Design The Twitch API

Design every API endpoint that's interacted with when a user is on an individual Twitch streamer's channel page, watching their livestream.

*Many systems design questions are intentionally left very vague and are literally given in the form of `Design Foobar`. It's your job to ask clarifying questions to better understand the system that you have to build.*

*We've laid out some of these questions below; their answers should give you some guidance on the problem. Before looking at them, we encourage you to take few minutes to think about what questions you'd ask in a real interview.*

Clarifying Questions To Ask

**Question 1**

***Q: I'm going to list the different features on a streamer's channel page to confirm that I'm not missing any important functionality. A user can watch the actual video of the livestream (and pause / unpause it), they can see the stream's chat in real time, they can send messages in the chat, they can see the streamer's channel info (description text, follower count, etc.), they can follow and unfollow the streamer, they can subscribe to and unsubscribe from the streamer (they have to pay to subscribe), they can see the current number of concurrent viewers on the stream, and they can see other recommended channels that they follow on the left navbar. Does that cover everything?***

A: Yes, but you can ignore the recommended channels on the left navbar. Let's focus purely on the functionality that's relevant to the livestream on the page.

**Question 2**

***Q: For subscriptions, as far as I know, a user subscribes on a monthly basis for a flat monthly fee. Is this correct?***

A: There might be different tiers of subscriptions, but you can capture that in some SubscriptionInfo entity that you don't need to explicitly define.

**Question 3**

***Q: A lot of the information on the page is very dynamic and warrants getting updated frequently and automatically, like follower count, concurrent viewer count, and obviously chat messages and the livestream itself. I'm assuming that we should design the various API endpoints with this in mind; does that sound good?***

A: Let's make sure that the actual video and chat are updated in real time, as you would expect. Then, let's try to have the number of concurrent viewers be updated every so often (say, every 30 seconds); we don't want that number to be constantly changing, because that could be jarring, especially if the streamer has thousands of viewers tuning in and out of the stream. As for the follower count, we can let that be updated on page refresh.

**Question 4**

***Q: To clarify, for following and subscribing, we do want to show the current user's follow / subscription status to the streamer, and we want to immediately reflect changes to it on the UI when the user clicks on the relevant "Follow" / "Subscribe" buttons, right?***

A: Yes, absolutely. If a user isn't following a streamer, they should see that state reflected on the page, and if they press the "Follow" button, they should see the updated state immediately. The same goes for subscribing. What we don't care about is updating the total follower count in real time; that's what can be done on page refresh.

**Question 5**

***Q: Regarding the chat, I know that users can send custom Twitch emotes as well as Cheermotes, which are special animated emotes that are purchased with Bits, Twitch's virtual currency. Should we handle these different kinds of emotes in our messaging API? What about other chat-related functionality, like being banned from the chat and the presence of chat moderators?***

A: You should handle basic Twitch emotes that are available to all Twitch users, but you can disregard custom emotes that are only available to subscribers as well as Cheermotes. In fact, you can disregard the concept of Bits altogether for this design. You can also disregard chat moderators, but you should cover the banning feature. Specifically, if a user is banned from the chat, they can't send messages, but they need to somehow be alerted about this in the chat box; they shouldn't be allowed to even try sending messages.

**Question 6**

***Q: Regarding the actual video, I'm not very familiar with how videos—especially live videos—work on the web at scale, but is it fair to assume that the Twitch backend will be able to continuously provide some VideoInfo entity that will contain the data needed to display the video on the UI?***

A: Yes, for the sake of this design, that's totally fine. Don't worry about the underlying complexity of displaying live video, like the shape of the data and how to display it exactly.

Our solution walkthroughs are meant to supplement our video solutions. We recommend starting with the video solution and using the walkthrough either as a point of reference while you watch the video or as a review tool if you need to brush up on this question's solution later on.

**1. Gathering Requirements**

As with any API design interview question, the first thing that we want to do is to gather API requirements; we need to figure out what API we're building exactly.

We're designing every API endpoint that's interacted with when a user is on an individual Twitch streamer's channel page, watching their livestream. Specifically, we need to handle:

* displaying the streamer's channel info (description text, follower count, etc.)
* following and unfollowing the streamer
* subscribing to and unsubscribing from the streamer
* seeing the live chat and sending messages; sending messages should only be allowable if the user isn't banned
* seeing the livestream and being able to pause / unpause it
* seeing the number of concurrent viewers of the stream, which should automatically be updated every 30 seconds or so

**2. Coming Up With A Plan**

It's important to organize ourselves and to lay out a clear plan regarding how we're going to tackle our design. What are the major, potentially contentious parts of our API? Why are we making certain design decisions?

Fortunately for us, the various functionalities that we have to support effectively lay out a step-by-step plan for us, so we'll simply follow that.

Of note, all of the API endpoints that we'll define will take in, by default, the caller's user-specific authentication token as an authorization header. This token will be used by the backend to identify which user is calling each API endpoint.

We'll also be passing a *channelId* as a parameter to all of the endpoints, which will be the unique username of the streamer in question.

**3. Channel Info**

This is the most straightforward piece of functionality on the page, since it only consists of displaying static data about the streamer.

The user will call the *GetChannelInfo* endpoint, which will return the relevant entity, *ChannelInfo*, to be displayed on the page.

**ChannelInfo**

* name: *string*
* description: *string*
* currentStreamTitle: *string*
* followerCount: *int*

This entity might have more fields, but these are the most important ones.

GetChannelInfo(channelId: string)

=> ChannelInfo

**4. Following**

The follow status is binary: either the user is following the streamer, or they aren't. Thus, we can support the follow functionality with a single endpoint that uses a toggle mechanism, whereby the backend sets the follow status to the opposite of what's currently stored in the database.

ToggleFollow(channelId: string)

=> FollowState (**FOLLOWING** or **NOT\_FOLLOWING**)

Naturally, this endpoint will be called when the user presses the "Follow" / "Unfollow" button.

Note that we haven't yet handled how to know what the user's follow state is. In other words, how do we know whether to show "Follow" or "Unfollow" to the user? See the **Relationship To Channel** section for details.

**5. Subscribing**

Subscribing is similar to following. However, unlike following, since there are more details to be provided when a user subscribes to a channel (subscription tier and payment information), we'll separate the acts of subscribing and unsubscribing into two endpoints.

CreateSubscription(channelId: string, subscriptionInfo: SubscriptionInfo, paymentInfo: PaymentInfo)

=> Subscription

CancelSubscription(channelId: string)

=> Subscription

Naturally, these endpoints will be called when the user presses the "Subscribe" / "Unsubscribe" button.

Note that we haven't yet handled how to know what the user's subscription state is. In other words, how do we know whether to show "Subscribe" or "Unsubscribe" to the user? See the **Relationship To Channel** section for details.

**6. Chat**

To handle the chat's functionality, we'll need two endpoints and a *Message* entity.

**Message**

* sender: *string*, the username of the sender
* text: *string*
* timestamp: *string*, in ISO format

StreamChat(channelId: string)

=> Message

SendMessage(channelId: string, message: string)

=> string | Error if user is banned

The *StreamChat* endpoint streams the stream's chat messages over a long-lived websocket connection and will be called once on page load.

The *SendMessage* endpoint will naturally be called whenever the user sends a message, and we can have the backend take care of timestamping messages and providing both the sender and the timestamp on the *Message* entity.

We can handle Twitch emotes by representing them with a special string format, like wrapping unique emote IDs in colons, as follows: **:emote-id:**. A Twitch a message will therefore look like this in string format:

"This stream is so fun to watch :kappa:"

The UI knows to detect this special string format and to display emotes appropriately. The UI also knows not to display messages sent by the user in question and received via *StreamChat*, since those messages will be displayed as soon as the user sends them via *SendMessage*.

While *SendMessage* returns an error if the user is banned from the chat, we won't actually allow the user to hit this endpoint if they're banned. That being said, we haven't yet handled how to know whether a user is banned. See the **Relationship To Channel** section for details.

**7. Video**

To display the actual video of the livestream, we'll open another long-lived websocket connection on page load, which will stream the video.

StreamVideo(channelId: string, videoQuality: VideoQuality)

=> VideoInfo

When this endpoint is called, we can imagine that the backend increases the concurrent-viewer count of the relevant stream in some database, which will be used in the next section to display the number of concurrent viewers to the user. When the long-lived connection is terminated (on tab close or page leave), the backend will decrease the relevant concurrent-viewer count in the database.

Lastly, when the user pauses the video, the UI still streams the video, but it simply doesn't display it.

**8. Concurrent Viewers**

Displaying the number of concurrent viewers watching a stream at any given time can easily be accomplished by polling an endpoint every 30 seconds or so, which reads from the database that stores every stream's concurrent-viewer count.

GetConcurrentViewers(channelId: string)

=> int

**9. Relationship To Channel**

There are a few pieces of functionality on the page that have to do with the relationship between the user and the streamer. Namely, whether the user is following the streamer, whether they're subscribed to the streamer, and whether they're banned from the streamer's chat.

One way to handle the follow and subscription states would be to fetch the user's profile info, which could contain all of their followed and subscribed streamers. This would be used with the streamer's name from *GetChannelInfo* to display the correct states (buttons) on the UI. The only problem is that a user could theoretically be following / subscribed to thousands of streamers, so we would maybe want to paginate the lists of followed and subscribed streamers, which would complicate things.

To make things simpler, and since we also have to handle the banned state, we can rely on a *GetRelationshipToChannel* endpoint, which will return the relevant entity, *RelationshipToChannel*, to be used to display the correct states on the page.

**RelationshipToChannel**

* isBanned: *boolean*
* isFollowing: *boolean*
* subscription: *Subscription | null*

GetRelationshipToChannel(channelId: string)

=> RelationshipToChannel

If the user is banned, we'll prevent them from sending chat messages (and calling the *SendMessage* endpoint) altogether.