggest the following simplified approach, as detailed in the diagram below -

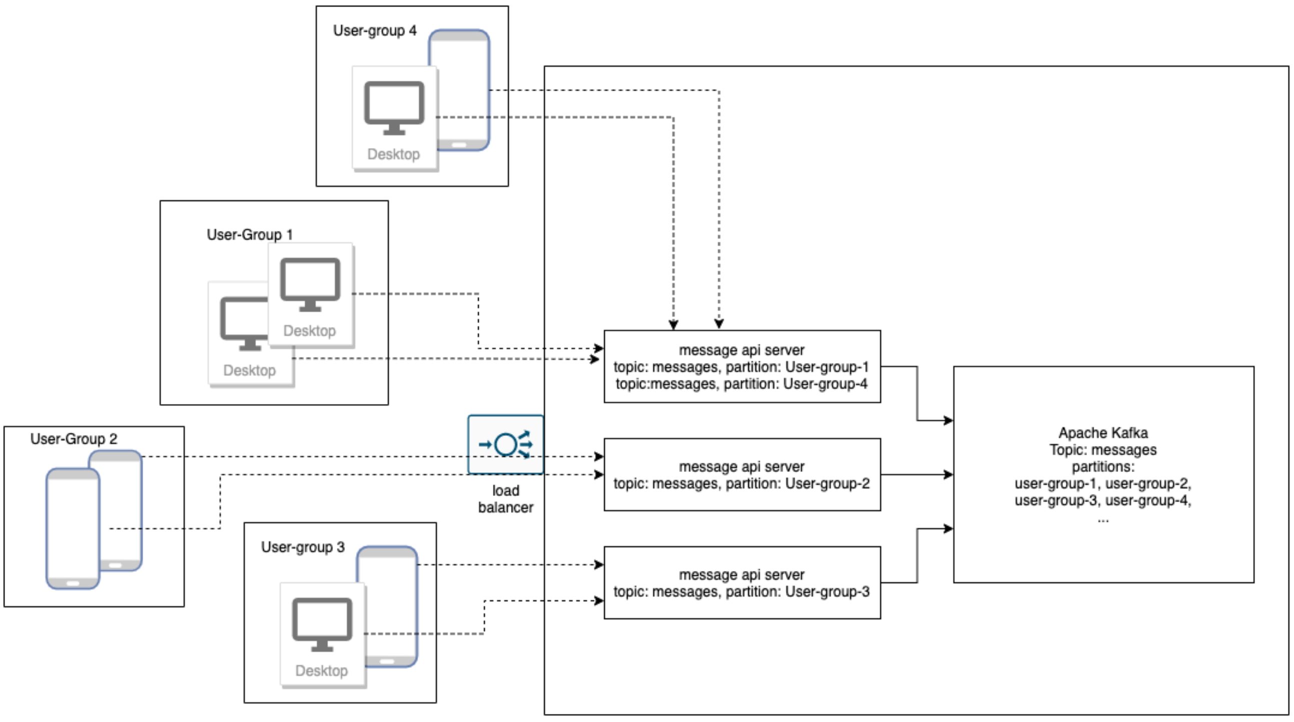
User-group 4
Desktop
User-Group 1
Desktop
Desktop
User-Group 2
load balancer
User-group 3
Desktop
message api server
topic: messages, partition: User-group-1
topic:messages, partition: User-group-4
message api server
topic: messages, partition: User-group-2
message api server
topic: messages, partition: User-group-3
Apache Kafka
Topic: messages
partitions:
user-group-1, user-group-2,
user-group-3, user-group-4,
...

Fig - 2 Messaging with multi-server multi user-group environment.

When creating a User-Group, one api server can connect to the Kafka topic partition corresponding to the user-group topic partition. This can be the same server that originated the user-group creation request (in case both messages and user-groups modules running on the same server) or a different randomly assigned message api server.

When the browser / frontend app loads the message interface, it shall be provided with the direct address of the api server responsible for handling the given topic partition (we may want to bypass the load balancer here.)
Once the client connects via web socket with the intended api server, it can start sending the messages. Any messages received for a Kafka topic partition will now be picked up by exactly one api server which will in turn be connected to all the clients using that user-group.

The message server now can make use of /**send** or /**receive** endpoints and accept incoming messages or broadcast the new messages to all the connected clients.

For direct messaging between two users shall also work in a similar way, i.e. a temporary topic partition is created when a user initiates a direct message, the originating api server may subscribe to the topic partition. The recipient user may be notified via any of the existing web socket connections (if online) or via http/rest api upon the next request. In both the cases, the recipient would be redirected to the appropriate message api server that is listening to the temporary topic partition. The following diagram illustrates the process -

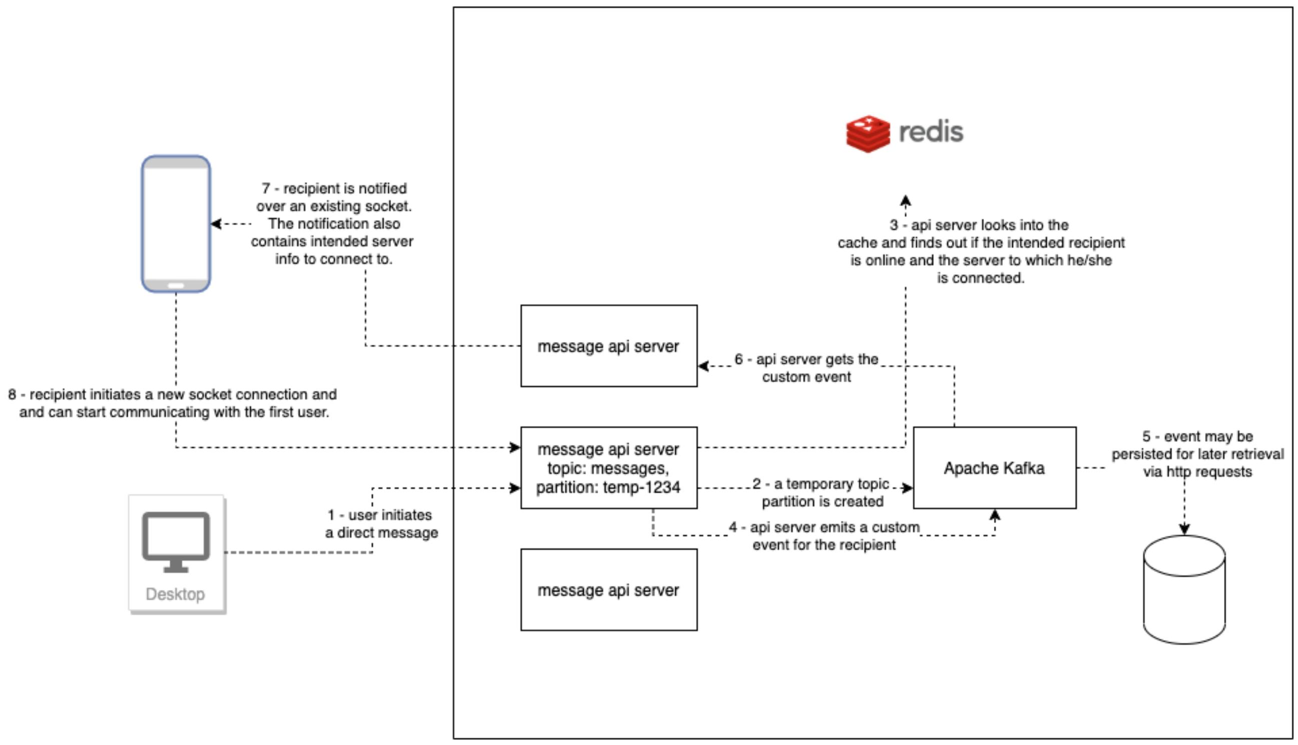
redis
3 - api server looks into the cache and finds out if the intended recipient is online and the server to which he/she is connected.
7 - recipient is notified over an existing socket. The notification also contains intended server info to connect to.
message api server
6 - api server gets the custom event
8 - recipient initiates a new socket connection and and can start communicating with the first user.
message api server topic: messages, partition: temp-1234
Apache Kafka
5 - event may be persisted for later retrieval via http requests
2 - a temporary topic partition is created
1 - user initiates a direct message
4 - api server emits a custom event for the recipient
Desktop
message api server

Fig 3 - A more detailed use case explaining direct user messaging in a scalable environment.

Apis, Schema and Data Flow

All the above listed component for the **c18n** platform are expected to use REST guidelines to implement their apis. We will use json as the primary format for data exchange between the different components as well as to/from end users.

The apis will use the format /api/<version>/<resource-pluralized-name>/[action if any]
Following the standard REST guidelines, we will be using POST requests for resource creation, PUT / PATCH requests for resource update, GET for retrieval, while DELETE shall be used for soft-deletion or deactivating a resource. (Related, see Security Concerns and Guidelines section.)

Below listed table presents the various apis associated to the components listed earlier, along with their sample request / response formats -

Users APIS

	Create a new user
Request	POST /api/v1/users/new Accept: application/json, text/plain, */* Accept-Encoding: gzip, deflate, br Connection: keep-alive Host: api-server:80 User-Agent: browser UA Content-Type: application/x-www-form-urlencoded Content-Length: 76 username=<username>&email=<username@domain.com>&password=<password>

Response	HTTP/1.0 200 OK
	Content-Type: application/json
	Content-Length: 397
	server: API Web Server
	Date: Sun, 25 Jul 2021 11:21:25 GMT
	Location: /api/v1/users/<user-resource-id>
	
	{
	  "_id": "user-resource-id",
	  "email": “username@domain.com",
	  "exp": 1627797085.108836,
	  "iat": 1627192285.108832,
	  "name": "username",
	  "sub": "user-jwt-token",
	  "token": "user-jwt-token,
	  "ver": 1
	}

Comments	In the PoC app we will be using the above api for creating dummy users in the system, the post request here would require okta validated JWT token to ensure only admin user can create the dummy users. In a real system, this api shall require user email/phone validation workflow and multi-factor authentication workflows to be integrated.
Data Flow	Front End / User App -> Users api server

	Retrieve a user
Request	GET /api/v1/users/
	or
	GET /api/v1/users/<user-resource-id>
	Accept: application/json, text/plain, */*
	Accept-Encoding: gzip, deflate, br
	Connection: keep-alive
	Host: api-server:80
	User-Agent: browser UA
	Content-Length: 0
Response	HTTP/1.0 200 OK
	Content-Type: application/json
	Content-Length: 397
	server: API Web Server
	Date: Sun, 25 Jul 2021 11:21:25 GMT
	Location: /api/v1/users/<user-resource-id>
	{
	"_id": "user-resource-id",
	"email": “username@domain.com",
	"exp": 1627797085.108836,
	"iat": 1627192285.108832,
	"name": "username",
	"sub": "user-jwt-token",
	"token": "user-jwt-token,
	"ver": 1
	}
Comments	The first api may be accessible to regular users to fetch their own profile information, the second format can be made available to admin users to obtain info on any of the platform’s users.
Data Flow	Front End / User App -> Users api server

	Update a user

Request	PATCH /api/v1/users/<user-resource-id>
	Accept: application/json, text/plain, */*
	Accept-Encoding: gzip, deflate, br
	Connection: keep-alive
	Host: api-server:80
	User-Agent: browser UA
	Content-Length: 123
	
	{
	  "name": “new-user-name",
	}

Response	HTTP/1.0 200 OK
	Content-Type: application/json
	Content-Length: 397
	server: API Web Server
	Date: Sun, 25 Jul 2021 11:21:25 GMT
	Location: /api/v1/users/<user-resource-id>
	
	{
	  "_id": "user-resource-id",
	  "email": “username@domain.com",
	  "exp": 1627797085.108836,
	  "iat": 1627192285.108832,
	  "name": “new-user-name“,
	  "sub": "user-jwt-token",
	  "token": "user-jwt-token,
	  "ver": 1
	}

Comments	Api may allow only specific fields (e.g. name, display-name, location etc) to be updated.
Data Flow	Front End / User App -> Users api server

	Search user(s)
Request	GET /api/v1/users/search?name=foo&location=bar Accept: application/json, text/plain, */* Accept-Encoding: gzip, deflate, br Connection: keep-alive Host: api-server:80 User-Agent: browser UA Content-Length: 0

Response	HTTP/1.0 200 OK
	Content-Type: application/json
	Content-Length: 397
	server: API Web Server
	Date: Sun, 25 Jul 2021 11:21:25 GMT
	
	[{
	  "_id": “user-resource-id",
	  "name": “user-name“,
	  "location": “http://api-server/api/v1/users/<user-resource-id>“,
	},
	{
	  "_id": "user2-resource-id",
	  "name": “user2-name“,
	  "location": “http://api-server/api/v1/users/<user2-resource-id>“,
	},
	…
	]

Comments	All the GET apis that are expected to return more than 1 resources must be paginated. Related - see Security Concerns and guidelines. Also note that this api should NOT return the entire user object(s), but only the basic / public information.
Data Flow	Front End / User App -> Users api server

Communities APIs

	Create a new community
Request	POST /api/v1/communities/new
	Accept: application/json, text/plain, */*
	Accept-Encoding: gzip, deflate, br
	Connection: keep-alive
	Host: api-server:80
	User-Agent: browser UA
	Content-Type: application/x-www-form-urlencoded
	Content-Length: 76
	name=<community-name>&description=<community-description>&tags=<csv list of tags>
Response	HTTP/1.0 200 OK
	Content-Type: application/json
	Content-Length: 397
	server: API Web Server
	Date: Sun, 25 Jul 2021 11:21:25 GMT
	Location: /api/v1/communities/<coommunity-resource-id>
	{
	"_id": "community-resource-id",
	“name": “community-name“,
	"description": “community-description“,
	“created_on": 1627797085,
	“created_by": “user-resource-id”,
	“tags”: “list of tags”,
	"users": “http://api-server/api/v1/communities/<community-resource-id>/users”,
	“posts": http://api-server/api/v1/communities/<community-resource-id>/posts",
	}
Comments	The api shall create a new community with a given name / description / tags, and return its metadata. It shall also return links to fetch users list and recent posts (both of which shall return empty set initially)
Data Flow	Front End / User App -> Communities api server

	Retrieve a community
Request	GET /api/v1/communities/<community-resource-id>
	Accept: application/json, text/plain, */*
	Accept-Encoding: gzip, deflate, br
	Connection: keep-alive
	Host: api-server:80
	User-Agent: browser UA
Response	HTTP/1.0 200 OK
	Content-Type: application/json
	Content-Length: 397
	server: API Web Server
	Date: Sun, 25 Jul 2021 11:21:25 GMT
	Location: /api/v1/communities/<community-resource-id>
	{
	"_id": "community-resource-id",
	“name": “community-name“,
	"description": “community-description“,
	“created_on": 1627797085,
	“created_by": “user-resource-id”,
	“tags”: “list of tags”,
	"users": “http://api-server/api/v1/communities/<community-resource-id>/users”,
	“posts": http://api-server/api/v1/communities/<community-resource-id>/posts",
	}
Comments	The api shall retrieve the community object given its resource id. Communities object shall also include links to fetch users and recent posts for the given community.
Data Flow	Front End / User App -> Communities api server

	Update a community info

Request	PUT/PATCH /api/v1/communities/<community-resource-id>
	Accept: application/json, text/plain, */*
	Accept-Encoding: gzip, deflate, br
	Connection: keep-alive
	Host: api-server:80
	User-Agent: browser UA
	Content-Length: 123
	
	{
	  "name": “new-community-name”,
	  "description": “new-community-description”,
	  "tags": “csv list of tags”,
	}

Response	HTTP/1.0 200 OK
	Content-Type: application/json
	Content-Length: 397
	server: API Web Server
	Date: Sun, 25 Jul 2021 11:21:25 GMT
	Location: /api/v1/communities/<community-resource-id>
	
	{
	  "_id": "community-resource-id",
	  “name": “new-community-name“,
	  "description": “new-community-description“,
	  “created_on": 1627797085,
	  “tags”: “list of tags”,
	  “created_by": “user-resource-id”,
	  "users": “http://api-server/api/v1/communities/<community-resource-id>/users”,
	  “posts": http://api-server/api/v1/communities/<community-resource-id>/posts",
	}

Comments	
Data Flow	Front End / User App -> Communities api server

	Invite a user to a community
Request	POST /api/v1/communities/<community-resource-id>/users/<user-resource-id>/invite Accept: application/json, text/plain, */* Accept-Encoding: gzip, deflate, br Connection: keep-alive Host: api-server:80 User-Agent: browser UA Content-Length: 0

Response	HTTP/1.0 200 OK
	Content-Type: application/json
	Content-Length: 397
	server: API Web Server
	Date: Sun, 25 Jul 2021 11:21:25 GMT
	Location: /api/v1/communities/<community-resource-id>/users/<user-resource-id>/invite
	
	{
	  "_id": “invite-resource-id",
	  "status": “default-invite-status”,
	  "created_on": 1627797085,
	}

Comments	The api create an invite for the given user id to a community id, and returns a location where the inviter can check the status of the invite. A successful call to this api shall emit an event to Kafka suggesting a new user invite has been generated, the consumer for the event shall send a notification to the target user. The originator of the invite may use the user search api to obtain list of users he/she may want to invite to a community.
Data Flow	Front End / User App -> Communities api server

	Accept an invite
Request	PUT /api/v1/communities/<community-resource-id>/invite/<invite-resource-id>/accept Accept: application/json, text/plain, */* Accept-Encoding: gzip, deflate, br Connection: keep-alive Host: api-server:80 User-Agent: browser UA Content-Length: 0

Response	HTTP/1.0 200 OK
	Content-Type: application/json
	Content-Length: 397
	server: API Web Server
	Date: Sun, 25 Jul 2021 11:21:25 GMT
	Location: /api/v1/communities/<community-resource-id>
	
	{
	  "_id": "community-resource-id",
	  “name": “community-name“,
	  "description": “community-description“,
	  “created_on": 1627797085,
	  “created_by": “user-resource-id”,
	  “tags”: “list of tags”,
	  "users": “http://api-server/api/v1/communities/<community-resource-id>/users”,
	  “posts": http://api-server/api/v1/communities/<community-resource-id>/posts",
	}

Comments	The api is originated by the recipient of an invite, accepting the invite shall add the user to the named community. Accepting an invite should take the user to the community page. The community object returned by the api should then show a list of recent posts. This api shall also trigger an event to Kafka, the consumers may want to delete / update the invite status.
Data Flow	Front End / User App -> Communities api server

	Decline an invite
Request	PUT /api/v1/communities/<community-resource-id>/invite/<invite-resource-id>/decline Accept: application/json, text/plain, */* Accept-Encoding: gzip, deflate, br Connection: keep-alive Host: api-server:80 User-Agent: browser UA Content-Length: 0
Response	HTTP/1.0 200 OK Content-Type: application/json Content-Length: 2 server: API Web Server Date: Sun, 25 Jul 2021 11:21:25 GMT {}
Comments	The api is originated by the recipient of an invite, declining the invite shall trigger an event to Kafka, the consumers may want to delete / update the invite status.
Data Flow	Front End / User App -> Communities api server

	Subscribe to a community
Request	PUT /api/v1/communities/<community-resource-id>/subscribe Accept: application/json, text/plain, */* Accept-Encoding: gzip, deflate, br Connection: keep-alive Host: api-server:80 User-Agent: browser UA Content-Length: 0

Response	HTTP/1.0 200 OK
	Content-Type: application/json
	Content-Length: 397
	server: API Web Server
	Date: Sun, 25 Jul 2021 11:21:25 GMT
	Location: /api/v1/communities/<community-resource-id>
	
	{
	  "_id": "community-resource-id",
	  “name": “community-name“,
	  "description": “community-description“,
	  “created_on": 1627797085,
	  “created_by": “user-resource-id”,
	  “tags”: “list of tags”,
	  "users": “http://api-server/api/v1/communities/<community-resource-id>/users”,
	  “posts": http://api-server/api/v1/communities/<community-resource-id>/posts",
	}

Comments	A user may want to join a community (without an invite). The api shall add the current user to the members of community and emit an event to Kafka.
Data Flow	Front End / User App -> Communities api server

	Unsubscribe a community
Request	PUT /api/v1/communities/<community-resource-id>/unsubscribe Accept: application/json, text/plain, */* Accept-Encoding: gzip, deflate, br Connection: keep-alive Host: api-server:80 User-Agent: browser UA Content-Length: 0
Response	HTTP/1.0 200 OK Content-Type: application/json Content-Length: 2 server: API Web Server Date: Sun, 25 Jul 2021 11:21:25 GMT {}
Comments	A user may want to leave a community. The api shall remove the current user from the list of community member and emit an event to Kafka.
Data Flow	Front End / User App -> Communities api server

	Search for communities
Request	GET /api/v1/communities/search?q=<search-str>&tags=<csv list of tags>
	Accept: application/json, text/plain, */*
	Accept-Encoding: gzip, deflate, br
	Connection: keep-alive
	Host: api-server:80
	User-Agent: browser UA
Response	HTTP/1.0 200 OK
	Content-Type: application/json
	Content-Length: 1234
	server: API Web Server
	Date: Sun, 25 Jul 2021 11:21:25 GMT
	[{
	"_id": "community-resource-id",
	“name": “community-name“,
	"description": “community-description“,
	“created_on": 1627797085,
	“created_by": “user-resource-id”,
	“tags”: “list of tags”,
	"users": “http://api-server/api/v1/communities/<community-resource-id>/users”,
	“posts": http://api-server/api/v1/communities/<community-resource-id>/posts",
	},
	{
	"_id": "community2-resource-id",
	“name": “community2-name“,
	"description": “community2-description“,
	“created_on": 1627798196,
	“created_by": “user2-resource-id”,
	“tags”: “list of tags”,
	"users": “http://api-server/api/v1/communities/<community2-resource-id>/users”,
	“posts": http://api-server/api/v1/communities/<community2-resource-id>/posts",
	}
	…
	]
Comments	The api shall return a list of communities having name / description matching the given string or communities having one or more of given tags. The answer records shall be sorted by most relevance first.
Data Flow	Front End / User App -> Communities api server

Posts Apis

	Publish a post in a community
Request	POST /api/v1/communities/<community-resource-id>/posts/new Accept: application/json, text/plain, */* Accept-Encoding: gzip, deflate, br Connection: keep-alive Host: api-server:80 User-Agent: browser UA Content-Length: 1234 Content-Type: application/x-www-form-urlencoded content=urlencoded-post-content-here—may-include-third-party-urls.

Response	HTTP/1.0 200 OK
	Content-Type: application/json
	Content-Length: 1234
	server: API Web Server
	Date: Sun, 25 Jul 2021 11:21:25 GMT
	Location: /api/v1/communities/<community-resource-id>/posts/<post-resource-id>
	
	{
	"_id": "post-resource-id",
	"content": “post-content“,
	“created_on": 1627797085,
	“created_by": “user-resource-id”,
	“likes”: “like count”,
	“shares”: “share count”,
	“comments”: “http://api-server/api/v1/communities/<community-resource-id>/posts/<post-resource-id>/comments”,
	"user": “http://api-server/api/v1/users/<user-resource-id>”,
	“community": "http://api-server/api/v1/communities/<community-resource-id>",
	“shared_with”: “”,
	}

Comments	The api shall return a post resource, with links to the parent community, creator/user resource and post comments. The api server may generate an event for Kafka to notify it about the new post.
Data Flow	Front End / User App -> Posts api server

	Fetch recent posts for a community
Request	GET /api/v1/communities/<community-resource-id>/posts/ Accept: application/json, text/plain, */* Accept-Encoding: gzip, deflate, br Connection: keep-alive Host: api-server:80 User-Agent: browser UA

Response	HTTP/1.0 200 OK
	Content-Type: application/json
	Content-Length: 1234
	server: API Web Server
	Date: Sun, 25 Jul 2021 11:21:25 GMT
	
	[{
	  "_id": "post-resource-id",
	 "content": “post-content“,
	  “created_on": 1627797085,
	  “created_by": “user-resource-id”,
	  “likes”: “like count”,
	  “shares”: “share count”,
	  “comments”: “http://api-server/api/v1/communities/<community-resource-id>/posts/<post-resource-id>/comments”,
	  "user": “http://api-server/api/v1/users/<user-resource-id>”,
	  “community": "http://api-server/api/v1/communities/<community-resource-id>",
	  “shared_with”: "http://api-server/api/v1/communities/<community-resource-id>/posts/<post-resource-id>/shares",
	,
	
	},
	{
	  "_id": "post2-resource-id",
	 "content": “post2-content“,
	  “created_on": 1627797085,
	  “created_by": “user2-resource-id”,
	  “likes”: “like count”,
	  “shares”: “share count”,
	  “comments”: “http://api-server/api/v1/communities/<community-resource-id>/posts/<post2-resource-id>/comments”,
	  "user": “http://api-server/api/v1/users/<user2-resource-id>”,
	  “community": http://api-server/api/v1/communities/<community-resource-id>",
	  “shared_with”: "http://api-server/api/v1/communities/<community-resource-id>/posts/<post2-resource-id>/shares",
	
	},
	…
	]

Comments	The api shall return a list of posts, sorted by most recent ones first.
Data Flow	Front End / User App -> Posts api server

	Update a post
Request	PUT /api/v1/communities/<community-resource-id>/posts/<post-resource-id> Accept: application/json, text/plain, */* Accept-Encoding: gzip, deflate, br Connection: keep-alive Host: api-server:80 User-Agent: browser UA Content-length: 1234 Content-Type: application/x-www-form-urlencoded content=urlencoded-updated-post-content-here—may-include-third-party-urls.

Response	HTTP/1.0 200 OK
	Content-Type: application/json
	Content-Length: 1234
	server: API Web Server
	Date: Sun, 25 Jul 2021 11:21:25 GMT
	Location: /api/v1/communities/<community-resource-id>/posts/<post-resource-id>
	
	{
	  "_id": "post-resource-id",
	 "content": “post-content“,
	  “created_on": 1627797085,
	  “created_by": “user-resource-id”,
	  “likes”: “like count”,
	  “shares”: “share count”,
	  “comments”: “http://api-server/api/v1/communities/<community-resource-id>/posts/<post-resource-id>/comments”,
	  "user": “http://api-server/api/v1/users/<user-resource-id>”,
	  “community": "http://api-server/api/v1/communities/<community-resource-id>",
	  “shared_with”: "http://api-server/api/v1/communities/<community-resource-id>/posts/<post-resource-id>/shares"
	}

Comments	The api shall return a the updated post resource.
Data Flow	Front End / User App -> Posts api server

TODO:

Add apis for comments / likes / shares.

User-Groups apis

The user-groups will be maintained as topic partitions in Kafka. They should also be persisted to the database for long term storage and provide chat history to the new users joining a user-group.

However, if long term storage of chat/message history is not needed, we may use Kafka alone for storing the user-groups, its member's list and the chat/messages sent by the the user-group members.

	Create a user-group
Request	POST /api/v1/communities/<community-resource-id>/usergroups/new Accept: application/json, text/plain, */* Accept-Encoding: gzip, deflate, br Connection: keep-alive Host: api-server:80 User-Agent: browser UA Content-length: 1234 Content-Type: application/x-www-form-urlencoded name=<usergroup name>&description=<usergroup description>

Response	HTTP/1.0 200 OK
	Content-Type: application/json
	Content-Length: 1234
	server: API Web Server
	Date: Sun, 25 Jul 2021 11:21:25 GMT
	Location: /api/v1/communities/<community-resource-id>/usergroups/<usergroup-resource-id>
	
	{
	"_id": "usergroup-resource-id",
	"name": "usergroup name",
	"created_on": 1627797085,
	"created_by": "user-resource-id",
	"community": "http://api-server/api/v1/communities/<community-resource-id>",
	"users": "http://api-server/api/v1/communities/<community-resource-id>/usergroups/<usergroup-resource-id>/users"
	}

Comments	The api shall create a new user-group within a community, and return a user-group resource. It should create a new Kafka topic partition using the usergroup-resource-id, and subscribe the originator user-resource-id to the Kafka topic partition for the user-group. In effect, a user-group will work similar to a chatroom, where more users can later join in. The user-group object should also be persisted at the database for obtaining the chat/message history.
Data Flow	Front End / User App -> UserGroups api server

	Fetch usergroups within a community
Request	GET /api/v1/communities/<community-resource-id>/usergroups/ Accept: application/json, text/plain, */* Accept-Encoding: gzip, deflate, br Connection: keep-alive Host: api-server:80 User-Agent: browser UA Content-length: 0

Response	HTTP/1.0 200 OK
	Content-Type: application/json
	Content-Length: 1234
	server: API Web Server
	Date: Sun, 25 Jul 2021 11:21:25 GMT
	
	[{
	  "_id": "usergroup-resource-id",
	 "name": “usergroup name“,
	  “created_on": 1627797085,
	  “created_by": “user-resource-id”,
	 “community": "http://api-server/api/v1/communities/<community-resource-id>",
	  “users”: "http://api-server/api/v1/communities/<community-resource-id>/usergroups/<usergroup-resource-id>/users"
	},
	{
	  "_id": "usergroup2-resource-id",
	 "name": “usergroup2 name“,
	  “created_on": 1627797085,
	  “created_by": “user-resource-id”,
	 “community": "http://api-server/api/v1/communities/<community-resource-id>",
	  “users”: "http://api-server/api/v1/communities/<community-resource-id>/usergroups/<usergroup2-resource-id>/users"
	}
	…
	]

Comments	The api shall return list of user-groups within a community, the list could be sorted with the most recent active user-groups listed first.
Data Flow	Front End / User App -> UserGroups api server

	Get list of users within a user group
Request	GET /api/v1/communities/<community-resource-id>/usergroups/<user-group-id>/users Accept: application/json, text/plain, */* Accept-Encoding: gzip, deflate, br Connection: keep-alive Host: api-server:80 User-Agent: browser UA Content-length: 0

Response	HTTP/1.0 200 OK
	Content-Type: application/json
	Content-Length: 1234
	server: API Web Server
	Date: Sun, 25 Jul 2021 11:21:25 GMT
	
	[{
	  "_id": “user-resource-id",
	  "name": “user-name“,
	  "location": “http://api-server/api/v1/users/<user-resource-id>“,
	},
	{
	  "_id": "user2-resource-id",
	  "name": “user2-name“,
	  "location": “http://api-server/api/v1/users/<user2-resource-id>“,
	},
	…
	]

Comments	The api shall return the list of users that belong to a given user-group in a community.
Data Flow	Front End / User App -> UserGroups api server

	Invite a user to a user-group
Request	PUT /api/v1/communities/<community-resource-id>/usergroups/<user-group-id>/users/<user-resource-id>/invite Accept: application/json, text/plain, */* Accept-Encoding: gzip, deflate, br Connection: keep-alive Host: api-server:80 User-Agent: browser UA Content-length: 0

Response	HTTP/1.0 200 OK
	Content-Type: application/json
	Content-Length: 1234
	server: API Web Server
	Date: Sun, 25 Jul 2021 11:21:25 GMT
	Location: /api/v1/communities/<community-resource-id>/usergroups/<usergroup-resource-id>/users/<user-resource-id>/invite
	
	{
	  "_id": “invite-resource-id",
	  "status": “default-invite-status”,
	  "created_on": 1627797085,
	}

Comments	The api shall return an invite resource, where the originator can see the status of the invite.
Data Flow	Front End / User App -> UserGroups api server

	Accept an invite to join a user-group
Request	PUT /api/v1/communities/<community-resource-id>/usergroups/<user-group-id>/invite/ <invite-resource-id>/accept Accept: application/json, text/plain, */* Accept-Encoding: gzip, deflate, br Connection: keep-alive Host: api-server:80 User-Agent: browser UA Content-length: 0

Response	HTTP/1.0 200 OK
	Content-Type: application/json
	Content-Length: 1234
	server: API Web Server
	Date: Sun, 25 Jul 2021 11:21:25 GMT
	Location: /api/v1/communities/<community-resource-id>/usergroups/<usergroup-
	resource-id>
	
	{
	  "_id": "usergroup-resource-id",
	 "name": “usergroup name“,
	  “created_on": 1627797085,
	  “created_by": “user-resource-id”,
	 “community": "http://api-server/api/v1/communities/<community-resource-id>",
	  “users”: "http://api-server/api/v1/communities/<community-resource-id>/usergroups/
	<usergroup-resource-id>/users"
	}

Comments	The api shall add the user to the members of the user group by subscribing it to the and return the user group object. It should also update the invite status, and emit an event for Kafka to notify it of a user joining a user-group.
Data Flow	Front End / User App -> UserGroups api server

	Decline an invite to join a user-group
Request	PUT /api/v1/communities/<community-resource-id>/usergroups/<user-group-id>/invite/<invite-resource-id>/decline Accept: application/json, text/plain, */* Accept-Encoding: gzip, deflate, br Connection: keep-alive Host: api-server:80 User-Agent: browser UA Content-length: 0
Response	HTTP/1.0 200 OK Content-Type: application/json Content-Length: 1234 server: API Web Server Date: Sun, 25 Jul 2021 11:21:25 GMT Location: /api/v1/communities/<community-resource-id>/usergroups/<usergroup-resource-id> { "_id": "usergroup-resource-id", "name": "usergroup name", "created_on": 1627797085, "created_by": "user-resource-id", "community": "http://api-server/api/v1/communities/<community-resource-id>", "users": "http://api-server/api/v1/communities/<community-resource-id>/usergroups/<usergroup-resource-id>/users" }
Comments	The api shall add the user to the members of the user group and return the user group object. It will also clear the invite status.
Data Flow	Front End / User App -> UserGroups api server

Messages apis

The messages api shall be implemented via web-sockets, client application / browser shall initiate a web socket connection to messages api server.

The web server shall expose the following web-socket endpoints -

/messages/send
Receives messages sent by the client on this channel, Push the message to Kafka, where the consumer may do any pre-processing (e.g. extracting url to generate content preview) before persisting it to the database.

/images/send
Receives images sent by the clients and persists it on a distributed storage.

videos/send
Receives videos sent by the client / web browser, and persists it on a distributed storage.

Both the /images/send and /videos/send apis should limit the maximum amount of file size that can be sent on the channel.

Data Pipeline

Kafka - Topics, Producers and Consumers

At the heart of the system, A Kafka cluster shall handle most of the incoming events and provide various topics, which the consumer can subscribe to.

Kafka cluster shall provide the following topics -

- **messages**
- **likes**
- **shares**
- **comments**
- **posts**

Additional topics may be created based on the requirements.

Each of the above topics shall hold events / data corresponding the named type. The topics can be divided into topic-partitions to allow for parallel processing. Additional Kafka nodes may be added if the need arises.

Producers

The api server will act as the primary producers for the Kafka cluster. There may be other producers running independently in the system and generating events for various purposes,
For example -

- Content recommendation for the users
- Targeted ads

Consumers
For implementing the lambda architecture, we shall be using two consumers primarily,

1)
The speed layer may be implemented using **Kafka Streams**, **Spark Streaming, Apache Flink** or **Apache Storm**, all of which have support stream inputs from a Kafka cluster.

For PoC app, we may go ahead either with Spark Streaming or Apache Flink, as both of them have python based support (using pyspark and pyflink library.)

2)

For the PoC, we would go ahead with Mongo db along with the Mongo db sink connector. The mongo sink connector allows for at-least-once and exactly-once message delivery guarantees and readily be used to deliver and persist Kafka topic messages to a mongo database.

Other alternatives like Cassandra / Hadoop hbase may also be explored.

TODO:

Serving Layer with ElasticSearch

Delivery via Python / Flask + Vue.js

Design choices / improvements

Several design choices are currently influenced by our lack of experience / exposure to some of the streaming / big data technologies, as well as the time constraints for implementing the PoC application.

A thorough review and exploration of the other alternatives may lead to a better suited design for the community discussion application.

Apart from this, many other components may be better implemented with commercial applications, that are not accessible to us due to licensing and monetary requirements, and thus leaving us with the open-source solutions only.

Security concerns / guidelines

- Under no circumstances shall the data be deleted from the system and maximum efforts shall be made to preserve the history of every resource creation, update and access, except where the data policies (e.g. GDPR) or law-enforcement require the data to be deleted upon expiry or upon user request.
- All the GET apis that are expected to return more than 1 resources (e.g. search users, search communities etc) must have pagination support. The apis may support a configurable default value (say 10 records per query), while a non-configurable / hardcoded max value must restrict the number of items per search queries to an allowed maximum (say max 100 records per query).
- The backend api servers should accept connection only from the frontend servers, and must mandate csrf headers.
- All the http apis must support and validate the user JWT tokens, and do proper authorisation checks before allowing access to any resources, except for the apis which may be open for access without authentication (for example, search apis)

Future Scope

Several other feature may be required to make the application suitable for a global platform, including but not limited to -

Content suggestion

Content rating / popularity scores

Integration with other sites, link embedding

Support for user content in various languages.

Translation of non-native language content to a user’s native language.

Support for newer media formats like interactive 3-d videos and images.

Copyright infringement detection, copyright claims workflows and strike down policies.

User reputation system

User-driven reporting and flagging of malicious and offensive content, also automated detection of such content.

Throttling, DDoS prevention and prevention of other attack vectors like XSS

Search Engine optimisation.

Monetisation workflows.

Developer apis

Usability and UI / UX improvements

Support for various platforms, mobile applications

Reducing network latency, CDNs, geo load balancing, DNS servers.

2 factor / multi factor authentication

Logging, Monitoring

Deployment Strategies, Continuous Integration, Continuous Deployment.

…

Links / References

Implementation of python / web sockets chat
https://github.com/heroku-examples/python-websockets-chat

How Pinterest runs at scale
https://www.confluent.io/blog/running-kafka-at-scale-at-pinterest/

Streaming Kafka messages to internet facing clients over web-sockets
https://ably.com/topic/websockets-kafka

Integrating ElasticSearch with mongoldb made easy
https://hevodata.com/learn/integrating-elasticsearch-and-mongodb/

Mongo db with GridFS for large file storage
https://docs.mongodb.com/manual/core/gridfs/

Lambda architecture with no-sql
https://blog.couchbase.com/lamda-architecture-and-beyond-with-nosql/

Vue.js + JWT authentication
https://jasonwatmore.com/post/2018/07/06/vue-vuex-jwt-authentication-tutorial-example

Spark / Spark streaming for data pipeline
https://medium.com/walmartglobaltech/how-we-built-a-data-pipeline-with-lambda-architecture-using-spark-spark-streaming-9d3b4b4555d3

Multi-room chat application using Tornado and Redis
https://github.com/nellessen/Tornado-Redis-Chat

Mistakes when using apache kafka
https://blog.softwaremill.com/7-mistakes-when-using-apache-kafka-44358cd9cd6

Spark streaming programming guide
https://spark.apache.org/docs/latest/streaming-programming-guide.html

Mongo db sink connector
https://docs.mongodb.com/kafka-connector/master/kafka-sink/