Code & Tutorials

Tips to improve WebRTC video call browser performance

Recommendations for developers building video chat apps with Daily's core APIs



Jess Mitchell, Christian Stuff

Published: 21 Jul 2021 (Updated: 2 Jun 2022) • 14 min read



Christian Stuff, Daily's blazingly fast and talented front-end engineer, recently did some investigating into how to improve Daily Prebuilt's performance. Here's what he found.

One of our primary goals at Daily is to help our customers embed reliable and easy-to-use video calls into their apps in the shortest

developer time possible. One way we've found to do this is by offering Daily Prebuilt.

Daily Prebuilt is Daily's ready-to-use, embeddable video chat that can be added to any app with just a few lines of code. This is in comparison to our call object option, which enables customers to build their own custom video calls with our core APIs.

Basically, if Daily Prebuilt is your favourite meal served at a restaurant, Daily's customizable call object is a bag of groceries and spices with a recipe included. Which one you choose ultimately depends on what your goal is and how many of your own flavours you want to include.



Daily Prebuilt

Redesigning and improving Daily Prebuilt

Recently, we decided to redesign Daily Prebuilt to incorporate some helpful customer feedback we knew would substantially improve Daily Prebuilt's UX.

What some customers might not realize is that Daily Prebuilt is actually

built with our call object. (Yes, we are one of our own customers!) This new, redesigned version also gave us the opportunity to do a deeper dive on the most performant ways to implement Daily's call object.

Along the way, we've found several solutions to drastically improve Daily Prebuilt's performance, especially on mobile devices and the problem child of browsers for WebRTC: Safari.

To help our customers avoid having to learn these lessons on their own, we'll be covering our most important solutions related to improving performance while using Daily's call object mode. Many of these are also applicable to WebRTC development in general.

Who will find this tutorial helpful?

You'll find this tutorial useful if you're:

- A current Daily customer building a custom video chat app
- Shopping around for a video API to help you build a custom video experience
- Interested in learning more about browser quirks related to video calls

Who will find this tutorial helpful?: Tech stack edition

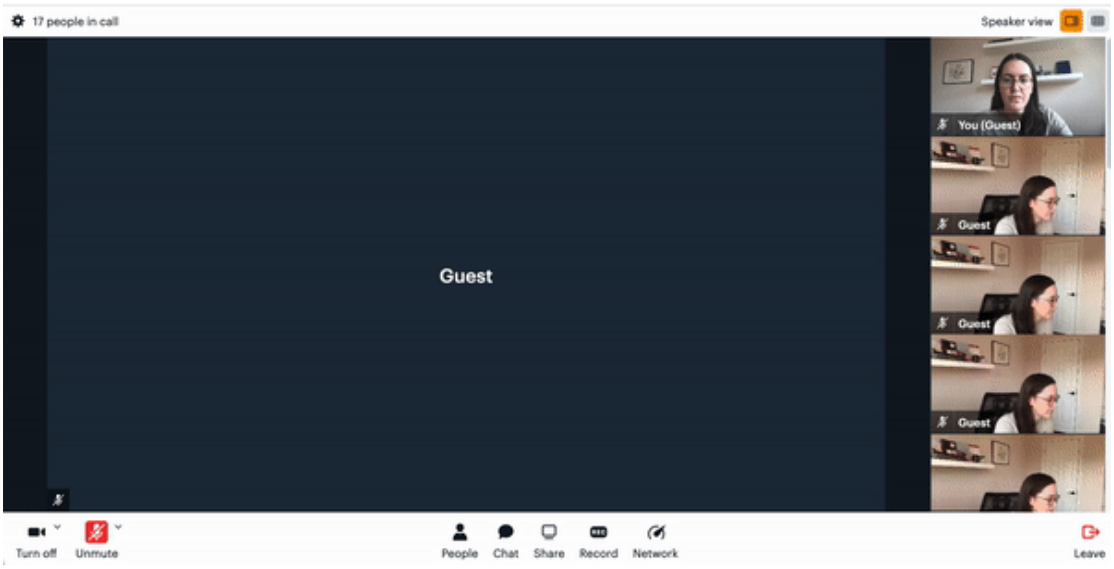
We're fond of React and Next.js at Daily, but these performance tips are mostly front-end framework-agnostic. Regardless of what you're building your web app with, you can apply these tips to get the most out of your Daily video calls.

Excuse me, sir: What exactly is the problem?

Before we dive into all the performance optimizations we used to improve Daily Prebuilt, let's first take a look at how we knew we had a problem.

One of the main motivators for improving performance has been due to our push to increase call sizes. (15,000 participants now, have you heard? 😎) All these additional participants create a new problem: loading participant media. For example, if you're in a Daily call in speaker mode and scroll through the participant list, videos should load efficiently as they come into view to create a positive user experience.

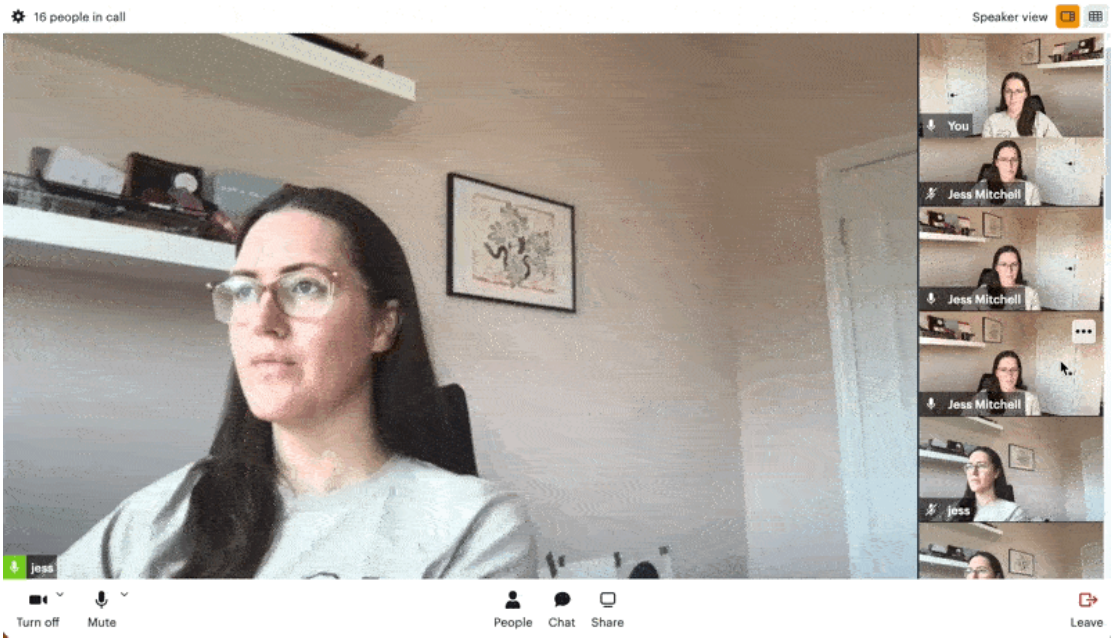
Here's an example of participant bar scrolling in one of the earliest internal versions of the new Daily Prebuilt:



Before: Slow video loads while scrolling the participant bar.

We felt the participant bar needed to load the videos faster and more reliably, as a user scrolls through. (Imagine that call with 1000 participants; no one's got time for that!)

For comparison's sake, let's take a look at the participant bar after we implemented the following performance improvements. It quickly recovers from a scroll much more efficiently.



That cool, confidence of knowing your participant videos are going to load

Another example of slow performance while the new Daily Prebuilt was in development was on mobile. We noticed issues like flickering videos, crackling audio, and delays to user interactions, like button presses. (We might have even heard the word "janky" a couple times during internal testing and cringed.)

We knew we could do better!

Improving performance in custom Daily apps

In this tutorial we'll cover 7 main lessons we learned about improving performance in a custom video chat app. These lessons include:

1. **Batching** `daily-js` events, i.e. participant-related events that trigger re-renders
2. **Manually subscribing** to media tracks in specific use cases
3. Using **virtual scrolling** in scrollable elements containing videos
4. Using **pagination** to limit the number of videos shown at a time
5. **Memoizing** elements prone to re-renders
6. **Reducing** how often media elements are added and removed from the DOM
7. **Checking if a video is paused** before playing it

1. Batching daily-js events

daily-js is an events-based library. Whenever a participant joins a call, updates their device, becomes the active speaker, or anything in between, `daily-js` sends an event so you can decide how to update your call UI. (Check out our docs to see a full list of Daily events. 👀)

For example, you can use the `participant-joined` event if you want to listen for when a new participant joins the current call.

```
callFrame.on('participant-joined', (event) => {
    console.log('participant-joined event', event);
    // add another video tile for the new participant
})
```

Adding an event listener for 'participant-joined' events

The event payload itself will look something like this:

```
const participantJoinedEvent = {
  action: 'participant-joined',
```

```
  callFrameId: '16257681634230.996506976694651',
  participant: {
    audio: false,
    audioTrack: false,
    cam_info: {},
    joined_at: 'Thu Jul 08 2021 14:18:21 GMT-0400 (Eastern Dayli
    local: false,
    owner: false,
    record: false,
    screen: false,
    screenTrack: false,
    screen_info: {},
    session_id: 'd8c55cfb-5eff-4f92-ccee-004989f6b077',
    tracks: { audio: {}, video: {}, screenVideo: {}, screenAudio
    user_id: 'd8c55cfb-5eff-4f92-ccee-004989f6b077',
    user_name: 'Name',
    video: false,
    videoTrack: false,
    will_eject_at: 'Wed Dec 31 1969 19:00:00',
  },
};
```

The `participant-joined` event structure

If a bunch of people all join a meeting you're in at the same time, you'll receive a `participant-joined` event for each and every one of them. It can be a lot to handle in calls with dozens (or hundreds!) of people! 😱

Now let's say you're updating a data store for each of these `participant-joined` events, such as updating a `participants` array in a React store. Updating the state for every `participant-joined` event would trigger a re-render for each one, which is not ideal. Instead, you can avoid this by batching `participant-joined` events

and only update your state every 250ms with all the newly joined participants at once.
Let's take a look at how this could look in React:

```
const joinedSubscriptionQueue = [];

const handleParticipantJoined = ({ participant }) => {
  joinedSubscriptionQueue.push(participant);
};
```

```
const joinBatchInterval = setInterval(() => {
  if (!joinedSubscriptionQueue.length) return;

  // Update participants list in React state based on the `joine
  // Reset queue
}, 250);

callFrame.on('participant-joined', handleParticipantJoined);
```

An example of collecting new participants before making a state update

In this solution, the `participant-joined` event triggers the `joinedSubscriptionQueue` to update. Then, an interval is set that waits 250ms for any other new participants to be added to the `joinedSubscriptionQueue` before actually triggering any state changes.

Even with such a small interval of 250ms, batching event-based changes can improve performance, especially in large calls.

One thing to keep in mind, too, is that when you should actually use event batching will depend on how you are responding to Daily events in your app. Your own implementation will vary based on what is triggering the most avoidable re-renders or UI updates.

In addition to `participant-joined`, batching is useful in other Daily events that are triggered often in calls, such as:

- `participant-updated`
- `participant-left`
- `track-started`
- `track-stopped`

Advanced Daily events batching: Manual track subscriptions

Let's take a look at a more advanced example of Daily event batching that uses manual track subscriptions. This is considered more advanced because Daily manages track subscriptions for you by default; turning on manual track subscriptions will add quite a bit of complexity to your

state management and is only recommended in specific use cases.

If we take the example from above, we can update it for implementing manual track subscriptions for new participants. Let's say we want to turn on track subscriptions for every new participant when they join, batching the subscriptions could look something like this:

```
const joinedSubscriptionQueue = [];

const handleParticipantJoined = ({ participant }) => {
  joinedSubscriptionQueue.push(participant.session_id);
};

const joinBatchInterval = setInterval(() => {
  if (!joinedSubscriptionQueue.length) return;

  const ids = joinedSubscriptionQueue.splice(0);

  const participants = callFrame.participants();

  const updates = ids.reduce((o, id) => {
    const { subscribed } = participants?.[id]?.tracks?.audio;
    if (!subscribed) {
      o[id] = {
        setSubscribedTracks: {
          audio: true,
          screenAudio: true,
          screenVideo: true,
        },
      };
    }
    return o;
  }, {});


  callFrame.updateParticipants(updates);
}, 250);

callFrame.on('participant-joined', handleParticipantJoined);
```

Manually turning on track subscriptions for new participants with event batching

In the code snippet above, we create a queue of new participants every 250ms and use the `updateParticipants` method to update all the new participants' subscribed tracks at the same time.

This version of event batching helps avoid updating each and every new participant individually without creating any noticeable UI delays in displaying participant videos.

2. Manual track subscriptions: Only subscribe to visible participants

You may be wondering about when to use the example right above, which demonstrates manual track subscription. By default, Daily will handle track subscriptions for you and, for the most part, this is the best solution; let us do the work for you.

In some situations, however, you may want to take advantage of Daily's call object option to manually subscribe to media tracks for participants. This can be useful for improving performance in large calls, as well as certain features like "breakout rooms" where a call is broken into sub-groups. (But, again, most apps do not need to use this feature!)

In terms of performance, manually subscribing or unsubscribing from tracks is useful in large calls where many videos are not visible. Since the video is not visible, you can unsubscribe from receiving the video tracks from those participants and reduce the amount of data being sent and received related to the call. Only when the participant is moved to being on-screen will you need to re-subscribe to the participant video track.

Using manual track subscription requires two main Daily methods:

- `setSubscribeToTracksAutomatically(false)` : Be sure to pass `false` as a parameter to override the default, which will automatically subscribe to all tracks.
- `updateParticipant()` or updateParticipants(), which updates several participants at once. To update which the tracks are subscribed to for a participants, pass a `setSubscribedTracks` value like so:

```
callFrame.updateParticipant(
```

```
  “participant-id-string",
  {
    setSubscribedTracks: {
      audio: true,
      video: false,
      screenVideo: false,
    },
  }
);
```

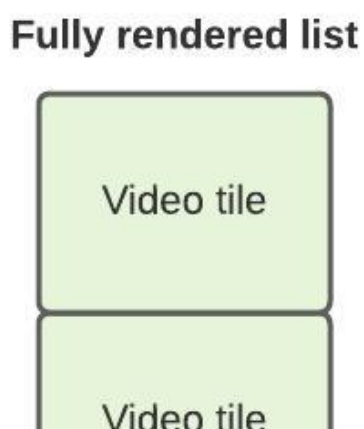
Manually subscribe with tracks for a single participant with the `updateParticipant` method

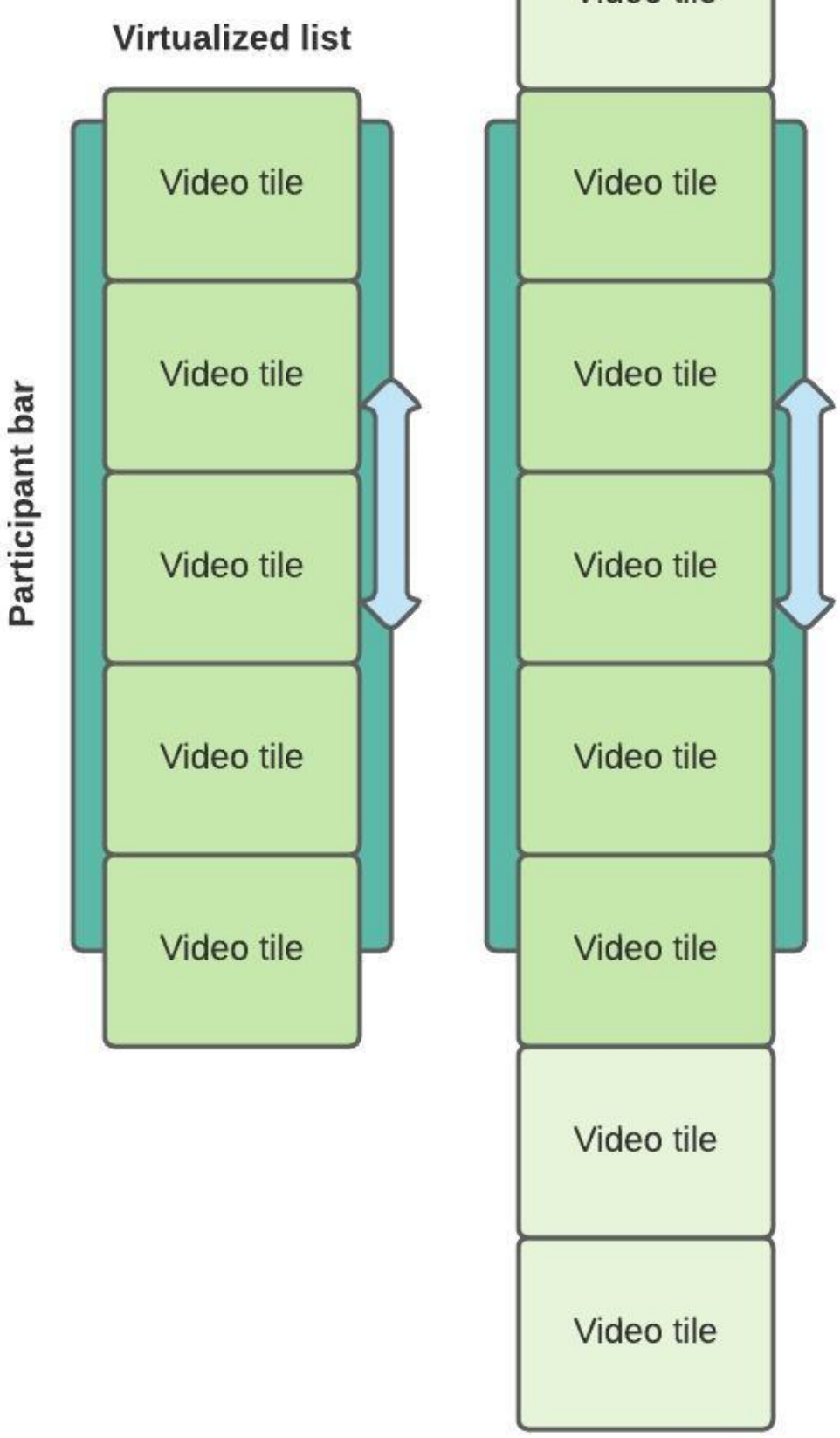
Note: If you're not sure if using manual track subscriptions is a good option for your app, feel free to ask our team. We're happy to help!

3. Using virtual lists to reduce how many `<video>` elements are rendered

One major update with the new Daily Prebuilt design was making the participant bar vertically scrollable. As Daily increases the call size limits, only rendering participant tiles that are actually visible provides a huge performance win. This can be achieved with virtualized lists.

Virtualized lists (or virtualized scrolling) refers to a list of items where only the visible subset of items is actually rendered in the DOM. As the list is scrolled through, new items (DOM elements) are rendered as they are scrolled into view (or into a “pre-render” area). Conversely, as DOM elements are scrolled out of view, they are destroyed. The goal here is to only render what is visually relevant to the user, and update the list as it is scrolled through.





Comparing the video elements renders with virtually scrolling vs. the full list of videos

The Daily Prebuilt participant bar is a good candidate for virtualized scrolling because there can be dozens or even hundreds of participants in the participant bar. Depending on the size of your browser window, you may only actually see 5-10 participants at a time.

To implement virtualized scrolling, there are thankfully several options. If you don't mind doing some math, you can calculate where you expect the item to be on the screen based on its position in the list, the size of the element, the scroll position, and the height of the container element. If it is visible, you can render it and otherwise not. (Check out this blog post on virtualized lists that explains this well.)

To simplify virtualized scrolling even more, you can also use one of the many libraries that will handle the rendering logic for you. React, for example, has several available libraries, like react-virtualized.

Lastly, if you're using Daily's React Native library, react-native-daily-js, you can use React Native's `FlatList` or `SectionList` components. They are both wrapper components for React Native's Virtualized List component and will handle all the rendering logic for you, as well.

4. Limiting video feeds with pagination

In Daily Prebuilt on desktop, we limit the rendered participant tiles in two ways:

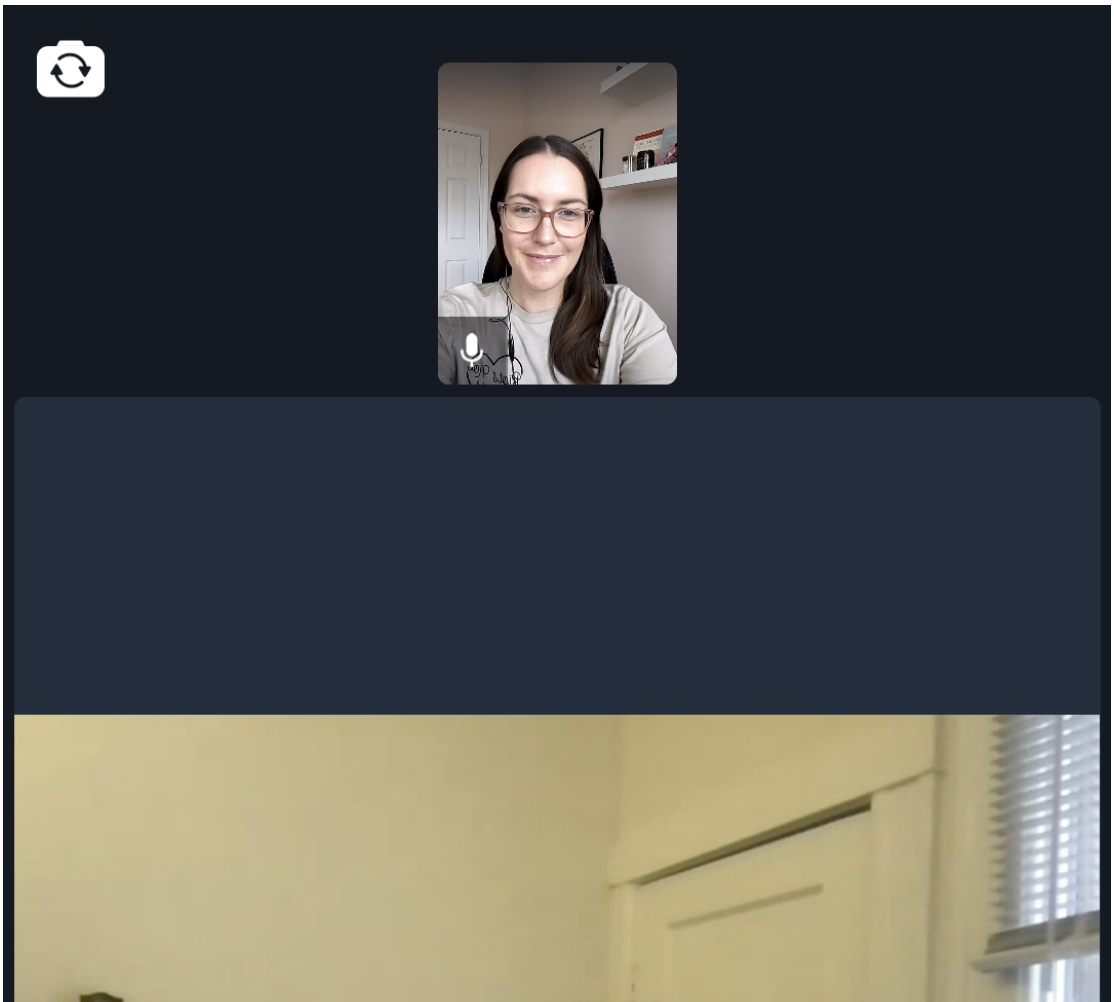
1. Virtualized lists
2. Pagination

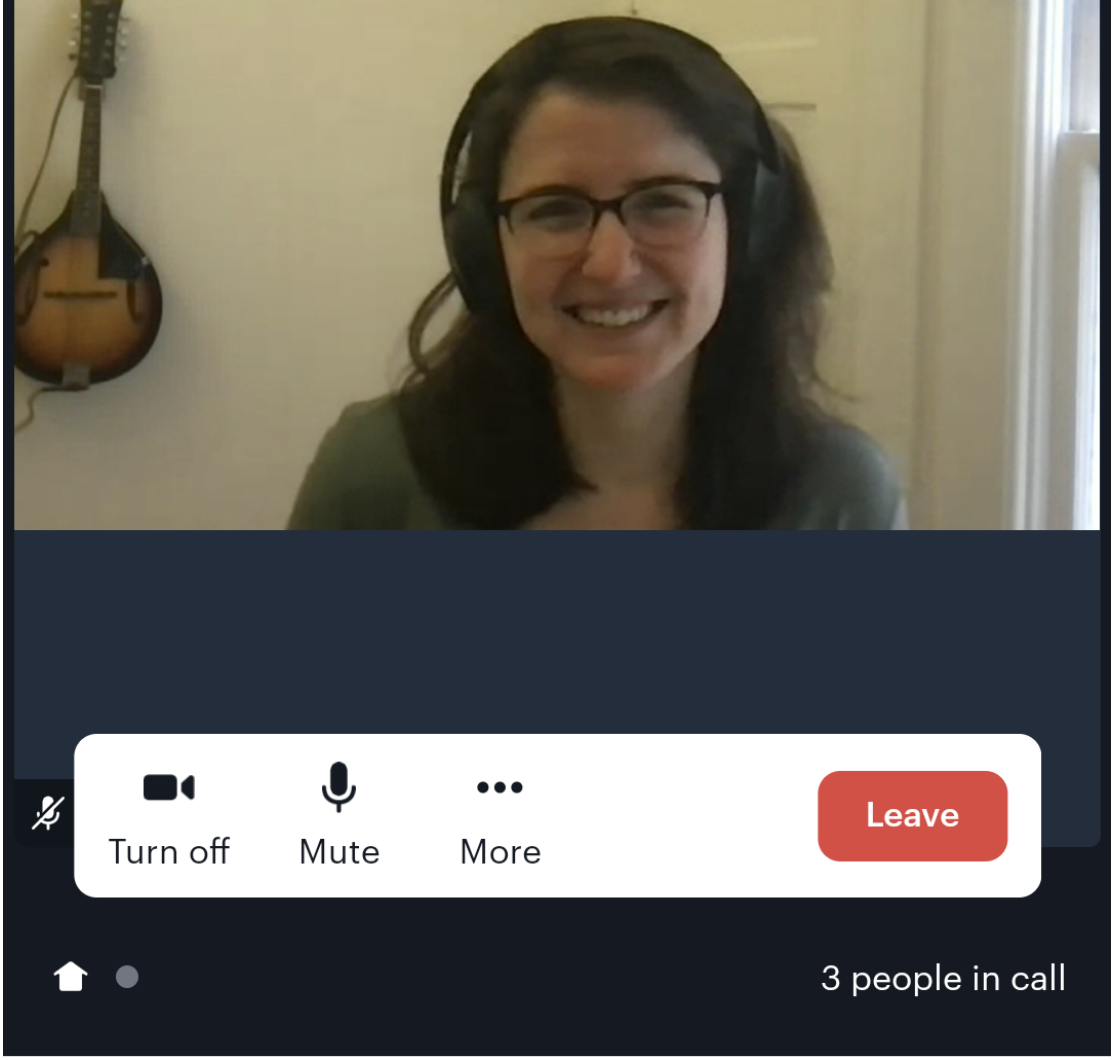
In speaker mode, we used virtualized scrolling, as mentioned, to manage the participant bar videos. In grid mode, however, we use pagination to limit how many videos are on the screen at any given time. This allows all participants to be viewable, just not all at the same time.



Daily Prebuilt grid view with pagination

The number of videos and the grid's tile dimensions ultimately depend on the browser window size and what fits best based on our video aspect ratio requirements.





Paginated calls on mobile. With mobile pagination, you can swipe to see more participants.

In Daily Prebuilt's mobile designs, we're a lot stricter with our grid layout and never render more than three remote participant tiles at a time. This is because mobile devices (especially iOS devices) use a noticeable

amount of CPU resources to decode video. We've found mobile devices often can't handle more than three (or so) videos at a time. Pagination helps manage this CPU bottleneck by allowing users to page through all participants while never rendering more than three remote videos.
To see an example of how pagination can be implemented with a grid of videos in Daily's call object, let's take a look at an example from a React app.

```
return (
    <div ref={gridRef} className="grid">
      {pages > 1 && page > 1 && (
        <button type="button" onClick={handlePrevClick}>
          Back
        </button>
```

```
      )}

      <div className="tiles">{tiles}</div>

      {pages > 1 && page < pages && (
        <button type="button" onClick={handleNextClick}>
          Next
        </button>
      )}
    </div>
  );
```

A React component return value for rendering a paginated grid

In the code block above, we render a parent `div` element. Inside the `div`, there's a `Back` button conditionally rendered if you're not on the first page. (Alternatively, you could render the button and disable it instead.) Next, we render the participant video tiles. Lastly, there's another conditional `Next` button if you're not on the last page.

Now let's take a look at the tiles being rendered:

```
const visibleParticipants = useMemo(() => {
  const participants = callFrame.participants();
  return (
    participants.length - page * pageSize > 0
      ? participants.slice((page - 1) * pageSize, page * pageSiz

      : participants.slice(-pageSize),
    [page, pageSize, callFrame]
  );
});

const tiles = useMemo(
  () => visibleParticipants.map((p) => <Video participant={p} />
  [visibleParticipants]
);
```

An example of how to determine which participants are visible in a grid

Here, we calculate which participants are visible by taking the total number of participants, the page number, and the number of participants per page. With those numbers, we can determine which participants should have tiles rendered for them.

Once we know the visible tiles, we can render a tile for each one. Each time the page number is increased or decreased by clicking the `Next` or `Back` buttons, the visible participants can be recalculated and the tile updates.

By restricting the number of tiles— and, therefore, the number of videos — being rendered at any given time, we can reduce the CPU load of a Daily video call substantially.

5. Memoizing components that re-render too often

You may have noticed in the example above, we're using a React hook called `useMemo` .

`const tiles = useMemo(() => {...}, [dependency1, dependency2]);`

`useMemo` is an example of how to "memoize" React components. Memoization is an effective way to avoid re-computing potentially "expensive" calculations by using the cached computed value until one of the dependencies has changed. (A dependency is a value that affects the rendered output.) Memoization is used here to only update the

`tiles` value when the dependencies— the values in the second parameter, the array— change.
Let's look at another example to see how memoization works. In React, if you have a paragraph element (`<p>`) that displays the sum of two numbers that are each passed as props to a component, you could represent it like so:

```
const displayedSum = useMemo(() => {
    return (
      <p>Total: {num1 + num2}</p>
    )
}, [num1, num2]);
```

Basic example of memoizing a paragraph element that displays the sum of two numbers

We can say pretty confidently that if `num1` and `num2` 's values don't change, the `displayedSum` element won't change. (2+2=4, right?)

By using `useMemo` , we're telling React that it doesn't need to keep re-rendering this element unless `num1` or `num2` change, because then it will actually need to calculate the total again.

In the case of `displayedSum` , adding two numbers is probably not a very "expensive" calculation in terms of CPU usage; however, with a grid of `<video>` elements, re-renders can get expensive fairly quickly, especially on mobile devices.

Preventing expensive re-renders via memoization (or any other methods) is one of the fastest ways to improve performance in your video or audio-only calls. If you've noticed any performance issues in your own Daily app, this is a great place to start.

6. Avoid unnecessarily removing and adding back videos

This one might sound contrary to what we've been saying so far but hear us out.

While it's important to remove `<video>` elements that aren't visible, you should avoid unnecessarily adding or tearing down media (video and audio) elements as much as possible. In React, for example, this could mean making sure your React hook dependencies are not too broad and you are not re-rendering media elements when you don't need to.
This is especially important on iOS, which will have a noticeable CPU hit when adding and removing media elements unnecessarily.

7. Only play paused `<video>` elements

You might be starting to notice a pattern here and, well, you'd be right. If we could sum up our suggestions in one (possibly condescending)

sentence, it would be, “Don’t do anything you don’t need to do.”

This is also the case for playing videos.

```
useEffect(() => {
  const video = videoRef.current;
  if (!video) return;
  if (participant.videoTrack) {
        video.srcObject = new MediaStream([videoTrack]);
  } else {
    video.srcObject = null;
  }

  const handleCanPlay = () => {
    if (!video.paused) return;
    video.play();
  };

  video.addEventListener('canplay', handleCanPlay);
  return () => {
    video.removeEventListener('canplay', handleCanPlay);
  };
}, [videoTrack, videoTrack?.id]);
```

A React `useEffect` hook example of checking if a video element is paused before playing it

In this snippet from Daily Prebuilt mobile code, we set the `srcObject` for the video element (represented by `videoRef`) if there’s a video

track (`videoTrack`) available. Otherwise, the source is set to `null` . We then add an event listener for the `canplay` event. The video element is then played as soon as it’s available if it is not already playing. For example, the video may get paused when disconnecting a Bluetooth audio device, so adding this event listener will help ensure the video is resumed as soon as its media stream is ready again.

You might be wondering if it really matters if you call `play()` on a video that’s not paused. It turns out checking if a video is actually paused before playing it does help performance, especially on iOS Safari.

As we discovered rewriting Daily Prebuilt for mobile, playing a video that

is already playing on iOS Safari is not a "no-op". The action of playing a video, even if it is already playing, takes about 300ms to complete.

This means adding a simple check to see if the video is paused before playing will actually reduce the CPU usage of your Daily call on mobile.

Wrapping up

If there's one thing we appreciate about WebRTC video calls at Daily, it's that getting performance right across browsers and devices is tough. Hopefully, these lessons we've learned along the way help you customize your Daily calls even faster.

To learn more about building a custom Daily video call, check out our React demo repo, as well as our tutorial that goes along with it.

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

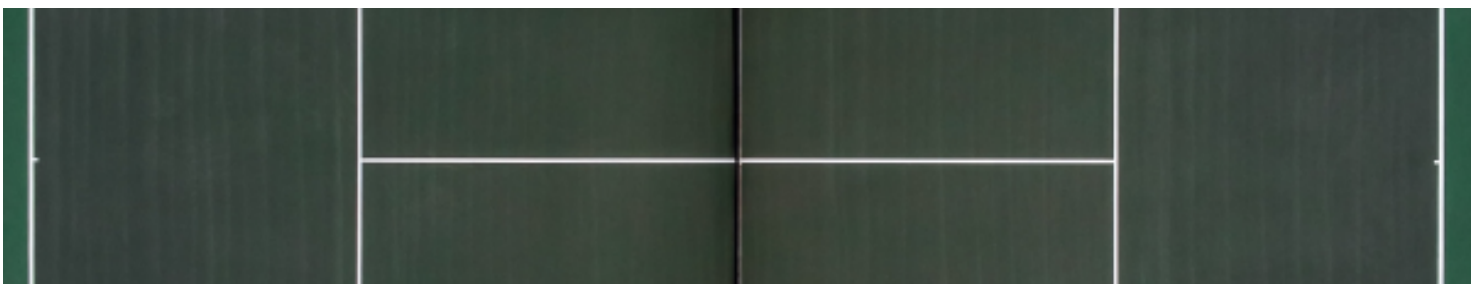
Stream with HLS while recording to a custom S3 bucket using Daily

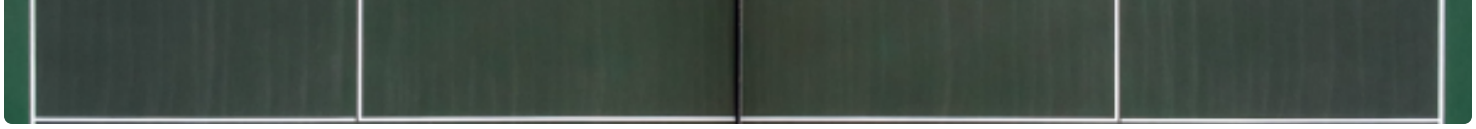
30 Aug 2022 - 5 min read

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 →





Front-End

Implement responsive aspect ratio on window resize with Daily Prebuilt

Use CSS, JavaScript, or React hooks to style your Daily Prebuilt iframe



Jess Mitchell
30 Jul 2021 – 8 min read



API Tutorial

Build a real time video chat app with Next.js and Daily

2022-04-07: The code snippets in this post reference an older version of the examples repo prior to the release of Daily React Hooks. Any code not using Daily React Hooks



Kimberlee Johnson, Daily Team
18 Jun 2021 – 6 min read