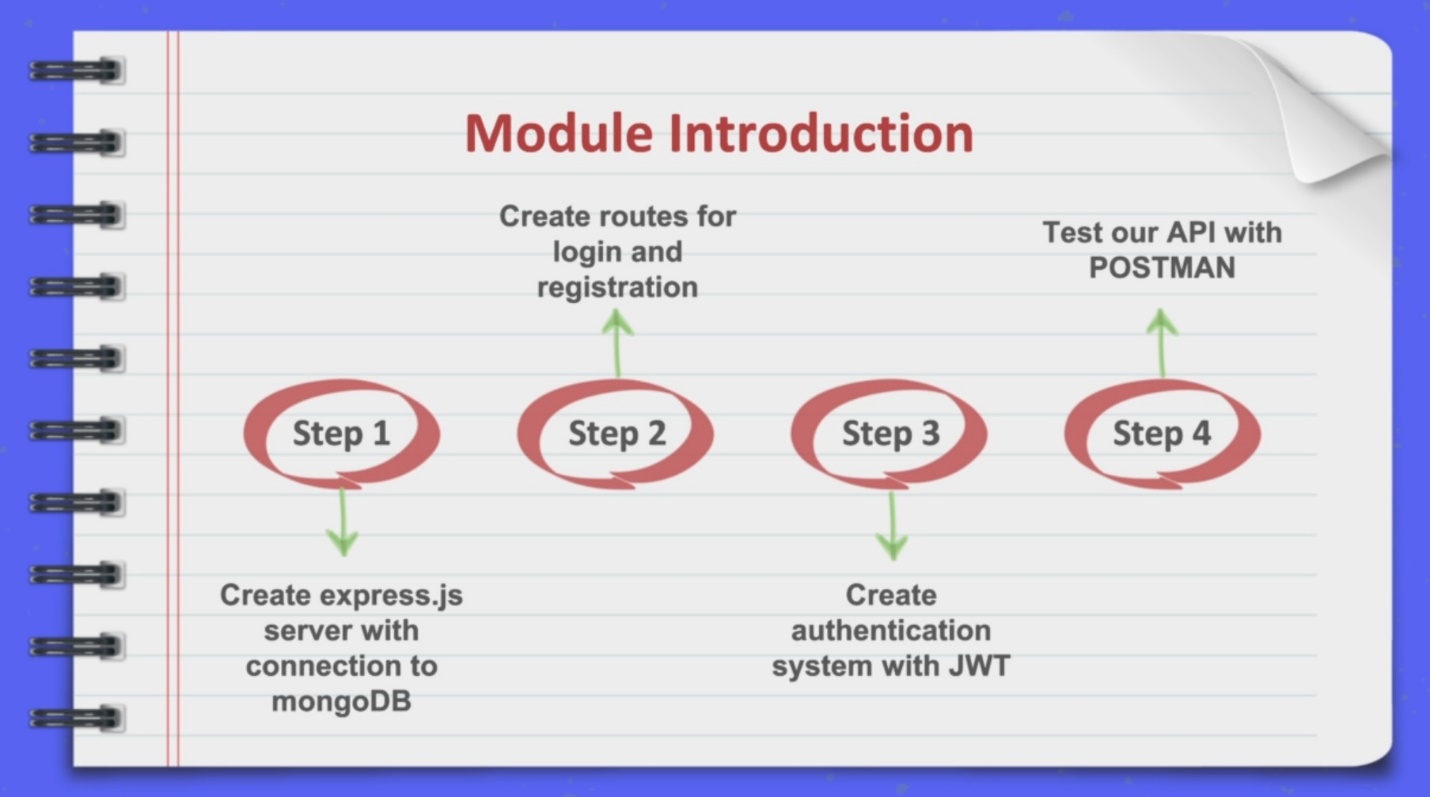
**MERN stack Video chat app**



**Step1:** We will be creating an **Express server** and we will also connect that to MongoDB and we’ll be storing all of the information about the users and all of the messages which will be created and on everything in our application in database and will be going for the **MongoDB in cloud**.

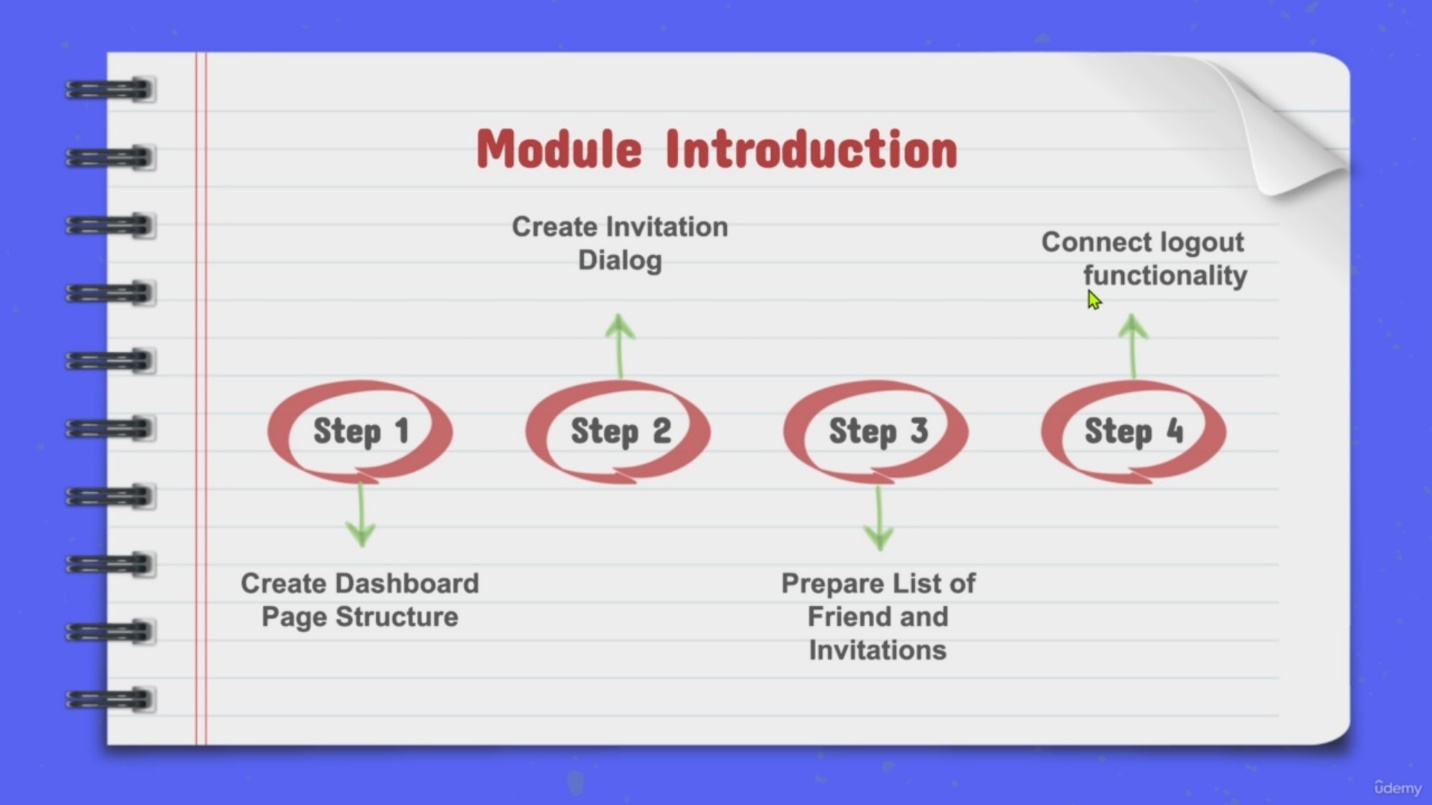
**Step2:** We’ll take care about creating **routes** for login and registration because first thing which we will be creating when we start creating our React app will be **a form**, which will allow us to **login or register**.

**Step3:** We’ll create the **authentication system with JWT token** that if the user will login or register successfully, the token will be returned to client site app and client will be able to access the **protected resources** at the server side.

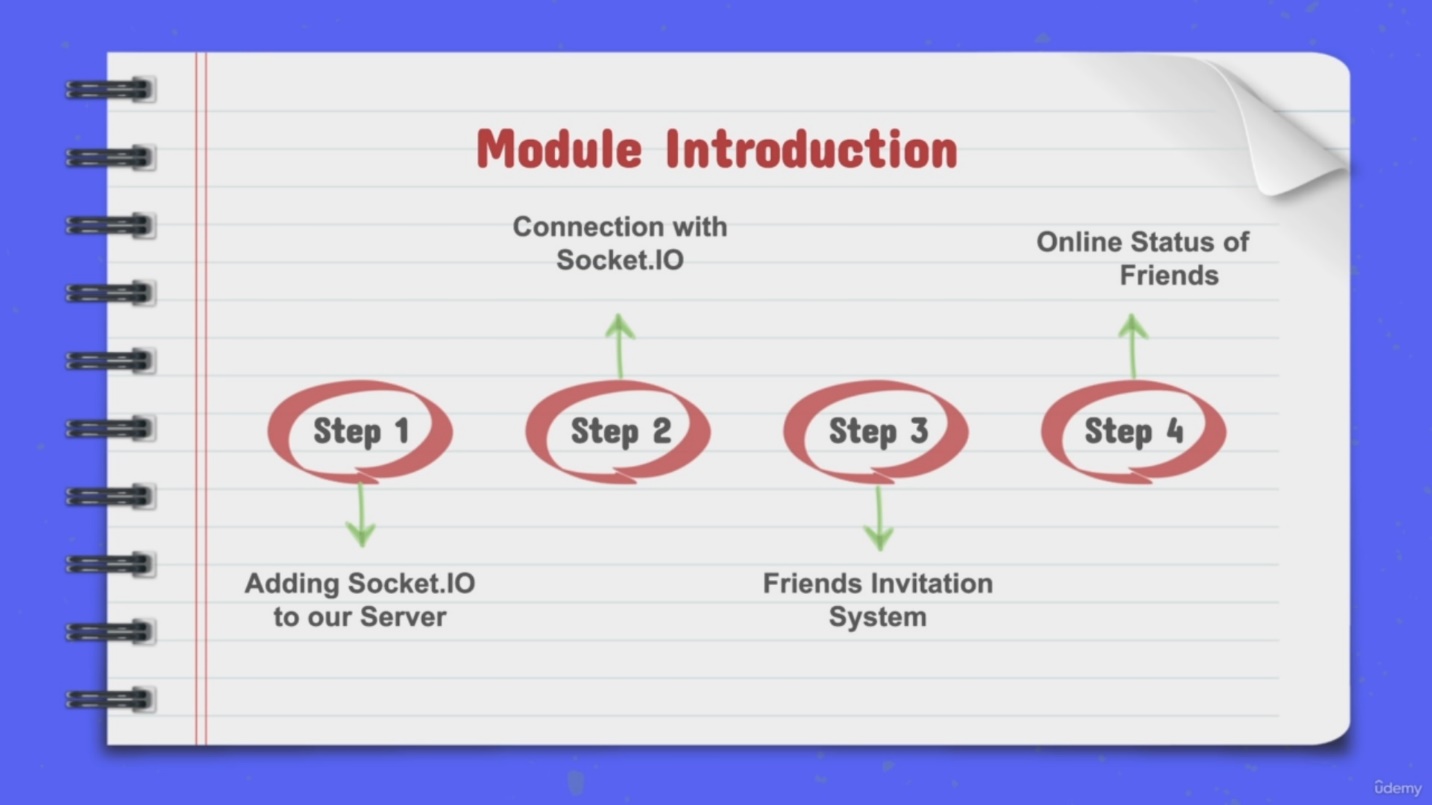
**Step4:** We’ll be **testing** our API with Postman.

/\*\*\*\*/

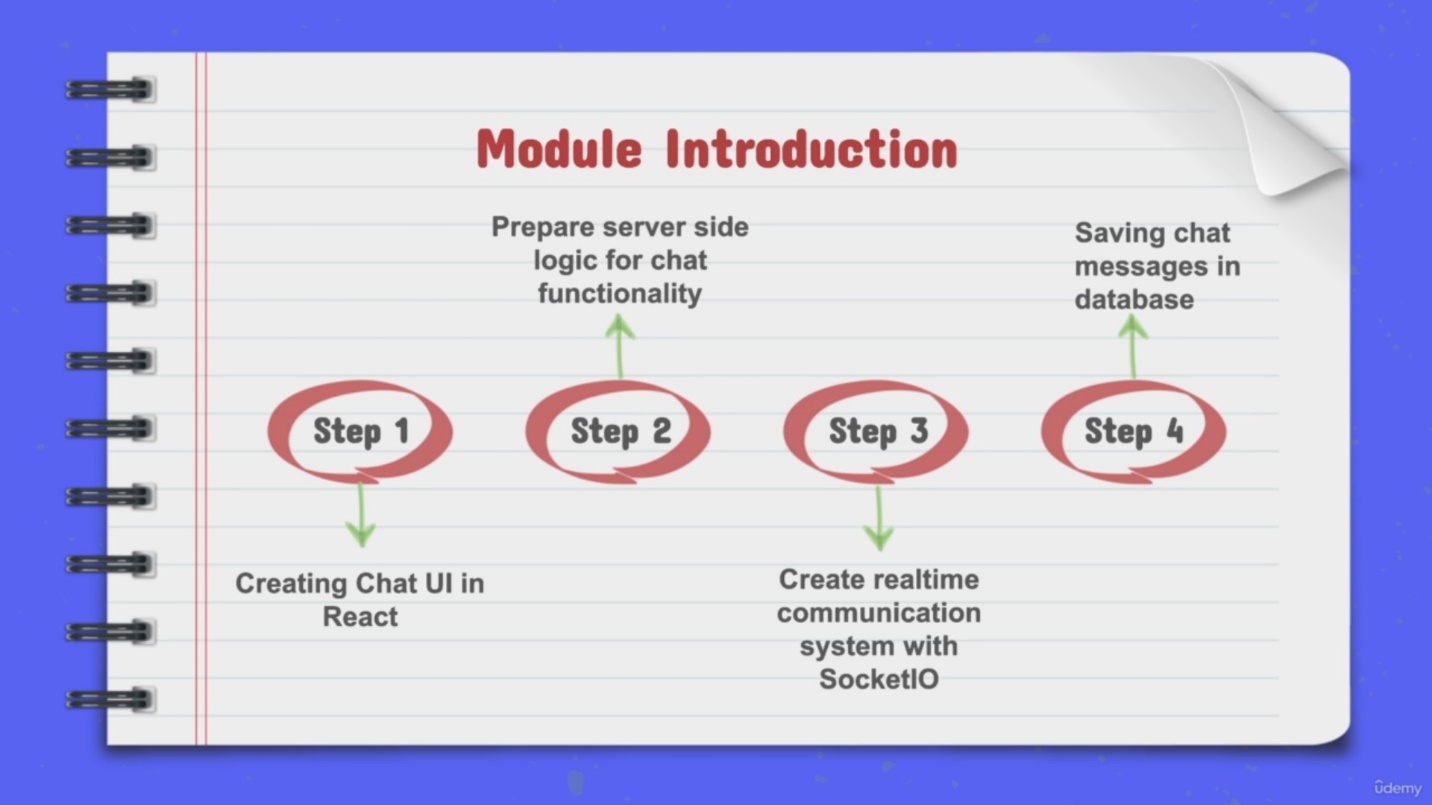
Module 4: Dashboard



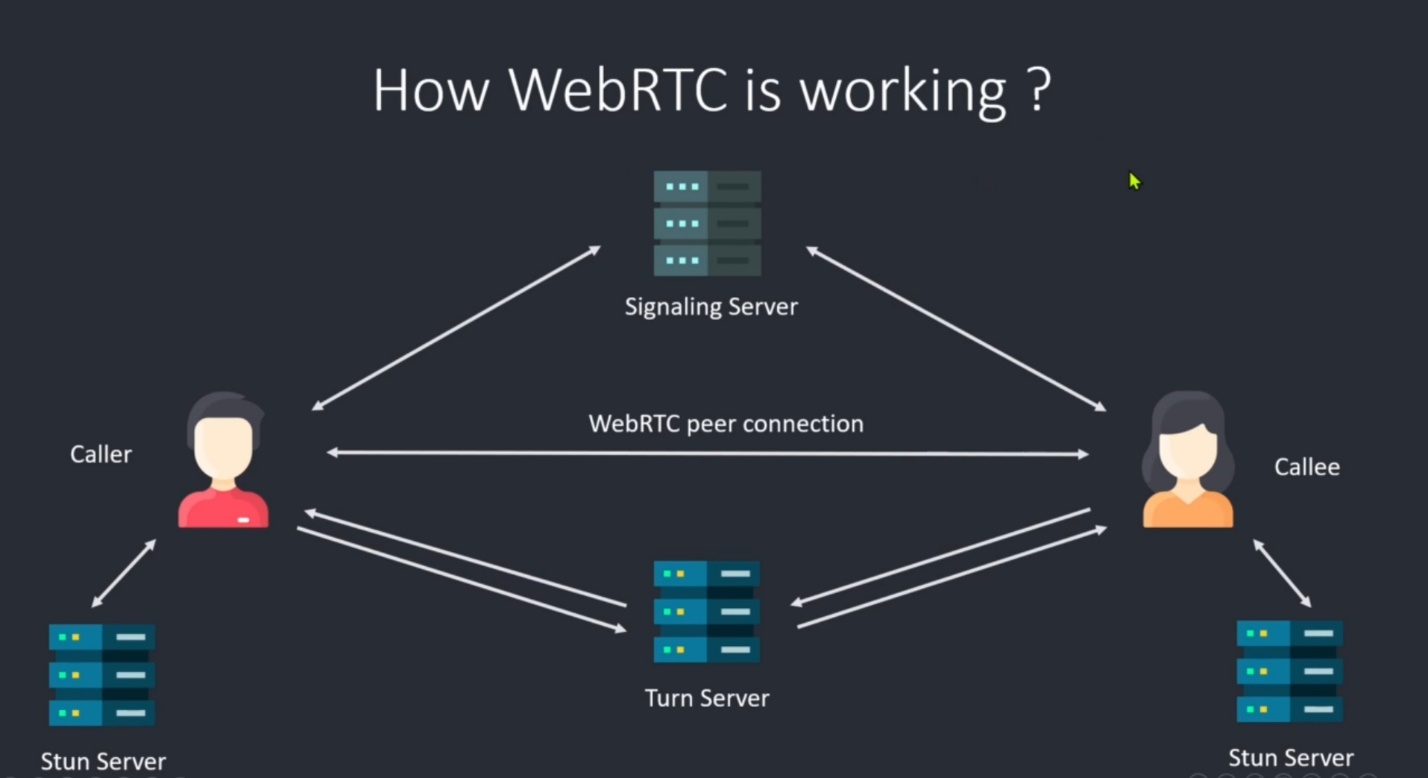
Module 5: Socket.io connection



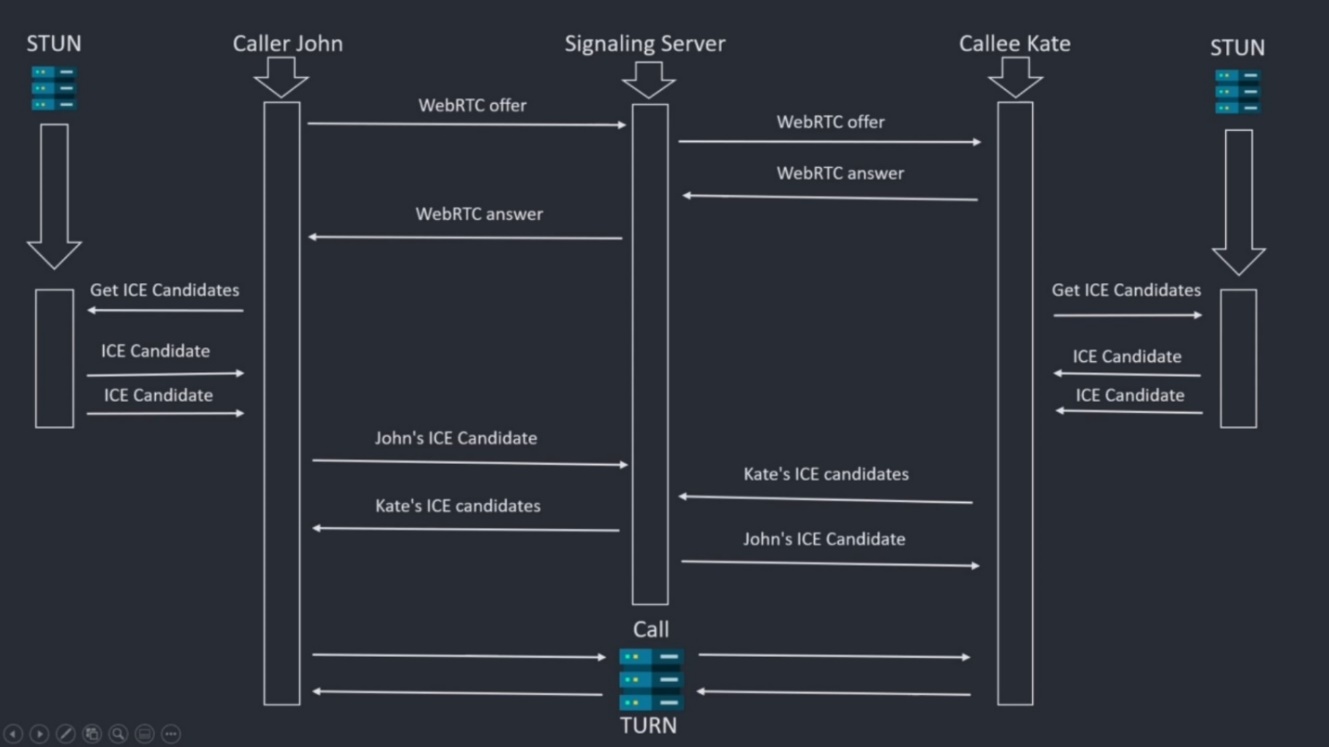
Module 6:



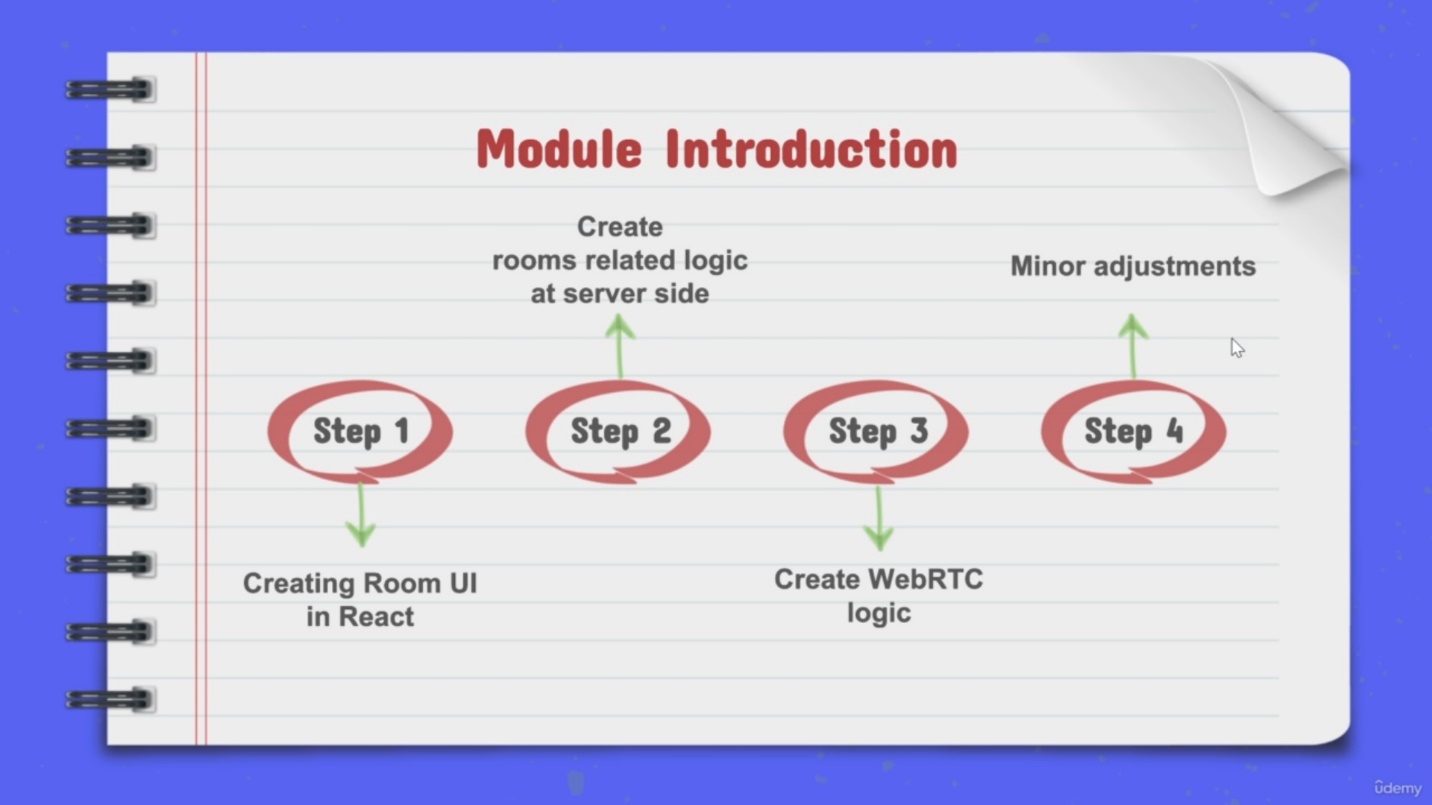
Module 7: WebRTC:

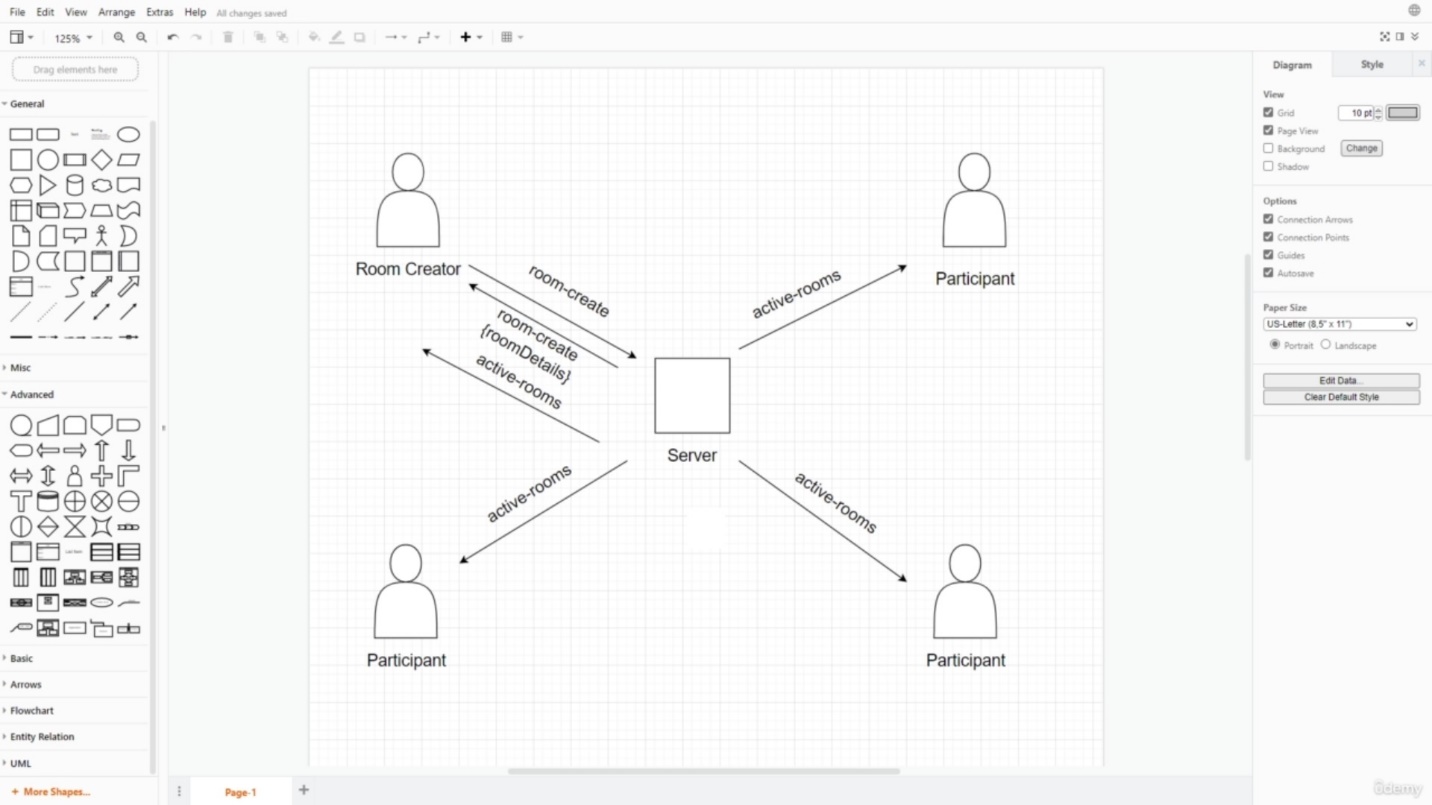


* WebRTC (Web Real Time Communication) is an open-source project that enables peer-to-peer communication between browsers or the applications (mobile). In other words, WebRTC allows us to exchange any kind of media through the web (such as video, audio and data) without any required plugin or framework.
* Applications which are using WebRTC:
  + Google Meet and Google Hangouts
  + Facebook Messanger
  + Discord
* To establish direct connection between 2 users we need a signaling server.
* Signaling server does not do anything that is WebRTC-specific. Signaling server helps to exchange necessary information which are required to establish direct connection between users. For signaling we can use whatever we like, from webSocket to XMLHttpRequest.
* Before the webRTC peer connection is established we need to exchange some data, this data would be related to our internet connection details and the codecs we are using, information about our browser, etc. in our project we have used webSockets for this purpose.
* STUN (Session Traversal Utilities for NAT) that allows clients to discover their public IP address and the type of NAT they are behind. This information is used to establish the media connection.
* In 15-20% cases STUN server will fail and to establish connection between the peers we will need TURN server.
* TURN server is Traversal Using Relay NAT, and it is a protocol for relaying network traffic. TURN server will be used if STUN server will fail. It is used as an assist to establish connection between the peers.
* TURN servers are not public because of the costs which they can generate because of the traffic which is going through them.
* Before establishing the connection between peers, we need to exchange some data, the first one is the SDP (Session Description Protocol).
* The Session Description Protocol is a format to describing multimedia communication sessions for the purposes of session announcement and session invitation.
* It does not deliver the media data but is used for negotiation between peers of various audio and video codecs, source address timing information of audio and video.
* So, the working of webRTC in brief can be explained like this:
  + Caller takes his internet connection details from STUN Server and along with his SDP information creates a webRTC offer.
  + He sends this offer to the Callee via signaling server.
  + Callee sees the webRTC offer along with the internet and SDP details of Caller.
  + Callee prepares her answer to the offer. If she wants to establish a connection with the caller then she prepares answer with her SDP and internet details, which she takes from her STUN server and sends back to the Caller through the signaling server.
  + So both users once connected will know each other’s SDP as well as internet details.
  + Exchanging the SDP details is the first step when creating the direct connection using webRTC. The second step is to share internet connection details known as ICE candidates details which we get from STUN server.
* Exchanging information about the media (discussed above in offer/answer and SDP), peers must exchange information about the network connection. This is known as an ICE candidate and details the available methods the peer is able to communicate (directly or through a TURN server). Typically, each peer will propose its best candidates first, making their way down the line toward their worse candidates. Ideally, candidates are UDP (since it’s faster, and media streams are able to recover from interruptions relatively easily), but the ICE standard does allow TCP candidates as well.
* How to establish connection between Peers:



Module 8:





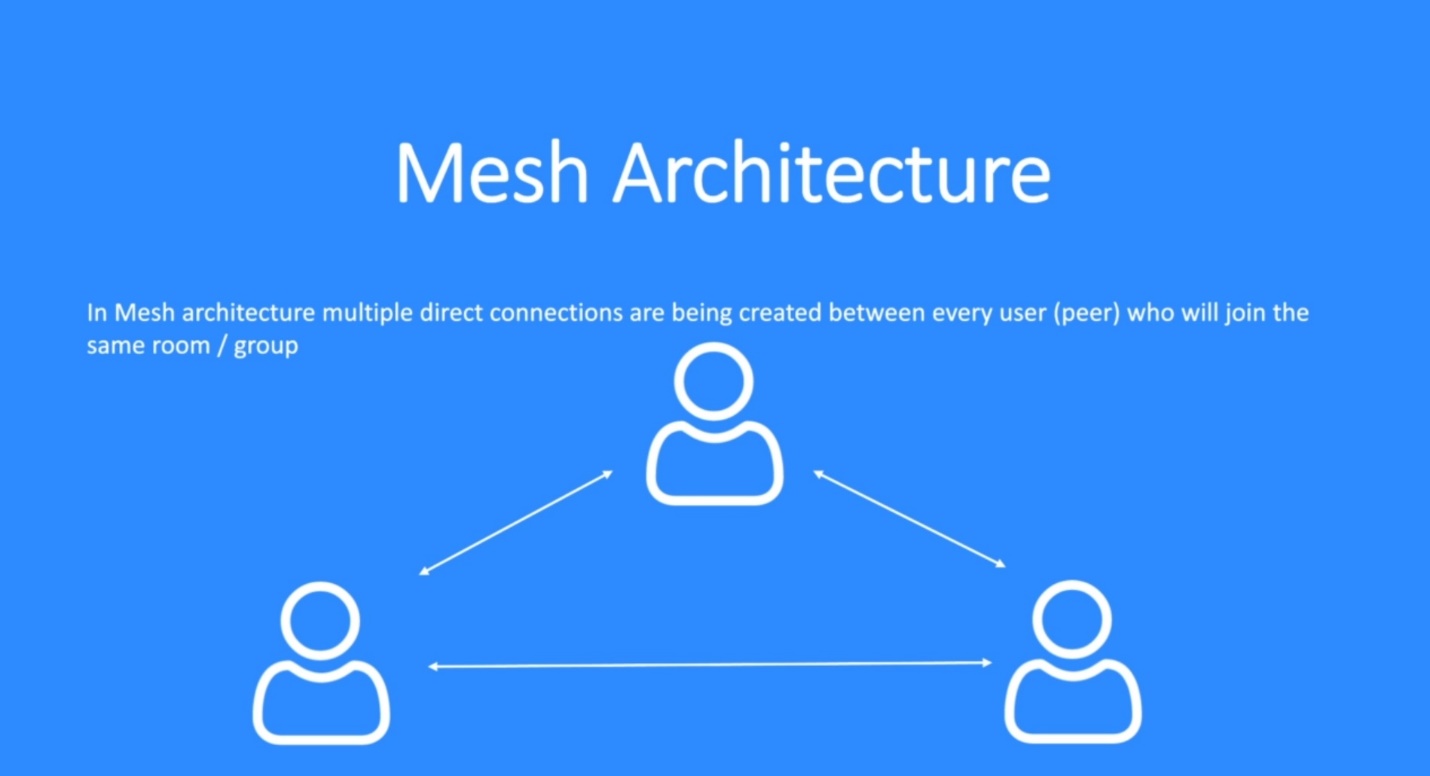
WebRTC Mesh Architecture:



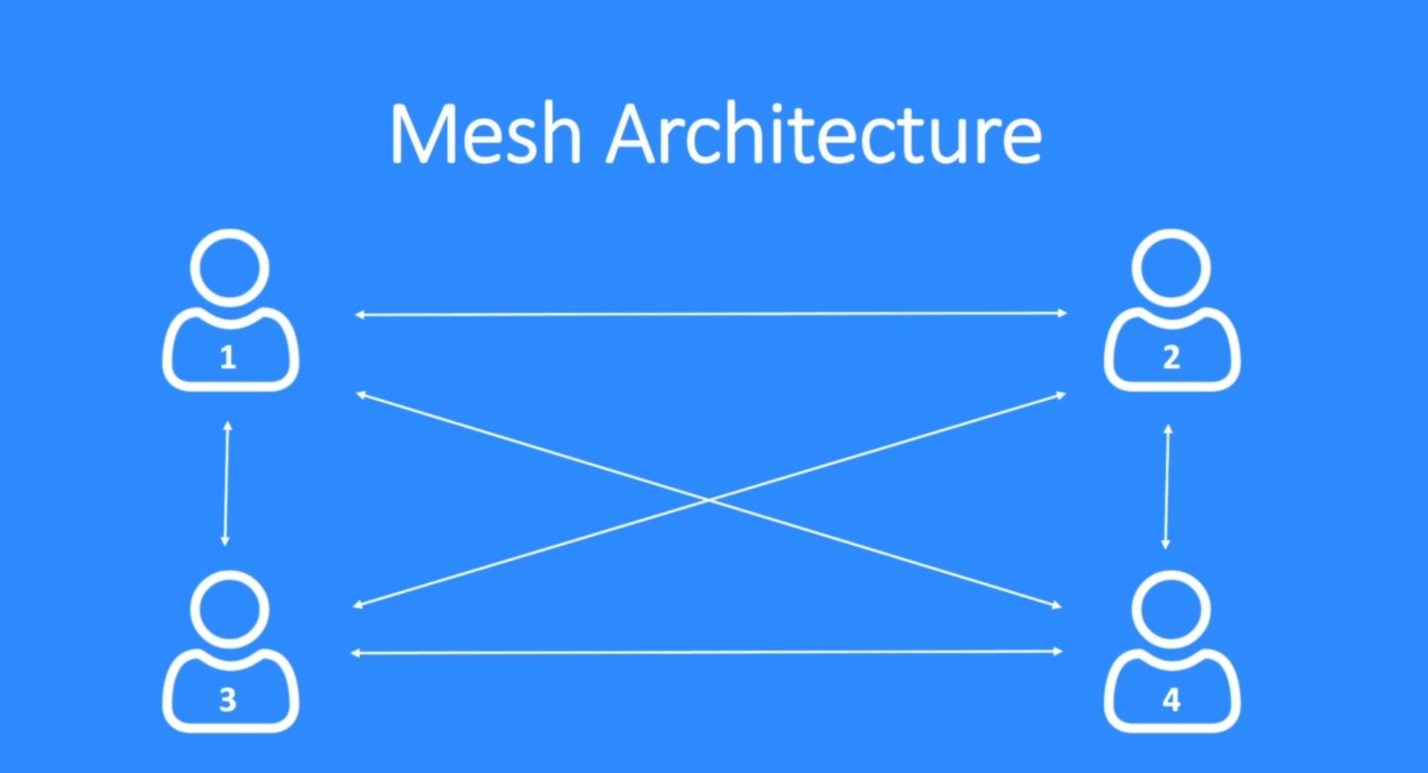
* To allow users to exchange their streams, video or audio doesn’t matter, we need to create a direct connection between these users. For that, we will be using the webRTC and exactly will create the webRTC mesh architecture.
* In its simplest way, webRTC allows us to set up a live peer-to-peer direct connection between two browsers to exchange video, audio and data between them.
* What would happen if we add another user to the connection? Then another? And another? This is defined as multi-party. With it, the rules of the game change. Here we create a group call.
* Strategies for group calls:



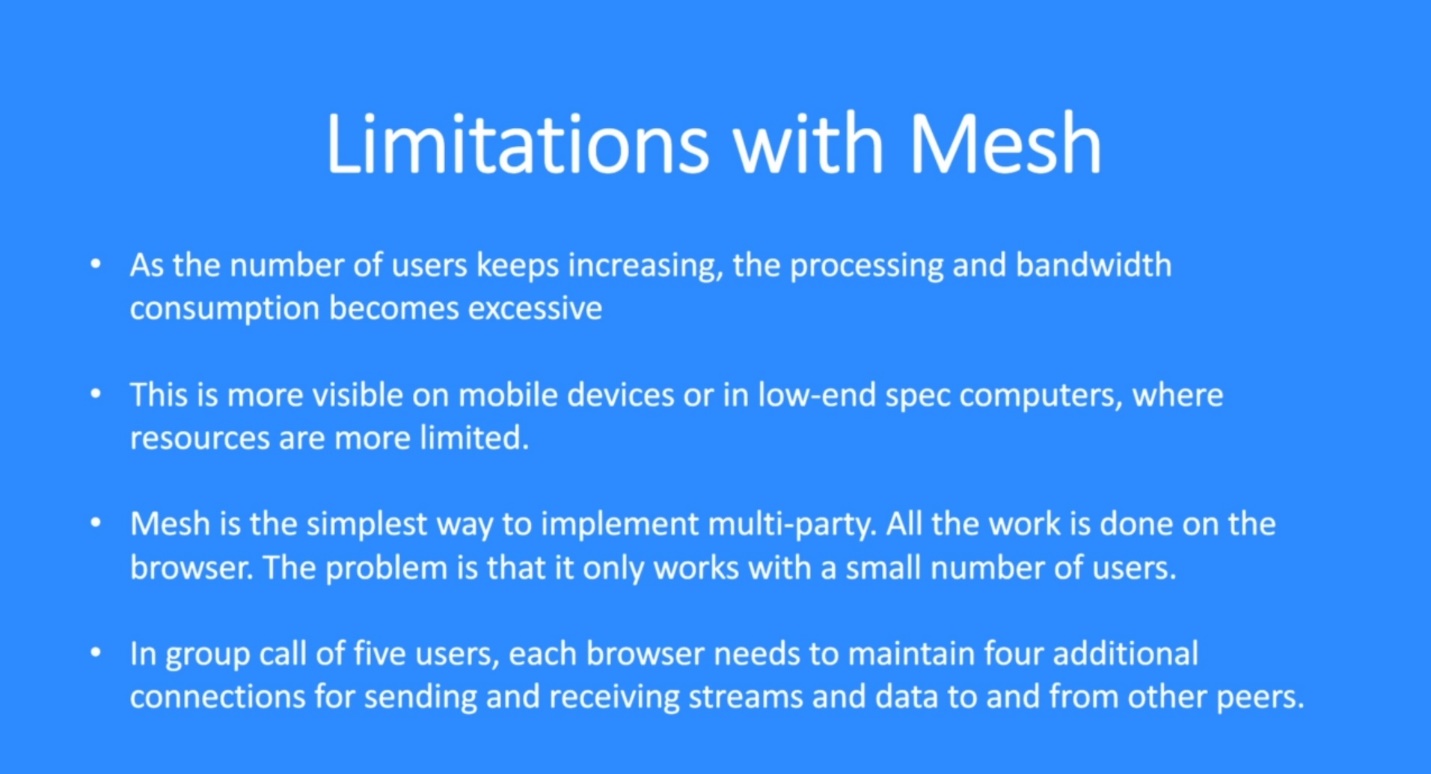
* + Mesh – Multiple Direct connections
    - If we have 4 users in a room, we will create 4 connection between all the users in the room. This is best suited for smaller applications. We will be using this for our app.
  + MCU – Multiple Control Units
    - MCU and SFU are to have more numbers of users in the same room. Big corporations and apps use these technologies.
  + SFU – Selective Forwarding Units
    - MCU and SFU are to have more numbers of users in the same room. Big corporations and apps use these technologies.
* Mesh architecture gives us possibility to create between every user connected to this room direct connection. If we want to create our app for maximum 4 users in a room, we will create direct connection between all 4 users. If we increase maximum number of users in a room the Mesh architecture will become complex and we will need to look for other architectures.



* In our case every user will be sending and receiving 3 streams from every other user. To do this simultaneously costs a lot of network and computer resources, that is the another reason, we should not use Mesh architecture for higher number of users.



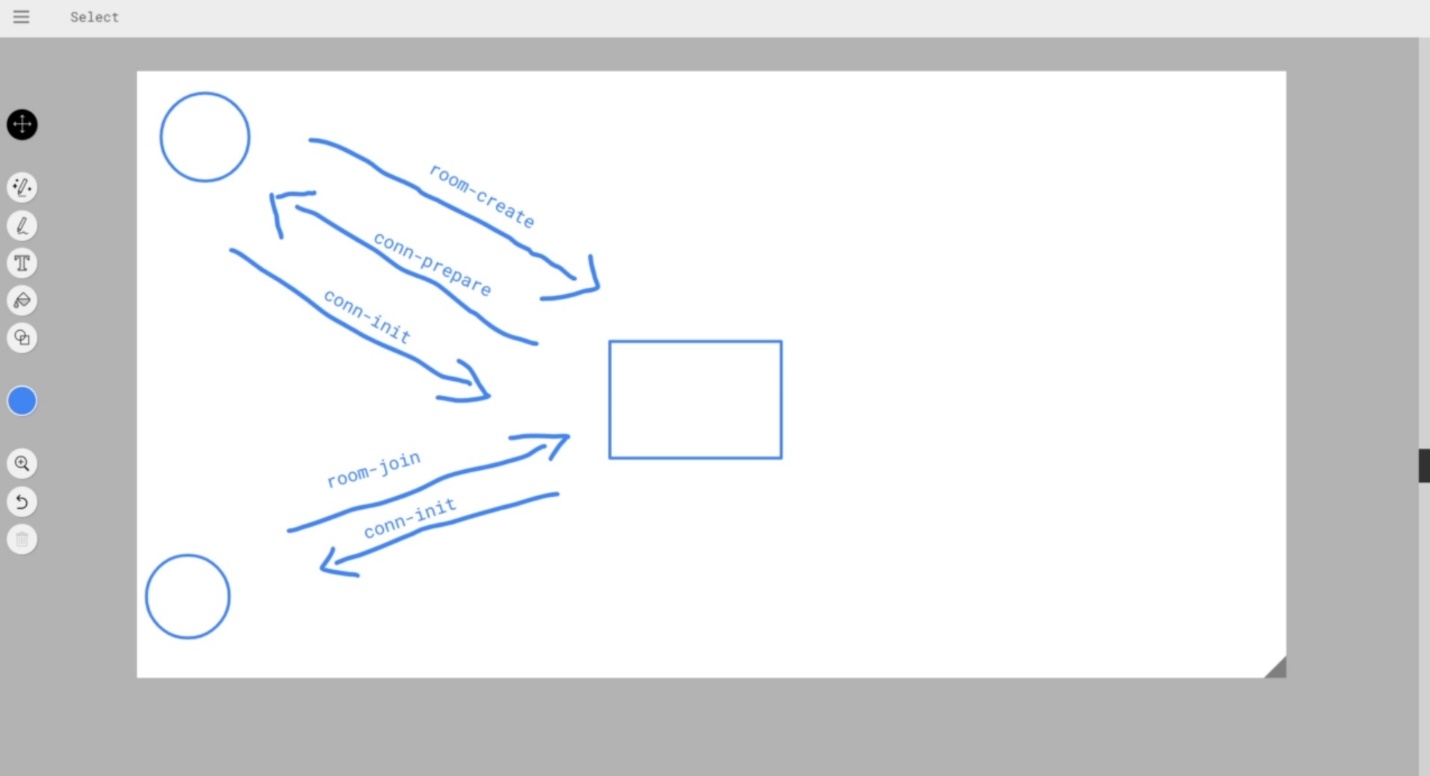
* Limitations with Mesh Architecture:



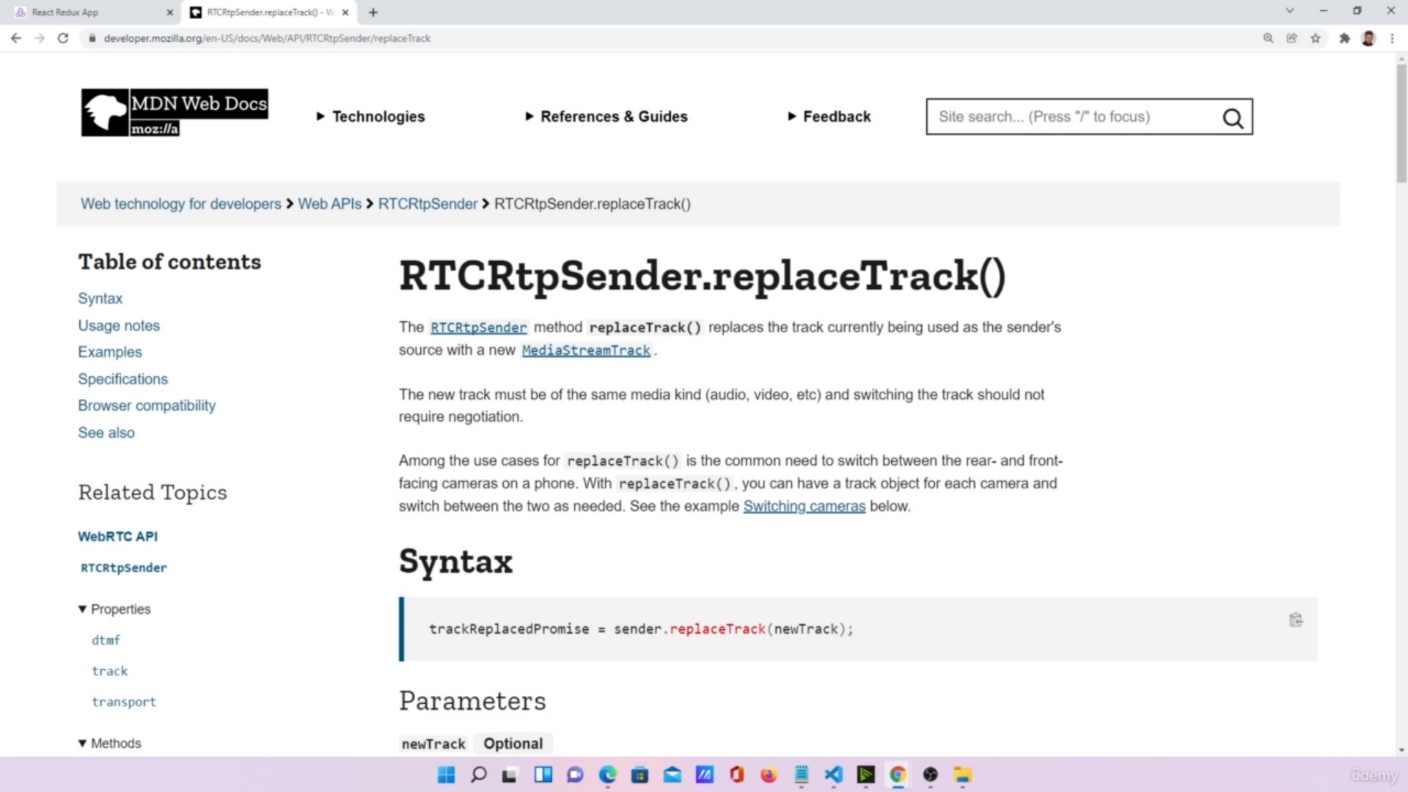
* To create our webRTC logic, we will use the simple-peer package. This is a simple wrapper for webRTC functions that are delivered by the browsers. This package makes it simple for us by doing webRTC configuration on it’s end. Read more about this package on [https://github.com/feross/simple-peer](https://github.com/feross/simple-peer%20)

Event related with the connections:

* Now we will discuss about the events that a user needs to emit, when a new user joins the room. They will be connected using the webRTC. The new user will enter the room, other users which are already in the room will need to prepare their incoming peer connection object of webRTc to inform this new user, that we are ready to establish the new direct connection.



* Switching outgoing Video Tracks in Active Peer Connections:



* On every Peer object which is already connected with other users with webRTC technology, we have a function replaceTrack(). But first we need to get all the sender. With simple-peer package it becomes very easy because when we are working with the native webRTC implementation, if we have the peer connection object, first we need to get the senders and the sender are responsible for sending the audio and video tracks to other users. Simple-peer has a little different logic on that.
* On switching the media stream (like we are doing in case of screen sharing), we need to change the track that is sent by the peer object to every other user connected directly, which is a little complicated process.
* asdf