Code & Tutorials

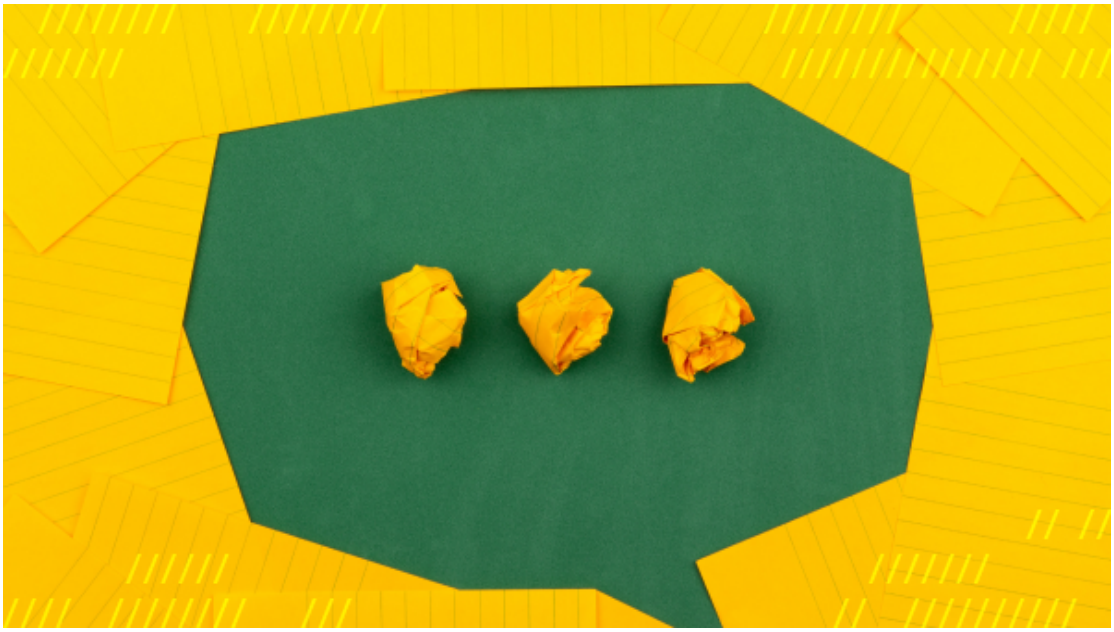
Add chat history to your Daily calls with only client-side React code

Show new call participants existing chat history using a custom React hook



Jess Mitchell

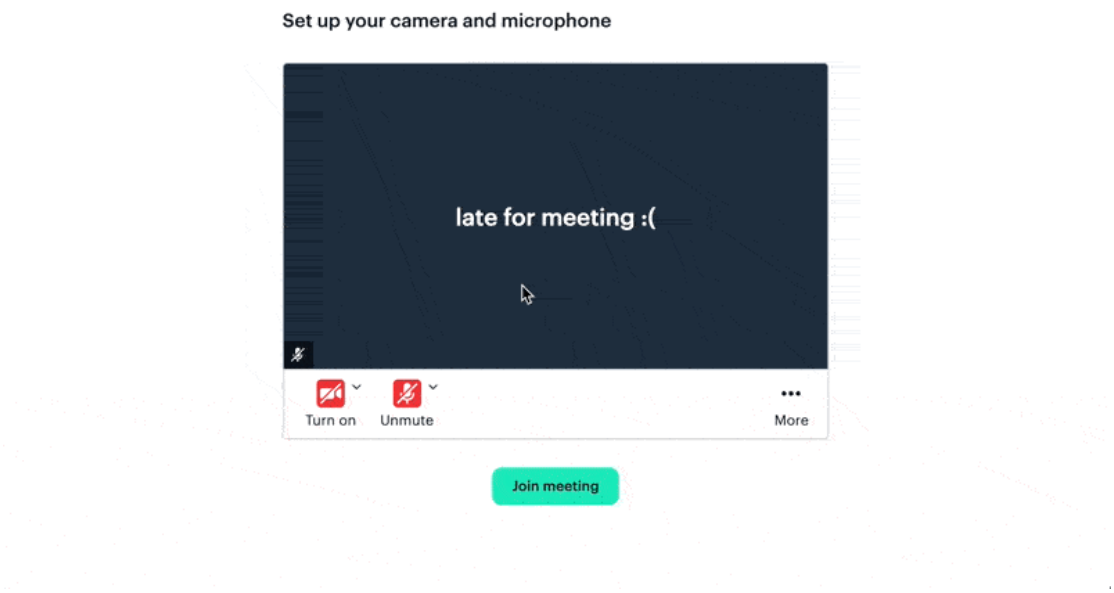
Published: 9 Dec 2021 (Updated: 24 Mar 2022) • 9 min read



Chat history is one feature that can be a bit challenging to incorporate to a custom Daily chat widget because it requires storing the existing chat messages in some way.

At Daily, we try to make sure you don't have to worry about this by doing it for you; our embeddable video chat widget — Daily Prebuilt — has chat history built right in, which means when a new participant joins a

call, the chat sidebar will include all the messages sent before they joined.



A late meeting attendee feeling blessed there's chat history in the call

When going the custom route, however, we know one potential downside for customers is having to figure out how to replicate Daily Prebuilt features in the simplest way possible. Thankfully, Daily Prebuilt is built with Daily's customizable call object, so we're able (and happy!) to share how we achieve features like adding chat history in a way you can easily replicate, too.

Recently, we came up with a way to add chat history in custom Daily chat widgets using only client-side React code — specifically, using our handy sendAppMessage method.

This means when a call participant joins a call where other participants

have already been sending text-based chat messages, the new participant can see the whole existing discussion. Not only that, it's possible without relying on those messages being stored in a database — only local state. 😎

Note: It also means HIPAA-compliant calls remain compliant even with chat turned on. (Bonus!🙌)

Today's agenda

In today's tutorial we'll introduce a custom React hook used in our Daily examples repo that can be used to add chat history to your custom Daily chat widgets, too!

Requirements

Since this is a custom hook, it will only be compatible with Daily apps that contain React code, whether that be React, a React framework like Next.js, or React Native.

That said, the essence of this solution can be written for other frameworks, so if you can read React code, this may still be a helpful tutorial for you.

Getting started with pseudo code

Before we get to the actual hook, let's describe what we're trying to achieve and how we plan to break that down into smaller steps.

But first, an analogy

When it comes to the chat history feature, we can think of it like when you get to class late, the lecture has already started, and your

classmates are already writing notes. You have a test coming up so you ***really*** need to know what has been said already.



You sit down between a couple of people and ask the person to your left if you can have a copy of their notes.

“Sorry, I just got here, too!” they respond.

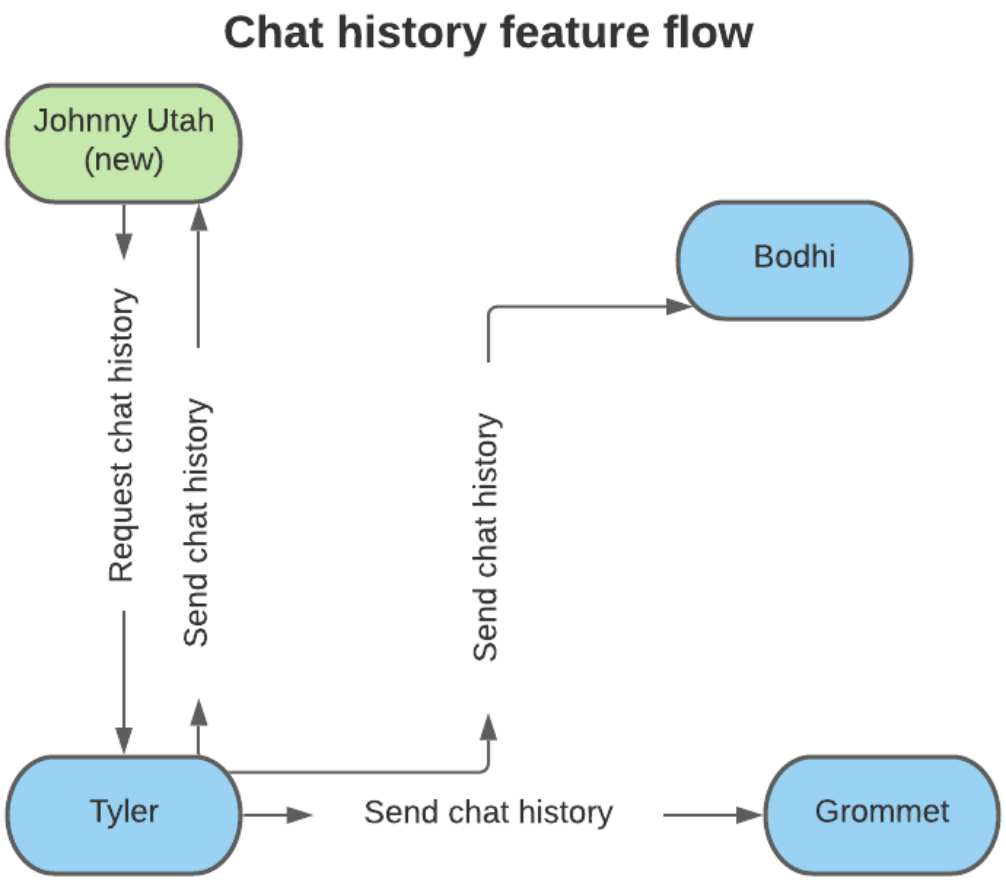
Whoops, picked the wrong person to ask. So, then you turn to the person on your right and say, “Hey, can I have a copy of your notes?”

“Sure, here’s a copy right here,” they say (thankfully).

That’s where the analogy falls apart because they probably don’t have a photocopier next to them, but you get the idea.

We're going to do the exact same thing to build our chat history feature using three main steps:

1. When a participant joins the call, they can pick someone in the call randomly and ask for the chat history they missed before joining.
2. When that new participant sends out a request, the random participant receiving the request will send their chat history in response. ***However***, that chat history that gets sent out won't be sent to just the person who requested it; we'll send it to ***everyone*** in the call. (This helps make sure everyone's chat is up-to-date and lets anyone waiting for the chat history get it as soon as possible.)
3. When the state is sent, it must then be received by the person who just joined and stored in their local state, as well. Once this is done, the chat history can be displayed in the UI.



Johnny asking the locals what they've been talking about

Getting to know `useSharedState`

The `useSharedState` custom hook can be used for sharing any state in a Daily call, but we'll focus on the chat history use case today.

One code example that uses the `useSharedState` custom hook for chat history can be found in the custom Daily text chat demo app. Specifically, it's used in the `ChatProvider` when the app's initial chat state is set up.

Follow the README included there to run the demo locally or try it on our deployed version.

Let's get to the code

To see the complete hook, refer to the `useSharedState.js` file in Daily's `examples` repo.

We'll step through the code in more manageable pieces here to see how it works.

First things first: we need our instance of the Daily call object, which manages all the details of our call and connects you to the Daily API.

```
export const useSharedState = ({
    initialValues = {},
    broadcast = true,
}) => {
 const { callObject } = useCallState();
 ...
```

In this example, the `callObject` is created in another file and imported to our hook. We won't go into details but keep in mind the `callObject` needs to be available, whether it's passed as a prop or imported.

Note: If you're not using the same app structure as the `examples` repo, you'll need to tweak this line of code that imports the call object.

Next, let's look at defining our steps as described above (requesting state on join and sharing/receiving state).

```
// Add event listeners to the Daily call object if it exists
useEffect(() => {
  if (!callObject) return;
  callObject.on('app-message', handleAppMessage);
  callObject.on('joined-meeting', handleJoinedMeeting);

  return () => {
    callObject.off('app-message', handleAppMessage);
    callObject.off('joined-meeting', handleJoinedMeeting);
  };
}, [callObject, handleAppMessage, handleJoinedMeeting]);
```

Here, we have an effect that is triggered when `callObject` or two functions (`handleAppMessage` and `handleJoinedMeeting`) change. This means, when `callObject` becomes truthy when initiated, it will add a Daily event listener for the following events:

- `app-message` : This is triggered when anyone in the call sends a message with sendAppMessage. (More on that below.)
- `joined-meeting` : This is triggered when the local participant has officially joined the call.

Joining a call

Let's step through this in the order it would happen if you were joining the call after it has already started. That means step one would be you joining a call, and the `joined-meeting` event firing, which would call the `handleJoinedMeeting` shown in the effect above.

```
// whenever local user joins, we randomly pick a
// participant from the call and request state from them.
const handleJoinedMeeting = useCallback(() => {
  const randomDelay = 1000 + Math.ceil(1000 * Math.random());

  requestIntervalRef.current = setInterval(() => {
    const callObjectParticipants = callObject.participants();
    const participants = Object.values(callObjectParticipants);
```

```
    const localParticipant = callObjectParticipants.local;

    if (participants.length > 1) {
      const remoteParticipants = participants.filter(
        (p) =>
          !p.local &&
          new Date(p.joined_at) < new Date(localParticipant.join
      );

       // don't send a message if 0 remote participants
       if (remoteParticipants?.length === 0) return;

      const randomPeer =
        remoteParticipants[
          Math.floor(Math.random() * remoteParticipants.length)
        ];

      // send the request for the shared state
      callObject.sendAppMessage(
        {
          message: {
            type: 'request-shared-state',
          },
        },
        randomPeer.user_id
      );
    } else {
      // if there is only one participant, don't try
      // to request shared state again
      clearInterval(requestIntervalRef.current);
    }
  }, randomDelay);
```

Let's step through what happens in `handleJoinedMeeting`:

- First, we pick a random delay. This is to avoid performance issues that can happen when a bunch of people join at the same time, all request the chat history, and cause a bottleneck of requests to be processed by the receiver.

Note: The chances are slim but we want to avoid it anyway!

```
const randomDelay = 1000 + Math.ceil(1000 *
Math.random());
```

- Next, we set up our interval for requesting the shared chat state. We need an interval instead of a one-time request in case you're

that unlucky person who asked someone who also doesn't have the chat history; you'll need to be able to ask another person.

```
requestIntervalRef.current = setInterval(() => {
    ...
 }, randomDelay);
```

- Then, we check how many people are in the call. If you're alone, we won't bother making a request because there's no one there to share state (and, therefore, no state to share! 🤯)

```
if (participants.length > 1) {
      ...
    } else {
      // don't try to request shared state
      // again for single participants
      clearInterval(requestIntervalRef.current);
    }
```

- After that, we get a list of all the remote participants who joined before us and pick one at random.

```
const remoteParticipants = participants.filter(
    (p) =>
        !p.local &&
    new Date(p.joined_at) < new Date(localParticipant.joined_at)
);

const randomPeer =
    remoteParticipants[
        Math.floor(Math.random() * remoteParticipants.length)
    ];
```

- Finally, we send the request for the chat history using sendAppMessage. That random participant will then receive an `app-message` with the type `request-shared-state`.

```
callObject.sendAppMessage(
    {
```

```
        message: {
            type: 'request-shared-state',
        },
    },
    randomPeer.user_id
);
```

Receiving the request: Reading you loud and clear

As you'll recall, our `callObject` has a Daily event listener on it for `app-message`, which triggers the `handleAppMessage` callback: `callObject.on('app-message', handleAppMessage);`

```
const handleAppMessage = useCallback(
  (event) => {
    switch (event.data?.message?.type) {
      case 'request-shared-state':
        // do not respond if there is no available state
        if (!stateRef.current.setAt) return;

        // send the shared-state to everyone in the call
        callObject.sendAppMessage(
          {
            message: {
              type: 'set-shared-state',
              value: stateRef.current,
            },
          },
          '*'
        );
        break;
      case 'set-shared-state':

        ...
    }
  },
  [stateRef, callObject]
);
```

As we just covered, once that chat history request is sent, the receiver gets an `app-message` with type `request-shared-state`, which gets us into the first case in our `handleAppMessage` switch statement:

```
const handleAppMessage = useCallback(
   (event) => {
     switch (event.data?.message?.type) {
       case 'request-shared-state':
       ...
```

Once in, we check if there is actually some state stored to be shared by checking our `stateRef` value. (More on that below.)

`if (!stateRef.current.setAt) return;`

If there isn't, we'll exit the switch statement and if there is, we'll continue by sending another `app-message` to everyone in the call with that chat history state.

```
callObject.sendAppMessage(
    {
        message: {
            type: 'set-shared-state',
            value: stateRef.current,
        },
    },
    '*'
);
```

Notice how the `app-message` type is now `set-shared-state` ? That brings us to our third step: receiving the chat history.

Receiving our chat history

This latest `app-message` with type `set-shared-state` will once

again trigger the `handleAppMessage` callback, but this time fall into the switch statement's second case:

```
const handleAppMessage = useCallback(
  (event) => {
    switch (event.data?.message?.type) {
      case 'request-shared-state':
        ...
      case 'set-shared-state':
        clearInterval(requestIntervalRef.current);
        // do not respond if the current state is
        // more up-to-date than the state being shared
```

```
        if (
          stateRef.current.setAt &&
          new Date(stateRef.current.setAt) >
            new Date(event.data.message.value.setAt)
        ) {
          return;
        }
        // update the state
        setState(event.data.message.value);
        break;
    }
  },
  [stateRef, callObject]
);
```

As a final step, we make sure the timestamp on the chat history state being shared is newer than any state we may have already and, if so, set our local state with what has been shared.

Another effect we have will then get triggered because of the state change:

```
useEffect(() => {
  stateRef.current = state;
}, [state]);
```

This will map our state to the `stateRef`, a `ref` initialized at the top of `useSharedState`. The main benefit of this is improving performance: we can use the `stateRef` as a dependency in our other

functions/hooks without triggering re-renders when `stateRef` changes.

And just like that, we have chat history with ***only*** client-side code! Pretty cool, right? 💫

Wait a minute, how do I actually use this hook?

Okay, fine, there is one more step. We know how our hook works but we need to actually use it in our app.

If we go back to our text-chat demo app and look at the `ChatProvider`

context, we initialize our chat history state in the app by using the `useSharedState` hook.

```
import {
    useSharedState,
} from '@custom/shared/hooks/useSharedState';
...
export const ChatProvider = ({ children }) => {
 ...
 const { sharedState, setSharedState } = useSharedState({
   initialValues: {
     chatHistory: [],
   },
   broadcast: false,
 });
```

Once that's done, `setSharedState` is available to update the `sharedState` value, as well. For example, after you've joined, you can start sending and receiving new messages that will become part of your shared chat history state.

Resources

We hope this helps simplify your custom Daily chat widgets without having to store chat messages in a dedicated backend. And, don't forget, you can use this hook for any shared state — not just chat! 💪

If you're looking for more resources for how to actually build a chat widget, check out some of our previous posts:

- Build a flexible React chat input using Daily's sendAppMessage
- Three ways to add chat to your video calls with the Daily API
- Build a custom chat widget in Vue with Daily's sendAppMessage

If you'd like to learn more about how Daily Prebuilt's chat was built, check out this conference talk from our frontend engineer, Christian! 👋

Get updates from Daily

Don’t miss a thing! Subscribe to our monthly newsletter and we’ll keep you up-to-date with the latest product releases, tutorials, and more from Daily.

Email*

Country*

Join peerConnection - our video/WebRTC developer community.

By clicking submit below, you consent to allow Daily.co to store and process the personal information submitted above to provide you the content requested.

Submit

More in Code & Tutorials

Working with video call participants’ media tracks

10 Oct 2022 - 9 min read

Stream with HLS while recording to a custom S3 bucket using Daily

30 Aug 2022 - 5 min read

Tutorial: Integrate CometChat’s chat widget into a Daily video app for an enriched chat experience

22 Aug 2022 - 13 min read

See all 99 posts →



Welcoming Sarah & Varun: A chat with new leaders at Daily

Hear from Sarah Milstein and Varun Singh, our new execs in engineering and growth, on why they joined and what's ahead.

Daily Team
13 Dec 2021 – 6 min read

Customer Stories

How Beepboop made language-learning classes 40% more effective with Daily

Beepboop switched from traditional video conferencing software to Daily to build a web app for their unique audio-only language classes.

Francine Boulanger
6 Dec 2021 – 6 min read

Latest Posts Twitter