Juntos System Design

Team	Beta
PRD	📄 Juntos - PRD
Wireframe Figma Mocks	Figma Mockup design
Author	Hassan Raza
Approver	Chirag Jain
Status	Approved
System Design GDoc	📄 Juntos System Design Doc
System Design Figma link	Figma Design Diagram

Services

Authentication Service

For authentication we will use passport js. Passport js is capable of implementing various authentication strategies including simple username/password auth and third party auth like Facebook and Google. Also, passport js implementation is secure and reduces the overhead of handling security concerns regarding authentication. Token (JWT) generated by passport js will be shared with the client, which will be used to secure our REST APIs. Security concerns regarding client side storage of token is described under the Security Concerns section.

Room Service

Room service enables users to join the chat room for conversations during media playback time. A user will be able to create a room and get a sharable room join link. Once all users are shared with the link, they can join the room and start their viewing together. A key component of this service is to generate a room link and allow authenticated users to be redirected to the appropriate application screen on visiting the join link. This service will also let the backend create a socket room via socket.io, where authenticated users will continue to communicate.

Chat Service

Once a socket room is created via Room Service, the users can share messages amongst themselves on a group chat. No REST API use is required for this service, all message communication will be handled through sockets. Each created room will have knowledge of all successfully joined users. This service also updates the required data in the database internally.

Playback Sync Service

This service is responsible for communicating sync data across connected users. For synchronising simple video controls like play and pause, the service will utilize a REST API. For continuous maintenance of a synchronized playback, the service will use sockets to send the current playback position to the server where a logic will be executed to check if all connected users players (Youtube iFrame) are running in sync. If any user(s) are not in sync, corresponding corrections will be sent to all the users. Corresponding correction sharing will depend on the control type priority set in the users group. It may be the case that all users are allowed to modify the playback or only a single user is allowed to do so.

Sockets Manager Service

This service will be implemented using socket.io. All socket implementation will be written under this service. All other services relying on sockets to communicate will use this common provided implementation.

Content Delivery Service

For now we will be using Youtube and its iFrame API for showing media content. If time allows other major video content service providers like vimeo, metacafe, dailymotion may be incorporated.

Audio Video Chat Service

Audio and video communication for group chat will be done via utilising WebRTC API. Under current implementation scope, priority of implementation will be audio chat and then video chat if time allows.

Capacity Estimation and Constraints

Our system will be write-heavy. There will also be considerable redirection requests while accessing join links. Let's assume a 100:1 ratio between write and read.

Traffic estimates:

Assuming, we will have 1M new group parties per month, with 8 maximum no of users in each group and with 100:1 write/read ratio, we can expect 9.6B requests during the same period:

1M * 8 * 2(login/signup, generate/join link) * 15(sync video) * 40(chat messages) => 9.6B

What would be Queries Per Second (QPS) for our system? Considering db modifying network calls

640 million / (30 days * 24 hours * 3600 seconds) = ~247 Network Calls/s

Considering 100:1 write/read ratio, Total Network Calls per second will be:

100 * 247 URLs/s = ~25K/s

Storage estimates:

Let's assume we store every request data for 5 years. Since we expect to have 1M new group parties every month, the total number of objects we expect to store will be 60 million:

1 million * 5 years * 12 months = 60 million

Let's assume that each stored object will be approximately 10 bytes (just an estimate – actual estimation to be done later). We will need 6TB of total storage:

60 million * 10 bytes = 6 TB

Bandwidth estimates:

For write requests, since we expect ~25K new requests every second, total outgoing data for our service will be 0.25MB per second:
25K * 10 bytes = 0.25 MB/s

For read requests, since every second we expect ~247 new requests every second, total incoming data for our service would be 250KB per second:

247 * 10 bytes = ~250 KB/s

High-level estimates:

Assuming 1 million new groups per month and 100:1 write:read ratio, following is the summary of the high level estimates for our service:

New Groups/Rooms	250/s
Network Requests	25K/s
Incoming data	250KB/s
Outgoing data	0.25MB/s
Storage for 5 years	6TB

System APIs

We will use REST APIs and sockets to expose the functionality of our services. Following could be the definitions of the APIs for managing resources under each type of service:

Auth Service APIs

POST		/signin
Request	Content-type	application/json
	Authorization	Bearer [Access_Token]
	Parameters	**Parameter Description**
	username: string	Play, pause controls
	password: String	
Response	Content-type	application/json
	Parameters	**Parameter Description**
200	status: String	"ok" for 200

	token : string	Jwt token
default	status: String	"error" for not 200 response

POST		**/signup**
Request	Content-type	application/json
	Authorization	Bearer [Access_Token]
	Parameters	**Parameter Description**
	fullname: string	
	username: String	
	email: String	
	password: String	
Response	Content-type	application/json
	Parameters	**Parameter Description**
200	status: String	"ok" for 200
	token : String	Jwt token

PUT		**/profile**
Request	Content-type	application/json
	Authorization	Bearer [Access_Token]
	Parameters	**Parameter Description**
	firstName: String	
	lastName: String	
	profileImage: string	Url for avatar
Response	Content-type	application/json
	Parameters	**Parameter Description**
200	status: String	"ok" for 200
default	status: String	"error" for not 200 response

GET		/profile
Request	Content-type	application/json
	Authorization	Bearer [Access_Token]
	Parameters	**Parameter Description**
Response	Content-type	application/json
	Parameters	**Parameter Description**
200	status: string	"ok"
	lastName: String	
	userName: String	
	email: String	
	profileImage: String	
	fname: string	

Room Service APIs

POST		/room
Request	Content-type	application/json
	Authorization	Bearer [Access_Token]
	Parameters	**Parameter Description**
	message: string	message that other user can see on join pop up component at frontend
Response	Content-type	application/json
	Parameters	**Parameter Description**
200	status: String	"ok"
	roomId : string	Socket link that other user can join
	videoId: string	Video id that is will be played for that current room
default	status: String	"error" for not 200 response

GET		/room
Request	Content-type	application/json
	Authorization	Bearer [Access_Token]
	Parameters	**Parameter Description**
	id: string	room id
Response	Content-type	application/json
	Parameters	**Parameter Description**
200	status: String	"ok"
	message: string	Message that was added while creating room
	videoId	String for YT id
default	status: String	"error" for not 200 response
Note	The user will join the shared link. The link will redirect the user to the appropriate screen in the application, asking the user to join the room. The user may see a join message sent by the creator of the room.	

Video Sync Service APIs

POST		/sync/playback
Request	Content-type	application/json
	Authorization	Bearer [Access_Token]
	Parameters	**Parameter Description**
	controlType: Number	Play, pause controls
	id: String	
Response	Content-type	application/json
	Parameters	**Parameter Description**
200	status: String	"ok" for 200
default	status: String	"error" for not 200 response

	errorId:	some error Id for debugging

- Media Playback current running position will be shared across all connected users through sockets for maintaining the synchronization of the media playback across users.

Chat Service APIs

Chat service will require the use of sockets to send and receive messages easily. As socket.io is implemented both at the server and client-side, a specific REST API for chat communication is not required. All communications necessary for a group chat can be achieved via sockets.

Communication through Sockets

- Send messages to the group channel through socket.
- Receive messages from the group channel through socket.

Database Design

A few observations about the nature of the data we will store:
1. We need to store records of users.
2. We need to store records of groups/rooms created.
3. We need to store the details of the videos being played in the group.
4. We need to store the conversation amongst the group members.
5. We need to store the activity log of the application.
6. Each object we store is small (less than 1K).
7. Our service is write-heavy.

Schemas

Relationships
- Room - User :   One-to-Many
- Room - Playback :   One-to-Many
- Chat - Room :   One-to-One
- User - Activity :   One-to-Many

USER	PLAYBACK	ROOM	CHAT
id	id	id	id
email	url	users	room_id
name	creation_date	share_link	from

token	status	video_id	message
profile_url		creation_date	datetime
creation_date			

ACTIVITY
uid
activity_type
activity_desc
datetime

System Design

[System Design Link - Figma](#)
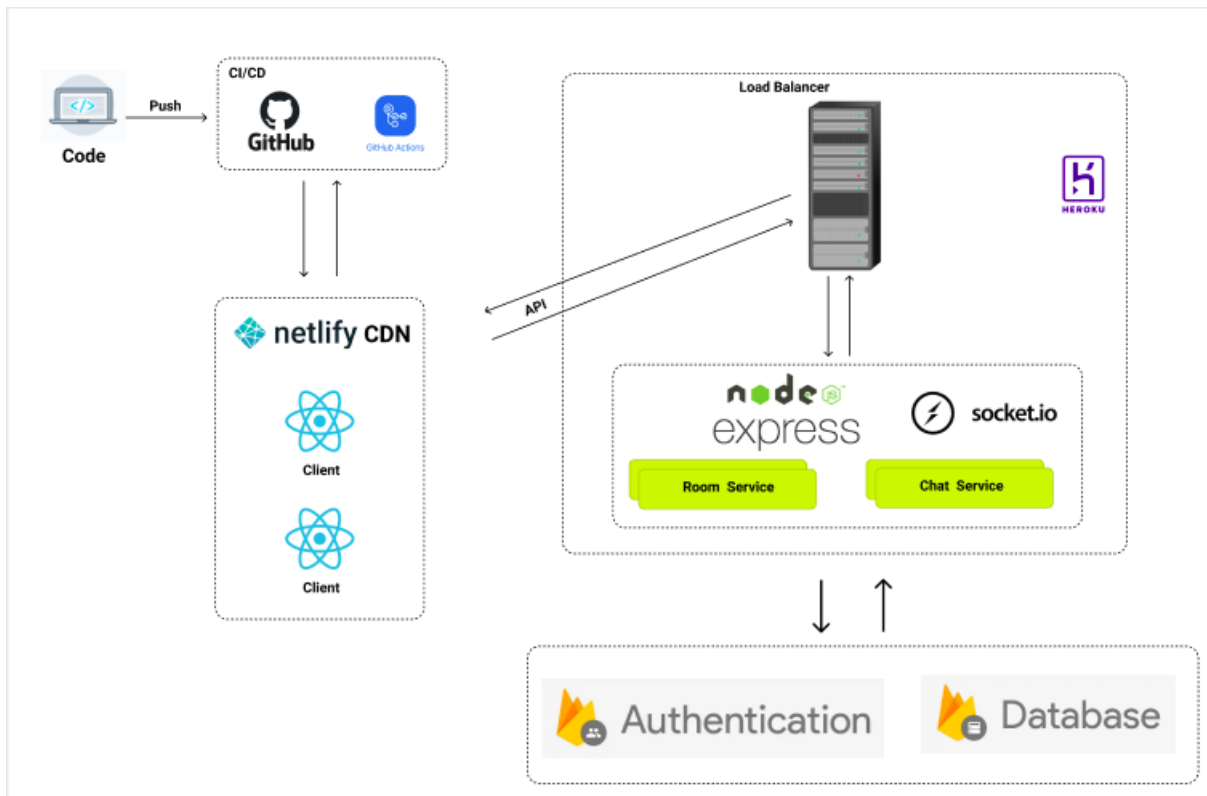
Scalability

Horizontally Scaling the Nodejs Application

In horizontal scaling we can duplicate the application instances across multiple cores within a machine. This way concurrent management of requests of Nodejs can be multiplied and parallelized. Horizontal scaling can be achieved by Nodejs native cluster module or through a process manager module like PM2. We will be using the PM2 cluster module feature in our application, utilising its ability to spawn N processes of the application as workers with round-robin balancing.

Vertically Scaling the Nodejs Application

Vertical scaling is to increase the individual machine capacity for scaling purposes. RAM and no CPU cores can be increased to apply this type of scaling. We will implement this using AWS's feature of Auto Scaling.

Decoupling application instance from DB

Application will be decoupled from DB for efficient scaling. Deploying databases separately on independent machines enables us to scale applications as required. This enables application and database to scale independently.

Stateless authentication with JWT

Stateless authentication will be achieved by JSON Web Tokens. This enables the authentication service to seamlessly process requests authentications. User data is embedded in the token and a simple comparison of server generated token and request token via the authentication service validates the incoming requests. Multiple requests can be authenticated via multiple processes independently which is quite efficient and scalable.

Security Concerns

Preventing XSS

1. Escaping

   Escaping user input. Escaping data means taking the data an application has received and ensuring it's secure before rendering it for the end user. By escaping user input, key characters in the data received by a web page will be prevented from being interpreted in any malicious way. In essence, you're censoring the data your

web page receives in a way that will disallow the characters – especially < and > characters – from being rendered, which otherwise could cause harm to the application and/or users.

2. Sanitizing

Sanitize user input. Sanitizing data is a strong defense. Sanitizing user input is especially helpful on sites that allow HTML markup, to ensure data received can do no harm to users as well as your database by scrubbing the data clean of potentially harmful markup, changing unacceptable user input to an acceptable format.

3. Validation

Validating input is the process of ensuring an application is rendering the correct data and preventing malicious data from doing harm to the site, database, and users.

Known vulnerabilities in dependencies

Some versions of third-party components might contain security issues. Check your dependencies and update when better versions become available. There are tools which can help validate this security concern through npm audit, snyk.

Technical Stack

Hosting

- AWS

Storage & Database

- AWS S3
- MongoDB

Frontend

- Bundler Tool Webpack
- ReactJS with Typescript

Backend

- NodeJS
- Express
- Socket.io
- Mongoose
- PM2

Testing

Frontend

- JEST testing framework,
- React-testing-Library (Unit and Integration testing).

Backend

- Mocha