CUSTOM DAILY API TUTORIAL CODE & TUTORIALS

Add a "knocking" feature to a custom Daily video call app

Allow guests to knock to enter private calls with Daily's Client SDK for JavaScript



Jess Mitchell
Jul 6, 2023 / 18 min read



Managing privacy settings is a crucial element of any video call app. In some cases, video call hosts don't need to know who is joining; however, it is common to have a specific invite list of

who is allowed to join.

In a previous post, we reviewed Daily's knocking feature for private rooms. Knocking allows video call participants to "knock" to enter the call, which alerts the host of their attendance, and lets the host decide if they can enter. We'll refer to some information from the previous post a few times in this tutorial to avoid repetition, so it's best to read that one first!

Today's agenda

In today's post, we'll look at sample code for Daily's knocking feature and step through how it works. The demo app we've built will specifically showcase knocking, and not some of the other features you'd typically see in a video call, like media controls. We'll recommend some more comprehensive demo apps at the end if you're interested in diving into other aspects of building video calls with Daily.

In terms of functionality, our app will:

- Allow a host (who we'll refer to as an "owner") to join a video call
- Allow a guest to knock to join the call
- Alert the owner of a guest's knocking
- Let the owner respond to the knocking with an "Accept" and "Reject" button
- Notify the guest of the owner's response once it's been made

Beyond that, we've kept this demo simple to avoid the nitty-gritty details of video calls from getting in the way of understanding our knocking feature. Only video is included in our demo app (not audio) to show when a participant has actually entered the call.

The sample code we'll be looking at is written in plain JavaScript, HTML, and CSS. It uses Daily's Client SDK for JavaScript, which allows you to build a custom video call UI. We'll explain the Daily code, but you'll need to already be fairly familiar with JavaScript and HTML to be able to follow along.

Now that we've gotten our requirements out of the way, let's get started!

Running the demo app locally

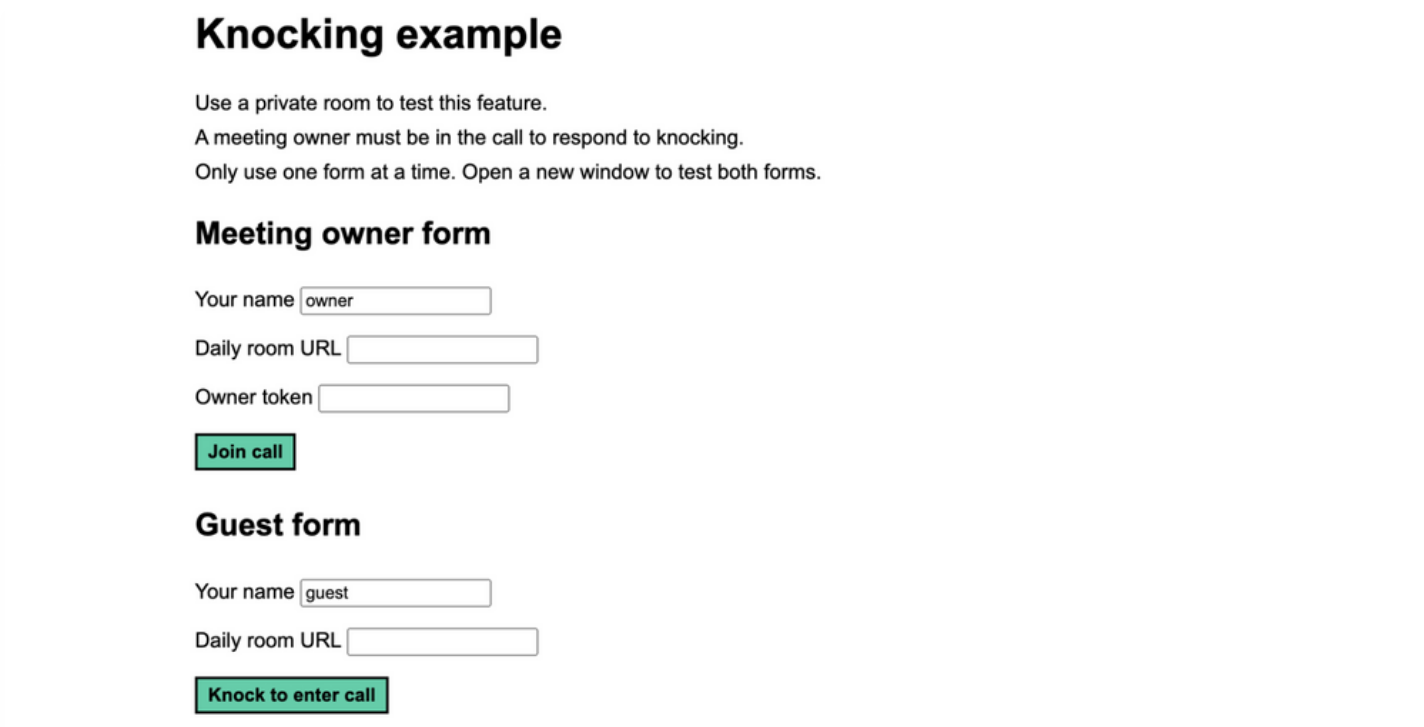
If you prefer to jump to the answer and test out this feature yourself, clone the `daily-samples-js` app and navigate to the `knocking` directory with the following commands in your terminal:

```
git clone https://github.com/daily-demos/daily-samples-js.git
cd daily-samples-js/samples/client-sdk/knocking/
```

You can either open the `index.html` file directly in your browser, or, from the `knocking` directory, run the following commands:

```
npm i
npm run start
```

This will start a local server. In your browser of choice, navigate to `http://localhost:8080/` to see the demo.



Knocking demo app UI

Refer to the README for more information if you hit any issues. It has a thorough explanation of how everything works.

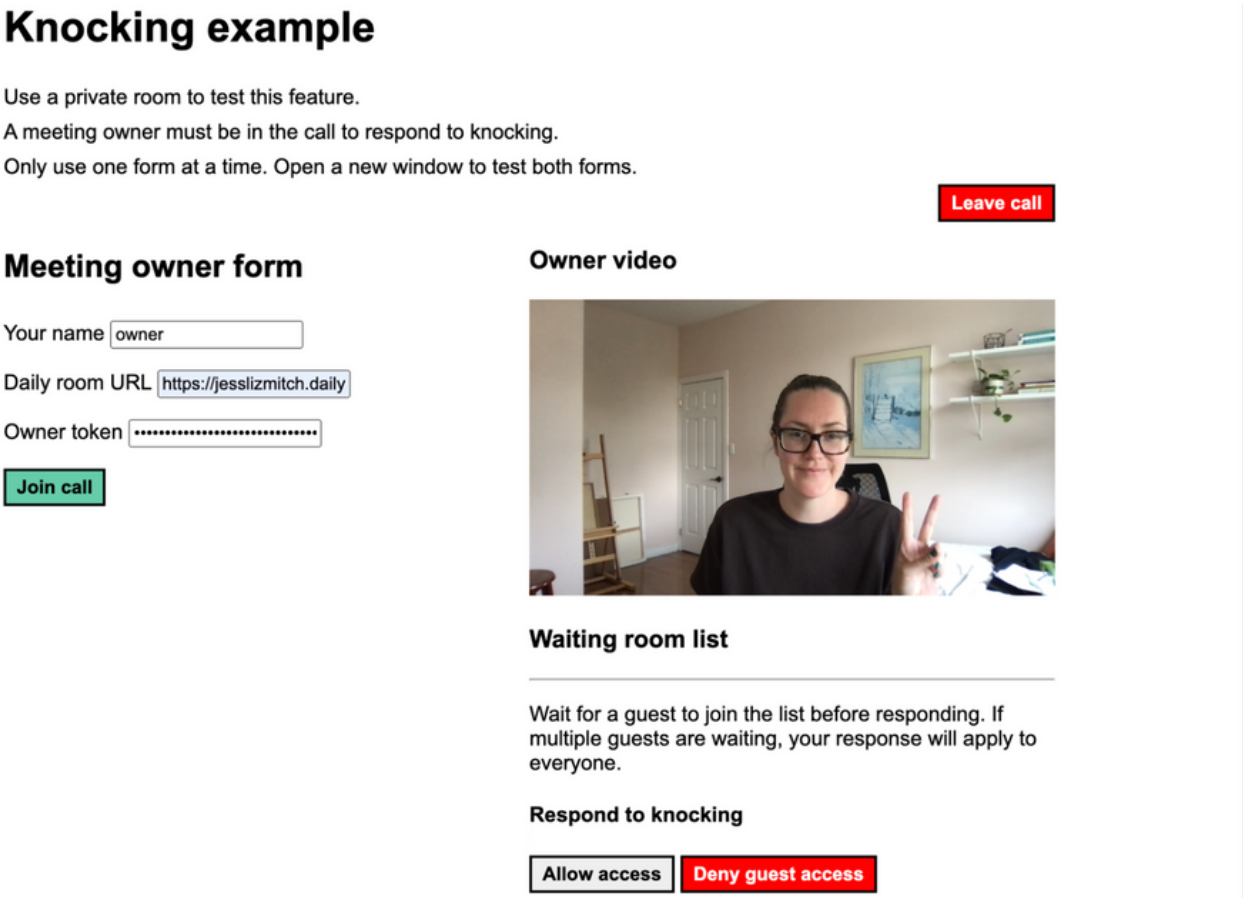
How to use the knocking demo app

This demo requires a couple of pieces of information to interact with it:

1. A private Daily room with knocking enabled. (The README and previous post explain how to create this if you aren't sure).
2. An *owner* meeting token. Only owners can respond to knocking so be sure it's specifically for owners. (Again, the README and previous post include instructions).

You will need to open the demo app in two browser windows to test out the knocking functionality. This will allow you to interact as a call owner in one window and as a guest in the other.

In the tab where you'll be joining as a meeting owner, fill out the "Meeting owner form" with your name, the Daily room URL, and your owner token. Click "Join call" and you should see this view:



Owner view after joining the call

In the image above, we see our local video and the “Waiting room list” below it. When a guest knocks, we’ll see their name pop up in the list. The “Allow access” and “Deny guest access” buttons will work when there is a guest who has knocked and is waiting to enter.

In the tab where you’ll be joining as a guest, enter your name and the Daily room URL. Click the “Knock to enter call” button to let the meeting owner know you’re trying to join the call.

Knocking example

Use a private room to test this feature.
A meeting owner must be in the call to respond to knocking.
Only use one form at a time. Open a new window to test both forms.

Leave call

Owner video

Guest form

Your name guest

Daily room URL https://jesslizmitch.daily

Knock to enter call

Guest video

Your are in the waiting room.

If an owner is in the call, they will be notified you have knocked and are waiting to enter.

Guest view after knocking to enter

Knocking example

Use a private room to test this feature.
A meeting owner must be in the call to respond to knocking.
Only use one form at a time. Open a new window to test both forms.

Leave call

Meeting owner form

Your name owner

Daily room URL https://jesslizmitch.daily

Owner token ••••••••••••••••••••••••••••••

Join call

Owner video



Waiting room list

- guest: 75ba2626-6cef-4e7d-ba35-01780a557c9e

Wait for a guest to join the list before responding. If multiple guests are waiting, your response will apply to everyone.

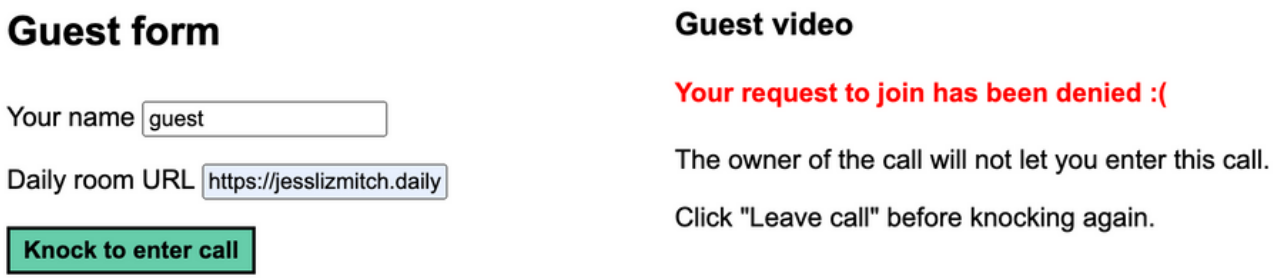
Respond to knocking

Allow access Deny guest access

Owner view after a guest knocked to enter

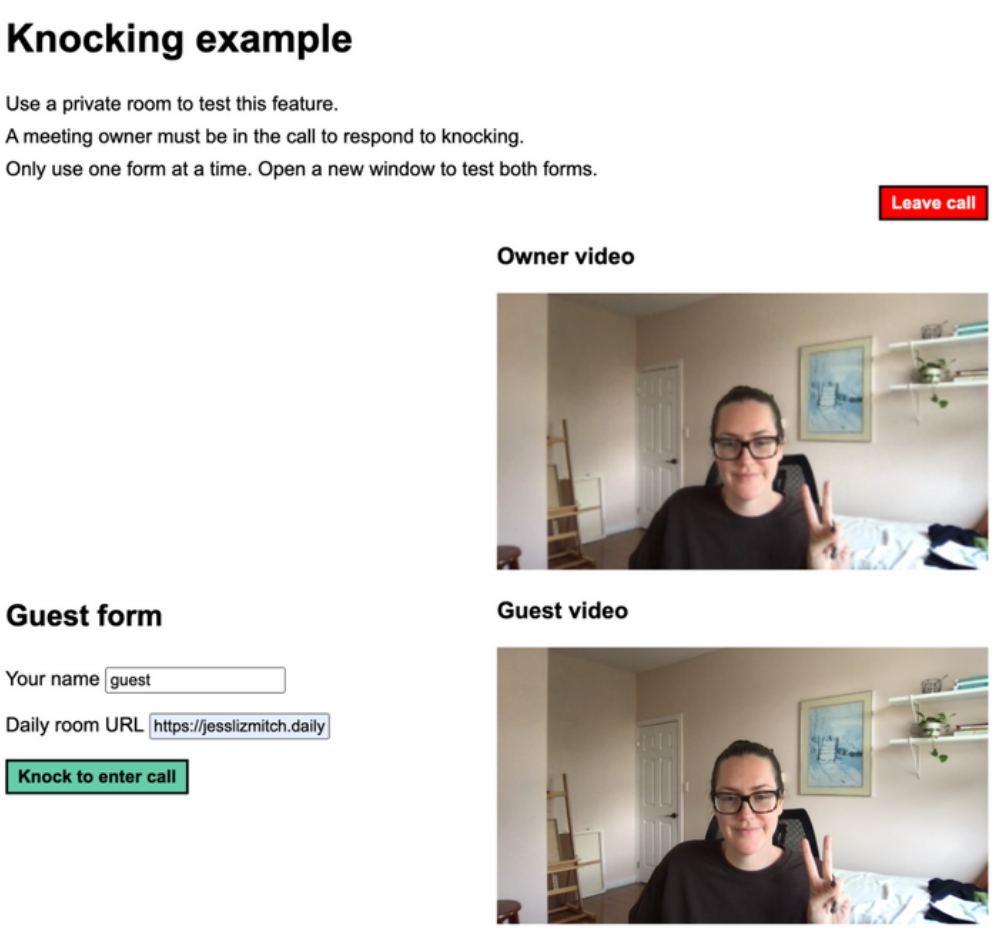
In your *owner* tab, we can see the “Waiting room list” has been updated with our knocking guest’s information. We can decide to either let them in or deny their request.

If the owner rejects the request, the guest will be notified with the following message:



Guest view after being rejected from the call 😕

If the owner accepts the request, the guest will automatically enter the call as soon as the owner’s response registers.



Guest view after being allowed into the private call

(This seems like a good time to defend the author's CSS skills and remind the reader this sample app is just meant to show functionality! 😉)

Now that we know what we're building, let's look at the code.

Coding our knocking feature

Including `daily-js` via a `script` tag

As a first step to building this Daily video call app, we need to include `daily-js`, which is the implementation of Daily's Client SDK for JavaScript. In `index.html`, include the script in the HTML `<head>` element:

```
<script crossorigin src="https://unpkg.com/@daily-co/daily-js"></script>
```

Important Daily methods and events related to our knocking feature

Before looking at the demo app's JavaScript code, let's familiarize ourselves with the Daily methods and events that will be used. To help understand how to use each method and event, they're ordered in terms of how they relate to the owner and guest's separate UI interactions:

1. The owner submits the "Meeting owner form" to join a video call:

- `createCallObject()` creates our call object instance. It is only used for custom Daily video apps – not Daily Prebuilt apps. This is how you – the developer – will interact with Daily's API and update your video call settings.
- `join()` is a call object instance method that allows the participant to join the call. In this case, since it's a private call and the owner is joining with a valid meeting token, "joining" will let the participant enter the call directly. For those who are not allowed to enter (e.g., a guest who needs to knock via the "Guest form"), calling `join()` allows the participant to get basic information about the call, but it does not let them enter the call yet. (Think of an apartment building: "joining" for guests is like entering the building but they still need to buzz or knock to be let in past the lobby.)

2. In a separate browser window, the guest submits the "Guest form" to knock to join the call. Note: This will let them "join" but not enter the actual call. It essentially adds them to the waiting room for the call:

- `createCallObject()` : The call object instance is created for the guest.
- `preAuth()` : Pre-authenticate the participant in the room. This allows us to get information related to the room, such as what the "access state level" for the participant will be when they join. In other words, it allows us to check if the participant will need to knock with the next method.
- `accessState()` : Check the access state of the call participant. If they're in the "lobby" (i.e., they need permission to join), they'll need to knock. We check this value to make sure we're not making the guest knock when they don't need to. This could happen if they use a public room instead of a private room in this demo, in which case they would be able to join directly without knocking.
- `join()` : The participant joins the call via this instance method. They will still be in the lobby/waiting room, since they still need the call owner's permission to enter the call.

- `requestAccess()` : This instance method allows the guest to actually knock. Calling this instance method will add them to the call's waiting room list.

3. The owner is alerted of a guest's knocking:

- `'waiting-participant-added'` : Listening to this Daily event allows the owner to know any time the list of guests in the waiting room changes.

4. The owner responds to the guest's knocking by clicking either the "Allow" or "Deny" button:

- `updateWaitingParticipant()` : This method will report if the guest's request to enter the call was granted or not based on the owner's decision to allow or deny it.
- `'waiting-participant-removed'` : This event lets us – the developer – know when the guest is officially removed from the list, which allows us to update the app UI for the owner. In other words, the owner will no longer see in the app's UI that the guest is waiting to enter after they've rejected or accepted the request.

5. The guest is notified of the owner's response once their decision is received:

- `'access-state-updated'` : On the guest's end, we can listen for this Daily event to know when the guest has been accepted to the call.
- `'error'` : Similarly, on the guest's end, we listen for an error event to know if the guest has been rejected. There will be a specific error message that lets us know the error is a rejection. (More on that below.)

Keep reading to see how each of these events/methods are implemented in our demo code. Any additional events/methods in the codebase are related to updating the app UI or cleaning up the app state, but they're not our focus for this feature.

Creating the owner form and initiating the Daily call on submit

In `index.html` , we have a form for the owner to fill out before joining the call:

```
<form id="ownerForm">
    <label for="ownerName">Your name</label>
```

```
    <input type="text" id="ownerName" value="owner" required />
    <br />
    <label for="ownerURL">Daily room URL</label>
    <input type="text" id="ownerURL" required />
    <br />
    <label for="token">Owner token</label>
    <input type="password" id="token" required />
    <br />
    <input type="submit" value="Join call" />
</form>
```

There is no `onsubmit` event directly on the form element because we'll add it programmatically in the accompanying JavaScript file.

Now, let's look at that file, `index.js`.

First, we add a handler for our `"submit"` event, which attaches the `submitOwnerForm()` function to the event:

```
const ownerForm = document.getElementById('ownerForm');
ownerForm.addEventListener('submit', submitOwnerForm);
```

Let's look at `submitOwnerForm()`:

```
const submitOwnerForm = (e) => {
  e.preventDefault();
  // Do not try to create new call object if it already exists
  if (callObject) return;
  // Get form values
  const { target } = e;
  const name = target.ownerName.value;
  const url = target.ownerURL.value;
  const token = target.token.value;


  // Log error if any form input is empty
  if (!name.trim() || !url.trim() || !token.trim()) {
    console.error('Fill out form');
```

```
    return;
  }
  // Initialize the call object and let the owner join/enter the call
  createOwnerCall({ name, url, token });
};
```

In `submitOwnerForm()`, the following happens:

1. The form is prevented from reloading the page with `e.preventDefault()`.
2. We get the form values that we need for the video call (the user's name, the Daily room URL, and the meeting token they submitted).
3. The form values are cleaned up by trimming whitespace and adding some error handling.
4. `createOwnerCall()` is called, which is where we start interacting with Daily's Client SDK for JavaScript.

In `createOwnerCall()`, we first need to create an instance of Daily's call object – assigned to `callObject` in our code here. As mentioned before, the call object is how we will interact with Daily's APIs. It manages everything related to our Daily video call.

```
const createOwnerCall = ({ name, url, token }) => {
  showLoadingText('owner');

  // Create call object
  callObject = window.DailyIframe.createCallObject();

  // Add Daily event listeners (not an exhaustive list)
  // See: https://docs.daily.co/reference/daily-js/events
  addOwnerEvents();

  // Let owner join the meeting
  callObject.join({ userName: name, url, token }).catch((error) => {
    console.log('Owner join failed: ', error);
    hideLoadingText('owner');
```

```
  });
};
```

Let's step through the above:

1. We show our "Loading" text in the UI while we're setting up the call.
2. We assign the `callObject` variable declared at the top of the file to our new instance of the Daily call object. Note: There can only be one simultaneous instance of a call object.
   `callObject = await window.DailyIframe.createCallObject();`
3. We attach Daily event listeners with `addOwnerEvents()` . (More on this below.)
4. We join the call. Since this is an owner with a meeting token, they can enter the call directly by calling the `join()` call object instance method.
   `const join = await callObject.join({ userName: name, url, token });`

In `addOwnerEvents()` , event listeners for various Daily **events** get added. Daily emits events that can be handled as a way to alert an app to update its state. The events you'll consume depend on your app and what information you're trying to display to your users.

```
cconst addOwnerEvents = () => {
  callObject
    .on('joined-meeting', handleOwnerJoinedMeeting)
    .on('left-meeting', handleLeftMeeting)
    .on('participant-joined', logEvent)
    .on('participant-updated', handleParticipantUpdate)
    .on('participant-left', handleParticipantLeft)
    .on('waiting-participant-added', addWaitingParticipant)
    .on('waiting-participant-updated', logEvent)
    .on('waiting-participant-removed', updateWaitingParticipant)
    .on('error', logEvent);
};
```

`handleJoinedMeeting()` is the most important function to be aware of at this point. Since Daily's `join()` event returns a **Promise**, the `'joined-meeting'` event will be triggered when the `join()` Promise resolves, and the local participant has officially joined.

```
const handleOwnerJoinedMeeting = (e) => {
  logEvent(e);
  const participant = e?.participants?.local;
  if (!participant) return;


  if (participant.owner) {
    console.log('This participant is a meeting owner! :)');
  } else {
    // this means they used a non-owner token in the owner form
    console.error('This participant is not a meeting owner!');
  }


  // Update UI
  hideLoadingText('owner');
  hideForms('guest');
  showOwnerPanel();
  showVideos();
  showLeaveButton();


  // This demo assumes videos are on when the call starts since there aren't
media controls in the UI.
  if (!participant?.tracks?.video) {
    // Update the room's settings to enable cameras by default.
    // https://docs.daily.co/reference/rest-api/rooms/config#start_video_off
    console.error(
      'Video is off. Ensure "start_video_off" setting is false for your room'
    );
    return;
  }
  // Add video tile for owner's local video
  addParticipantVideo(participant);
};
```

Once the local participant (in this case, the owner) has joined, we can update our UI and add their video to the DOM. Adding the video is handled by `addParticipantVideo()` as follows:

```
const addParticipantVideo = (participant) => {
  if (!participant) return;
  // If the participant is an owner, we'll put them up top; otherwise, in the
guest container
  const videoContainer = document.getElementById(
    participant.owner ? 'ownerVideo' : 'guestVideo'
  );


  let vid = findVideoForParticipant(participant.session_id);
  // Only add the video if it's not already in the UI
  if (!vid && participant.video) {
    // Create video element, set attributes
    vid = document.createElement('video');
    vid.session_id = participant.session_id;
    vid.style.width = '100%';
    vid.autoplay = true;
    vid.muted = true;
    vid.playsInline = true;
    // Append video to container (either guest or owner section)
    videoContainer.appendChild(vid);
    // Set video track
    vid.srcObject = new MediaStream([participant.tracks.video.persistentTrack]);
  }
};
```

Essentially, we're making sure the participant doesn't already have a video in the DOM and, if not, we create a `<video>` element, assign the relevant attributes, and attach it to the video container – a `<div>` element.

Note: The owner video is displayed on top of the screen and the guest video on the bottom, so we use a different `<div>` element for `videoContainer` depending on who is joining.

```
const videoContainer = document.getElementById(
  participant.owner ? 'ownerVideo' : 'guestVideo'
);
```

At this point, our owner has joined the call, can see their own video in the UI, and can see the waiting list.

Allow guests to join the waiting room and knock to enter to call

Next, let's build out the guest's perspective of using this app. We need to provide a form where they can knock to join the same room as the owner.

```
<form id="knockingForm">
  <label for="guestName">Your name</label>
  <input type="text" id="guestName" value="guest" required />
  <br />
  <label for="guestURL">Daily room URL</label>
  <input type="text" id="guestURL" />
  <br />
  <input type="submit" value="Knock to enter call" required />
</form>
```

The form looks a lot like the owner's form, but there's no token input since guests don't have a token. (If they did, they wouldn't need to knock!)

Just like before, we need to programmatically add an event listener for submitting this form, which we add in `index.js`:

```
const knockingForm = document.getElementById('knockingForm');
knockingForm.addEventListener('submit', submitKnockingForm);
```

The function `submitKnockingForm()` is attached to the `'submit'` event. It acts similarly to the owner's form, but with some key differences:

```
const submitKnockingForm = (e) => {
  e.preventDefault();


  // Do not try to create new call object if it already exists
  if (callObject) return;
```

```
  // Get form values
  const { target } = e;
  const name = target.guestName.value;
  const url = target.guestURL.value;


  // Log error if either form input is empty
  if (!name.trim() || !url.trim()) {
    console.error('Fill out form');
    return;
  }


  // If the user is trying to join after a failed attempt, hide the previous
error message
  hideRejectedFromCallText();


  // Initialize guest call so they can knock to enter
  createGuestCall({ name, url });
};
```

Above, we:

1. Prevent the form submission from reloading the page with `e.preventDefault()`
2. Get the form input values (the participant's name and room URL)
3. Make sure the inputs actually have values by confirming they're "truthy" after trimming any whitespace
4. Hide any previous error messages the user may have encountered with `hideRejectedFromCallText()`
5. Call `createGuestCall()`

Let's go straight to `createGuestCall()` :

```
const createGuestCall = async ({ name, url }) => {
  showLoadingText('guest');
```

```
  // Create call object
  callObject = window.DailyIframe.createCallObject();


  // Add Daily event listeners (not an exhaustive list)
  // See: https://docs.daily.co/reference/daily-js/events
  addGuestEvents();


  try {
    // Pre-authenticate the guest so we can confirm they need to knock before
calling join() method
    await callObject.preAuth({ userName: name, url });


    // Confirm that the guest actually needs to knock
    const permissions = checkAccessLevel();
    console.log('access level: ', permissions);


    // If they're in the lobby, they need to knock
    if (permissions === 'lobby') {
      // Guests must call .join() before they can knock to enter the call
      await callObject.join();
    } else if (permissions === 'full') {
      // If the guest can join the call, it's probably not a private room.
      …
    } else {
      console.error('Something went wrong while joining.');
    }
  } catch (error) {
    console.log('Guest knocking failed: ', error);
  }
};
```

There are several steps that occur here, including:

1. Show loading text while everything gets set up.

2. Create a new call object instance for the guest's call. (Reminder: This is in a different browser window than the owner, so the call object instance doesn't exist for the guest yet.)
3. Add Daily event listeners for the events related to our guest's experience. (More on that below.)
4. Call `preAuth()` with the form name and Daily room URL. This allows us to get the `accessState` of the room before trying to `join()`. In other words, we can find out if this specific participant is actually going to be put in the waiting room. If you used a public room with this demo, for example, there would be no waiting room to join.
5. Call `checkAccessLevel()`, which just calls Daily's `accessState()` method (i.e., returns that `accessState` value we just mentioned.)
6. Next, once we know the permissions are what we expect, we call `join()` to enter the waiting room (a.k.a., `'lobby'`).

There are quite a few steps here, but it's important to note that the order of Daily methods used does matter. We technically still haven't knocked yet, but we're waiting until we've officially "joined" the waiting room to knock. That will happen next, after the `'joined-meeting'` Daily event is emitted.

In terms of the Daily events we've attached, the important ones are a little different than for the owner, as shown in `addGuestEvents()`:

```
const addGuestEvents = () => {
  callObject
    .on('joined-meeting', handleGuestJoined)
    .on('left-meeting', logEvent)
    .on('participant-joined', logEvent)
    .on('participant-updated', handleParticipantUpdate)
    .on('participant-left', handleParticipantLeft)
    .on('error', handleRejection)
    .on('access-state-updated', handleAccessStateUpdate);
};
```

The events relevant specifically to the knocking guest are:

- `'joined-meeting'`, which tells us when the local participant has officially joined the call. (Reminder: In private rooms, this is just the waiting room for guests.)
- `'access-state-updated'`, which will tell us when a guest is accepted into a call after knocking.
- `'error'`, which will tell us when a participant's knocking is rejected. Note: This error event is used instead of `"access-state-update"` because it explicitly states the guest was rejected. This is explained more in the `requestAccess()` docs.

Since we've already called `join()`, the `'joined-meeting'` event should be emitted next. This invokes the `handleGuestJoined()` callback:

```
const handleGuestJoined = (e) => {
  logEvent(e);
  // Update UI to show they're now in the waiting room
  hideLoadingText('guest');
  hideForms('owner');
  showVideos();
  showWaitingRoomText();
  showLeaveButton();

  // Request full access to the call (i.e. knock to enter)
  callObject.requestAccess({ name: e?.participants?.local?.user_name });
};
```

We update the app UI to let the guest know they're now in the waiting room. From the waiting room, we call `requestAccess()`, which is the method for actually knocking. We have now officially knocked! 🎉



(We'll go through more information on the other events mentioned below.)
Tell the owner someone is knocking

Once the guest knocks, we need a way to tell the owner there's someone trying to join. If you recall, we added an event listener for the `'waiting-participant-added'` Daily event. This lets us know when someone new gets added to the list of people knocking to enter:

```
const addOwnerEvents = () => {
 callObject
   // ...
   .on('waiting-participant-added', addWaitingParticipant)
   // ...
```

In `addWaitingParticipant()`, we take the waiting participant's information (their name and ID), and display it in the DOM for the owner to see who's knocking:

```
const addWaitingParticipant = (e) => {
  const list = document.getElementById('knockingList');
  const li = document.createElement('li');
  li.setAttribute('id', e.participant.id);
  li.innerHTML = `${e.participant.name}: ${e.participant.id}`;
  // Add new list item to ul element for owner to see
  list.appendChild(li);
};
```

Next, the owner needs to respond to the request.

Let the owner accept or deny the guest's request to join

Now that the owner sees the knocking participant in the DOM, they can decide to let anyone from the waiting room list in or not.

There are two buttons to accept or deny a knocking participant's request already in the DOM for the owner. (In a production app, you probably only want to show them when someone's knocking, but we always show these buttons to keep things simple.)

```
<button id="allowAccessButton">Allow access</button>
<button id="denyAccessButton">Deny guest access</button>
```

Just like before, we need to add event listeners for the above DOM elements. Since these are `<buttons>` s, we'll add `'click'` event listeners.

```
const allowAccessButton = document.getElementById('allowAccessButton');
allowAccessButton.addEventListener('click', allowAccess);


const denyAccessButton = document.getElementById('denyAccessButton');
denyAccessButton.addEventListener('click', denyAccess);
```

Allowing or denying a participant is done using the `updateWaitingParticipant()` call object instance method. The object passed to `updateWaitingParticipant()` should include a

`grantRequestedAccess` value of either `true` or `false`. (`true` meaning they can enter, and `false` meaning they're denied.)

Let's look at `allowAccess()`:

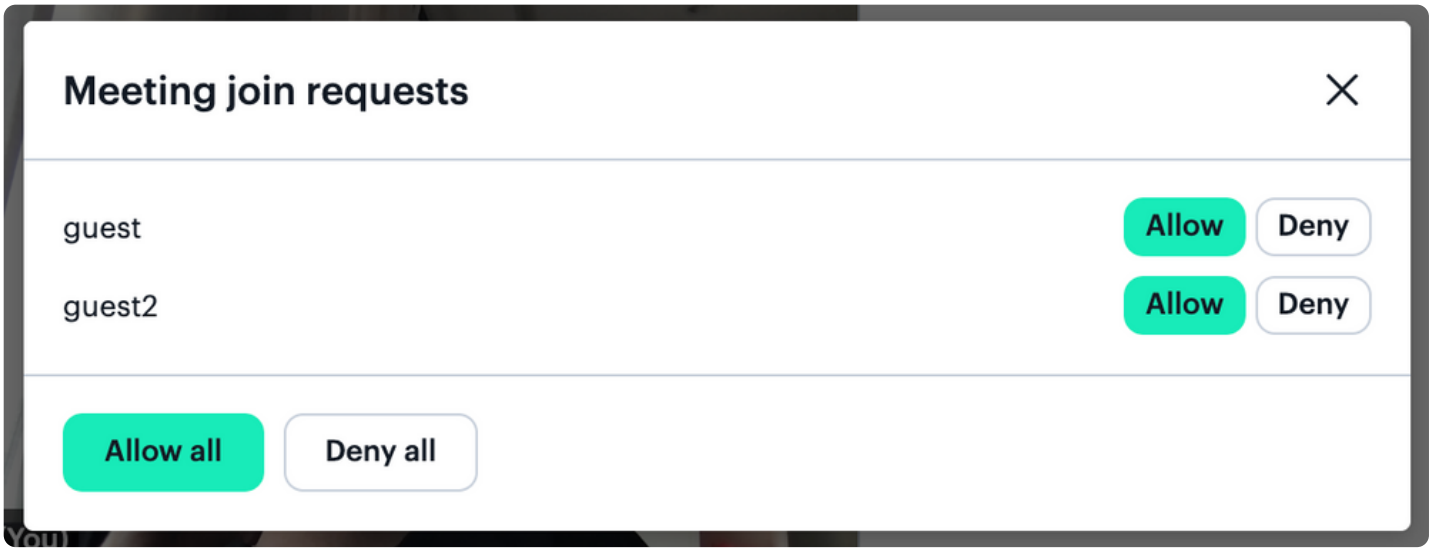
```
const allowAccess = () => {
  console.log('allow guest in');
  // Retrieve list of waiting participants
  const waiting = callObject.waitingParticipants();


  const waitList = Object.keys(waiting);


  // there is also a updateWaitingParticipants() method to accomplish    this in
one step
  waitList.forEach((id) => {
    callObject.updateWaitingParticipant(id, {
      grantRequestedAccess: true,
    });
  });
};
```

In this app, the "Allow" button acts as an "Allow all" button. Again, we're keeping things simple in this demo, but you could have buttons in your app to allow individual guests in or one to accept all guests who are waiting like we have.

In Daily Prebuilt, for example, there's a whole panel to individually allow or deny each waiting participant, or the option to allow all or deny all.



Daily Prebuilt's waiting room list

`updateWaitingParticipant()` can be used to respond to individual requests, or you can also use `updateWaitingParticipants()` to respond to every request at once. Which you use will depend on your UI.

`denyAccess()` looks almost identical, but `grantRequestedAccess` is set to `false`.

Give the guest the good (or bad) news about joining

After the owner provides a response, we can let the guest know and update the UI as needed.

If the owner accepts the knocker's request, we'll get an `'access-state-updated'` Daily event, which alerts us that the guest is no longer in the lobby. The `handleAccessStateUpdate()` method was attached to this event earlier, so let's see what it does:

```
const handleAccessStateUpdate = (e) => {
  // If the access level has changed to full, the knocking participant has been
let in.
  if (e.access.level === 'full') {
    // Add the participant's video (it will only be added if it doesn't already
exist)
    const { local } = callObject.participants();
    addParticipantVideo(local);
    // Update messaging in UI
    hideWaitingRoomText();
  } else {
```

```
    logEvent(e);
  }
};
```

Here, we check that the access level is now `'full'`, which means they have officially entered the call. As soon as that's confirmed, we can add their video with `addParticipantVideo()`, just like we did for the owner.

If they were rejected (😭) we'll let them know in the UI by hiding the waiting room message and showing a rejected message instead via `handleRejection()`:

```
const handleRejection = (e) => {
  logEvent(e);
  // The request to join (knocking) was rejected :(
  if (e.errorMsg === 'Join request rejected') {
    // Update UI so the guest knows their request was denied
    hideWaitingRoomText();
    showRejectedFromCallText();
  }
};
```

The rejected message is already in our HTML file, so we can just toggle a class to actually display it via `showRejectedFromCallText()`:

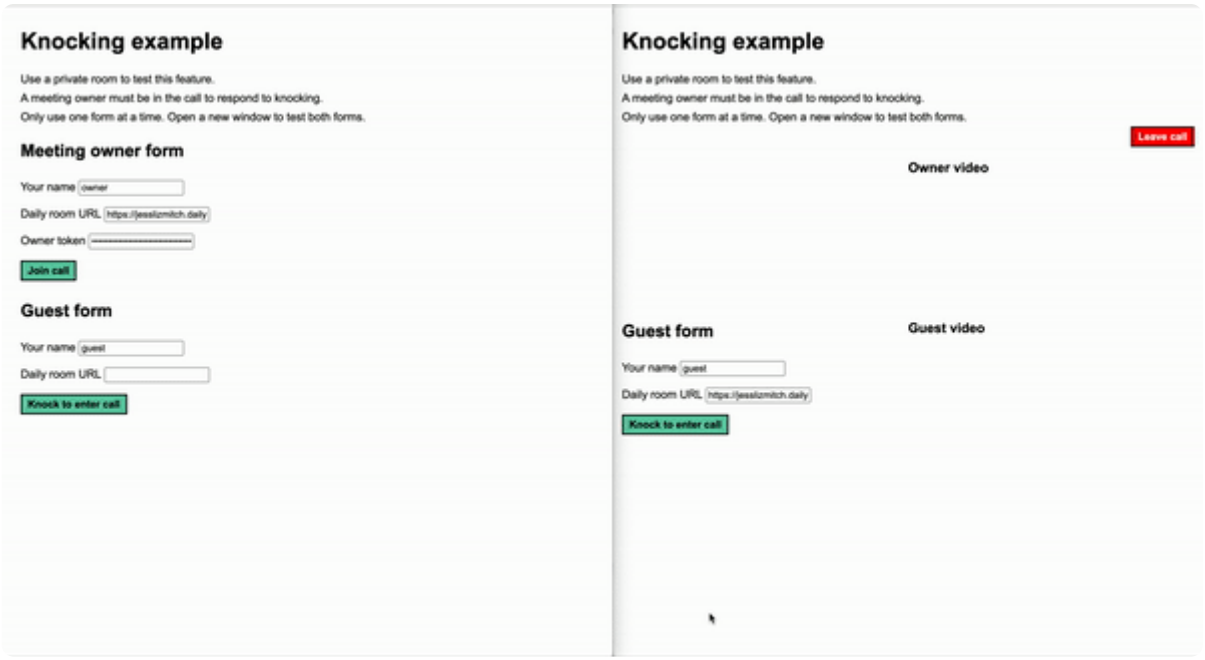
```
const showRejectedFromCallText = () => {
  // Show message a knocking request was denied
  const guestDenied = document.getElementById('guestDenied');
  guestDenied.classList.remove('hide');
};
```

Once this message is shown, we've officially covered all our (basic) use cases for this feature. There is also a "Leave" button to exit the call and some functionality around adding other participants' videos to the local participant's view, but we'll save that for another time.

Conclusion: Putting all these pieces together

In terms of coding this knocking feature, this tutorial has covered the most important aspects that developers working with Daily need to know. Let's take a look at how all these piece come together now:

Both owners and guests can join a Daily room, guests can knock to enter, and owners can respond to the guest's knocking. Mission accomplished!



Owner and guest joining the knocking example app

As a reminder, this code should be treated as a demonstration of how to use the Daily Client SDK for JavaScript to build a knocking feature – *not* how to build a full video call app.

If you're looking for more information on building a video call application with Daily, check out these additional resources:

- Custom video app in React, and its tutorial series
- Spatialization app, and its tutorial series

And, as always, let us know if you have any questions. ✨

Never miss a story

Get the latest direct to your inbox.

Email*

Country*

Join peerConnection - our video/WebRTC developer community.

By submitting, you consent to allow Daily.co to store and process the personal data from this form to fulfill your request.

Subscribe



daily

Interactive Live Streaming for Dummies: Daily.co Special Edition

Get your free copy

Startup program

Use Cases

Telehealth

Live Commerce

EdTech

Event Tech

Creator Technology

Virtual Collaboration

Social and Gaming

Read case studies

Dev Resources

Blog

Guides

Demos

Support

API status

Changelog

Contact us

Contact Us

Live chat

Support

Japan

Resources

Blog

Resource center

Events

Twillio alternative

Vonage alternative

Agora alternative

Videosaurus

Site index

Security & Privacy

Security center

Privacy policy

Terms of service

Cookie policy