



Level up your Twilio API skills in **TwilioQuest**, an educational game for Mac, Windows, and Linux.

Download
Now

BLOG

[DOCS](#) [CONSOLE](#) [TWILIO](#)

Build the future of
communications.

START BUILDING FOR FREE



BY MIGUEL GRINBERG • 2020-04-30

TWITTER

FACEBOOK

LINKEDIN

Build a Video Chat Application with Python, JavaScript and Twilio Programmable Video

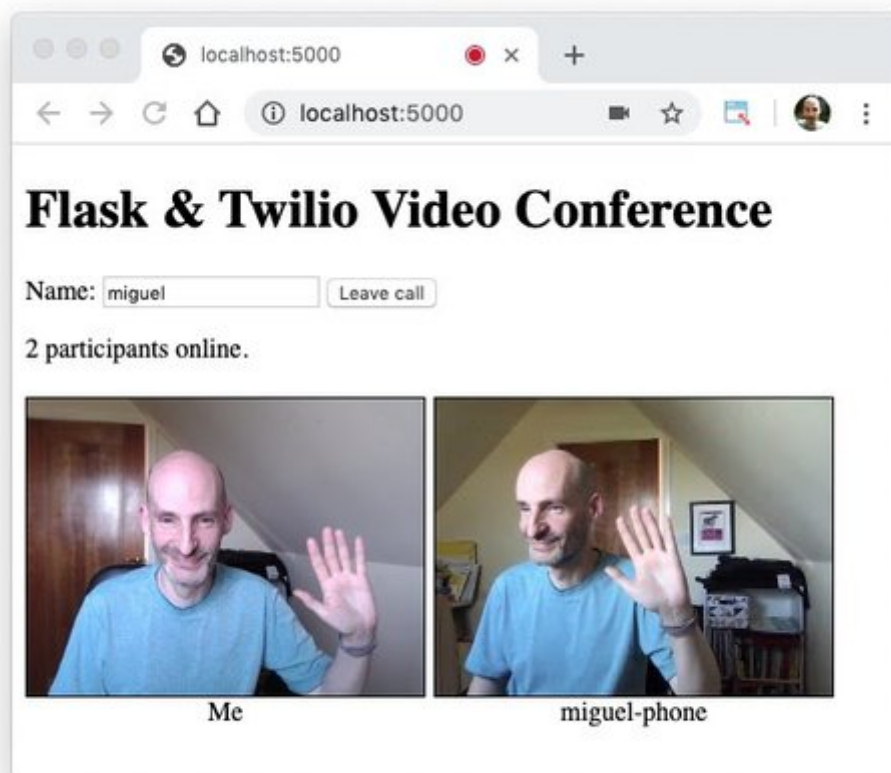
Build a Video Chat Application
with Python, JavaScript and
Twilio Programmable Video

The efforts to limit the spread of COVID-19 around the world have forced a large number of people to work from home and this has naturally created a surge in interest for communication and collaboration tools.

In this article we are going to look at a video conferencing solution, but instead of turning to a third-party system we are going to take the do-it-yourself approach and build our own. Our system is going to run on modern desktop and mobile web browsers, so participants will not need to download or install any software on their computers. The server-side portion of the project is going to use Python and the [Flask](#) framework, and the client-side is going to be built in vanilla JavaScript, with some HTML and CSS sprinkled in the mix for good measure.

If you are worried that this is going to be a long, difficult and obscure tutorial let me set your mind at rest. The magic that will allow us to build this project comes from the [Twilio Programmable Video](#) service, which does the heavy lifting.

Below you can see a test call in which I was connected with my laptop and my mobile phone.



This article goes over the implementation of the project in detail, so that you can follow along and build it on your computer. If you are interested in downloading the complete project

instead of building it step-by-step, you can find it in this GitHub repository:

<https://github.com/miguelgrinberg/flask-twilio-video>.

Tutorial requirements

To build the project you will need:

- Python 3.6 or newer. If your operating system does not provide a Python interpreter, you can go to python.org to download an installer.
- [ngrok](#). We will use this handy utility to connect the Flask application running on your system to a public URL that Twilio can connect to. This is necessary for the development version of the application because your computer is likely behind a router or firewall, so it isn't directly reachable on the Internet. If you don't have ngrok installed, you can [download a copy for Windows, MacOS or Linux](#).
- A free or paid Twilio account. If you are new to Twilio [get your free account](#) now! This link will give you \$10 when you upgrade.
- A web browser that is compatible with the Twilio Programmable Video JavaScript library (see below for a list of them). Note that this requirement also applies to the users you intend to invite to use this application once built.

Supported web browsers

Since the core video and audio functionality of this project is provided by Twilio Programmable Video, you have to use a web browser that is supported by this service. Here is the current list of supported browsers:

- Android: Chrome and Firefox
- iOS: Safari
- Linux: Chrome and Firefox
- MacOS: Chrome, Firefox, Safari and Edge
- Windows: Chrome, Firefox and Edge

Check the [Programmable Video documentation](#) for the latest supported web browser list.

Project structure

Let's begin by creating the directory where we will store our project files. Open a terminal window, find a suitable parent directory and then enter the following commands:

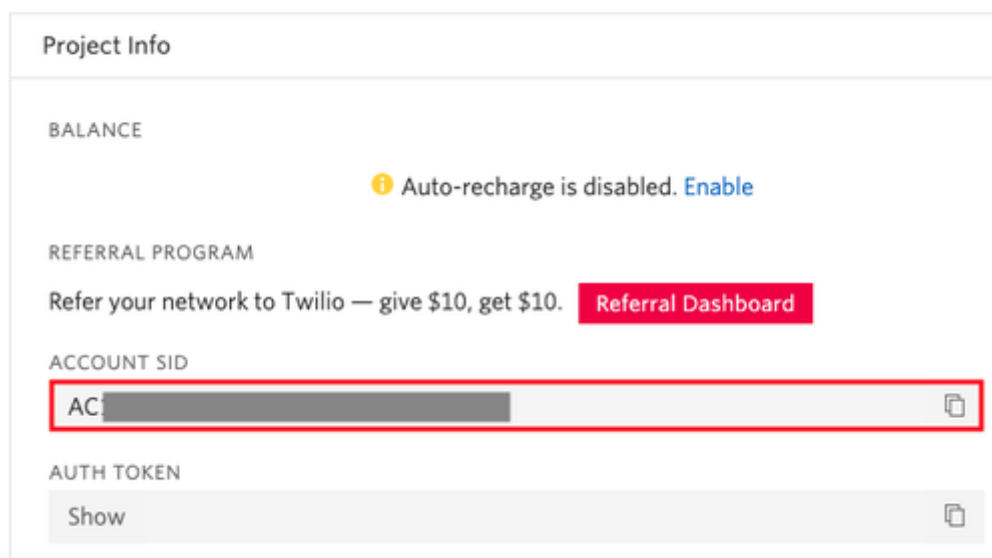
```
1 $ mkdir flask-twilio-video
2 $ cd flask-twilio-video
```

Following the most basic Flask application structure, we'll now create two sub-directories, *static* and *templates* to store the files that will be served to the client.

```
1 $ mkdir static
2 $ mkdir templates
```

Setting up your Twilio account

Log in to your Twilio account to access the [Console](#). In this page you can see the "Account SID" assigned to your account. This is important, as it identifies your account and is used for authenticating requests to the Twilio API.



Because we are going to need the Account SID later, click the “Copy to Clipboard” button on the right side. Then open a new file named `.env` in your text editor (note the leading dot) and write the following contents to it, carefully pasting the SID where indicated:

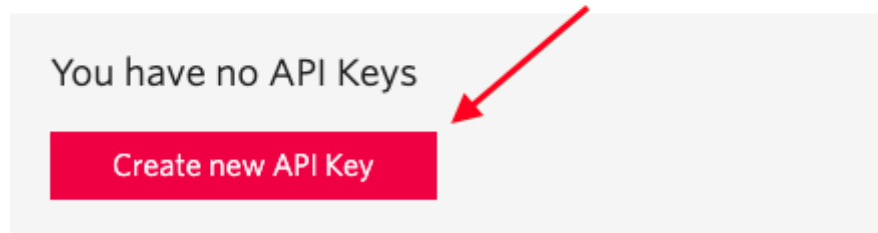
```
1 TWILIO_ACCOUNT_SID=<your-twilio-account-sid>
```

The Programmable Video service also requires a Twilio API Key for authentication, so in this step you will add one to your Twilio account. To begin, navigate to the [API Keys](#) section of the Twilio Console.

If you’ve never created an API Key before, you will see a “Create new API Key” button. If you already have one or more API Keys created, you will instead see a red “+” button to add one more. Either way, click to create a new API Key.

API Keys

API Keys are revokable credentials for the Twilio API. You can use Tokens, which are used by Twilio's Real-Time Communications SD



Enter **videochat** as the name of the key (or any name you like), leave the key type as “Standard” and then click the “Create API Key” button.

New API Key

Properties

FRIENDLY NAME

KEY TYPE Standard

Standard Keys cannot manage API Keys, Account Configuration, and Sub Accounts

Create API Key Cancel

Now you will be presented with the details of your newly created API Key. The “SID” and “SECRET” values are used for authentication along with the Account SID value that we saved earlier.

Open the `.env` file again in your text editor, and add two more lines to it to record the details of your API key:

```
1 TWILIO_ACCOUNT_SID=<your-twilio-account-sid>
2 TWILIO_API_KEY_SID=<your-twilio-api-key-sid>
3 TWILIO_API_KEY_SECRET=<your-twilio-api-key-secret>
```

Once you have your API key safely written to the `.env` file you can leave the API Keys page. Note that if you ever lose your API key secret you will need to generate a new key.

The information contained in your `.env` file is private. Make sure you don't share this file with anyone. If you plan on storing your project under source control it would be a good idea to configure this file so that it is ignored, because you do not want to ever commit this file by mistake.

Create a Python virtual environment

Following best practices, we are going to create a virtual environment where we will install our Python dependencies.

If you are using a Unix or MacOS system, open a terminal and enter the following commands to do the tasks described above:

```
1 $ python -m venv venv
2 $ source venv/bin/activate
3 (venv) $ pip install twilio flask python-dotenv
```

For those of you following the tutorial on Windows, enter the following commands in a command prompt window:

```
1 $ python -m venv venv
2 $ venv\Scripts\activate
3 (venv) $ pip install twilio flask python-dotenv
```

The last command uses `pip`, the Python package installer, to install the three Python packages that we are going to use in this project, which are:

- The Twilio Python Helper library, to work with the Twilio APIs
- The Flask framework, to create the web application
- Python-dotenv, to import the contents of our `.env` file as environment variables

For your reference, at the time this tutorial was released these were the versions of the above packages and their dependencies:

```
1 certifi==2020.4.5.1
2 chardet==3.0.4
3 click==7.1.1
4 Flask==1.1.2
5 idna==2.9
6 itsdangerous==1.1.0
7 Jinja2==2.11.2
8 MarkupSafe==1.1.1
9 PyJWT==1.7.1
10 python-dotenv==0.12.0
11 pytz==2019.3
```

```
12 requests==2.23.0
13 six==1.14.0
14 twilio==6.38.1
15 urllib3==1.25.8
16 Werkzeug==1.0.1
```

Creating a web server

Our project is going to be designed as a single page application. It will be driven by a web server that will serve the HTML, CSS and JavaScript files to clients, as well as respond to asynchronous requests issued from the JavaScript code running in the browser.

We'll start with the web server since it is such a core piece of the project. Once we have the web server running we'll start adding all the other pieces that we need.

As mentioned in the requirements section, we will be using the Flask framework to implement the logic in our web server. Since this is going to be a simple project we will code the entire server in a single file named *app.py*.

Below you can see the first version of our web server. Copy the code into a *app.py* file in the project directory.

```
1 from flask import Flask, render_template
2
3 app = Flask(__name__)
4
5
6 @app.route('/')
7 def index():
8     return render_template('index.html')
```

The `app` variable is called the “application instance”. Its purpose is to provide the support functions we need to implement our web server. We use the `app.route` decorator to define a mapping between URLs and Python functions. In this particular example, when a client requests the root URL for our server, Flask will run our `index()` function and expect it will provide the response. The implementation of our `index()` function renders a *index.html* file that we are yet to write. This file is going to contain the HTML definition of the main and only web page of our video chat application.

Even though it is still very early in the life of our project, we are ready to start the web server. If you are using a Linux or MacOS computer, use the following command:

```
1 (venv) $ FLASK_ENV=development flask run
```

If you use a Windows computer, use the following commands instead:

```
1 (venv) $ set FLASK_ENV=development
2 (venv) $ flask run
```

You should see something like the following output once the server starts:

```
1 * Environment: development
2 * Debug mode: on
3 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
4 * Restarting with stat
5 * Debugger is active!
6 * Debugger PIN: 274-913-316
```

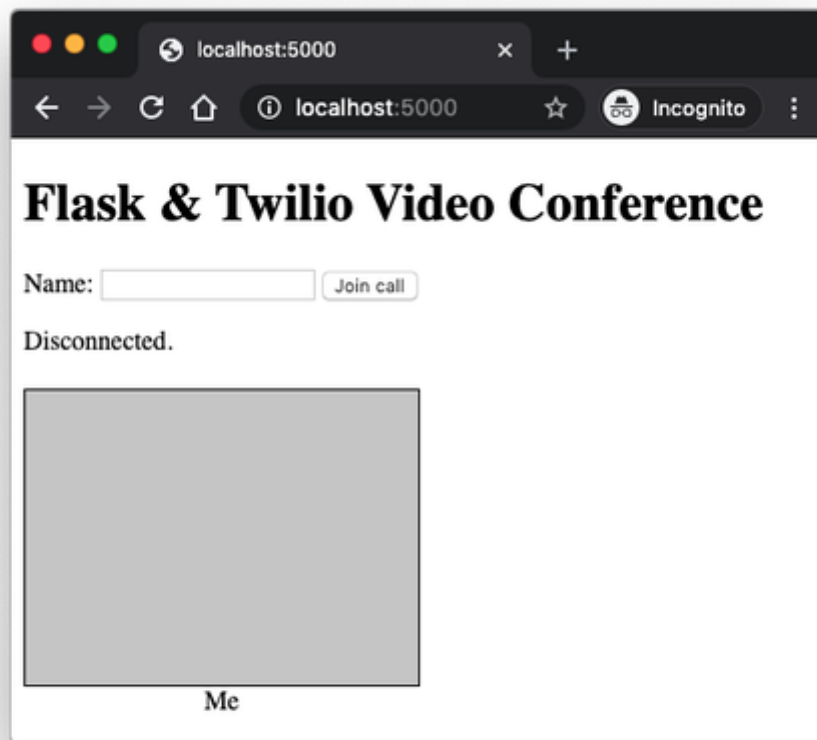
At this point you have the web server running and ready to receive requests. We have also enabled Flask's debug mode, which will trigger the web server to restart itself whenever changes are made to the application, so you can now leave this terminal window alone while we begin to code the components of our project.

If you try to connect to the application from your web browser you will receive a "template not found" error, because we haven't yet written the *index.html* file referenced by our main and only route. We will write this file in the next section and then we will have a first running version of the application.

Application page layout

Our page design is going to be very simple. We'll include a title, a web form where the user can enter their name and join or leave video calls, and then the content area, where the video streams for all the participants will be shown. For now we'll add a placeholder video for ourselves.

Here is how the page will look:



To create this page we need a combination of HTML and CSS. Below you can see the *templates/index.html* file.

```

1 <!doctype html>
2 <html>
3   <head>
4     <link rel="stylesheet" type="text/css" href="{{ url_for('static', filer
5   </head>
6   <body>
7     <h1>Flask & Twilio Video Conference</h1>
8     <form>
9       <label for="username">Name: </label>
10      <input type="text" name="username" id="username">
11      <button id="join_leave">Join call</button>
12    </form>
13    <p id="count"></p>
14    <div id="container" class="container">
15      <div id="local" class="participant"><div></div><div>Me</div></div>
16      <!-- more participants will be added dynamically here -->

```

```

17         </div>
18
19         <script src="//media.twiliocdn.com/sdk/js/video/releases/2.3.0/twilio-v
20         <script src="{{ url_for('static', filename='app.js') }}"></script>
21     </body>
22 </html>

```

The `<head>` section of this file references a *styles.css* file. We are using the `url_for()` function from Flask to generate the correct URL for it. This is nice, because all we need to do is put the file in the *static* directory and let Flask generate the URL. If you were wondering what is the difference between a template file and a static file this is exactly it; template files can have placeholders that are generated dynamically when the `render_template()` function you've seen above runs.

The `<body>` section of the page defines the following elements:

- An `<h1>` title
- A `<form>` element with name field and submit button
- A `<p>` element where we'll show connection status and participant count
- A container `<div>` with one participant identified with the name `local` where we'll show our own video feed. More participants will be added dynamically as they join the video call
- Each participant's `<div>` contains an empty `<div>` where the video will be displayed and a second `<div>` where we'll display the name.
- Links to two JavaScript files that we'll need: the official release of the *twilio-video.js* library and a *app.js* that we will write soon.

The contents of the *static/styles.css* file are below:

```

1 .container {
2     margin-top: 20px;
3     width: 100%;
4     display: flex;

```

```
5     flex-wrap: wrap;
6 }
7 .participant {
8     margin-bottom: 5px;
9     margin-right: 5px;
10 }
11 .participant div {
12     text-align: center;
13 }
14 .participant div:first-child {
15     width: 240px;
16     height: 180px;
17     background-color: #ccc;
18     border: 1px solid black;
19 }
20 .participant video {
21     width: 100%;
22     height: 100%;
23 }
```

These CSS definitions are all dedicated to the layout of the “container” `<div>` element, which is structured as a flexbox so that participants are automatically added to the right and wrapped to the next line as needed according to the size of the browser window.

The `.participant div:first-child` definition applies to the first child element of any `<div>` elements that have the `participant` class. Here we are constraining the size of the video to 240x180 pixels. We also have a darker background and a black border, just so that we can see a placeholder for the video window. The background is also going to be useful when the dimensions of the video do not exactly match our aspect ratio. Feel free to adjust these options to your liking.

With the HTML and CSS files in place, the server should be able to respond to your web browser and show the basic page layout you’ve seen above. While the server is running, open your web browser and type `http://localhost:5000` on the address bar to see the first version of the application running.

Displaying your own video feed

If you looked in the browser’s network log you likely noticed that the browser tried to load the `app.js` file that we reference at the bottom of `index.html` and this failed because we don’t have

that file in our project yet. We are now going to write our first function in this file to add our own video feed to the page.

Create the file and add the following code to *static/app.js*:

```
1 function addLocalVideo() {  
2     Twilio.Video.createLocalVideoTrack().then(track => {  
3         let video = document.getElementById('local').firstChild;  
4         video.appendChild(track.attach());  
5     });  
6 };  
7  
8 addLocalVideo();
```

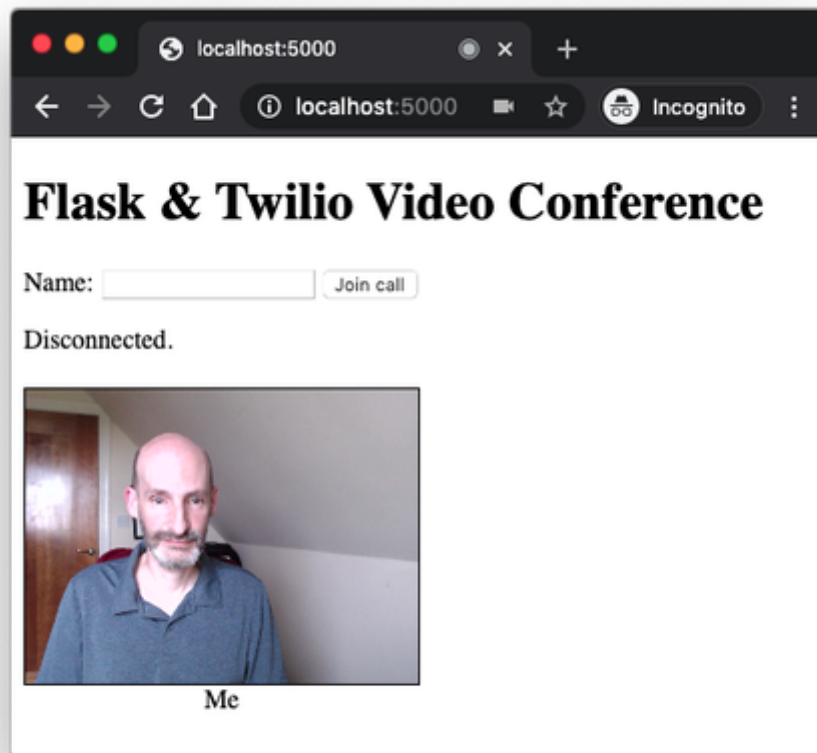
The `addLocalVideo()` function uses the Twilio Programmable Video JavaScript library to create a local video track. The `createLocalVideoTrack()` function from the library is asynchronous and returns a promise, so we use the `then()` method to add some logic in a callback function after the video track is created.

The callback function receives a LocalVideoTrack object as an argument. We use its `attach()` method to add the video element to the first `<div>` child of the `local` element. In case this is confusing, let's review the structure of the local participant from the *index.html* file:

```
1         <div id="local" class="participant"><div></div><div>Me</div></div>
```

You can see here that the `local` element has two `<div>` elements as children. The first is empty, and this is the element to which we are attaching the video. The second `<div>` is for the label that appears below the video.

You can refresh the page in the browser and you should have your video displayed. Note that most browsers will ask for your permission before enabling the camera.



Generating an access token for a participant

Twilio takes security very seriously. Before users can join a video call, the application must verify the user is allowed and generate an access token for them. The tokens must be generated in the Python server, as the secrets we stored in the `.env` file are required for this process.

In a real-world application, this is the place where the application would authenticate the user wanting to join the call. The connection request in such an application would likely include a password, authentication cookie or some other form of identification in addition to the user's name. An access token to the video chat room would only be generated after the user requesting access to the video call is properly authenticated.

The updated `app.py` file is shown below.

```
1 import os
2 from dotenv import load_dotenv
3 from flask import Flask, render_template, request, abort
4 from twilio.jwt.access_token import AccessToken
```

```
5 from twilio.jwt.access_token.grants import VideoGrant
6
7 load_dotenv()
8 twilio_account_sid = os.environ.get('TWILIO_ACCOUNT_SID')
9 twilio_api_key_sid = os.environ.get('TWILIO_API_KEY_SID')
10 twilio_api_key_secret = os.environ.get('TWILIO_API_KEY_SECRET')
11
12 app = Flask(__name__)
13
14
15 @app.route('/')
16 def index():
17     return render_template('index.html')
18
19
20 @app.route('/login', methods=['POST'])
21 def login():
22     username = request.get_json(force=True).get('username')
23     if not username:
24         abort(401)
25
26     token = AccessToken(twilio_account_sid, twilio_api_key_sid,
27                         twilio_api_key_secret, identity=username)
28     token.add_grant(VideoGrant(room='My Room'))
29
30     return {'token': token.to_jwt().decode()}
```

As mentioned above, we need the three secrets we stored in the `.env` file earlier, so we call the `load_dotenv()` function from the `python-dotenv` package to import those secrets, and then we assign them to variables for convenience.

The token generation happens in a new route that we are going to invoke from the JavaScript side, attached to the `/login` URL. The function will receive the username in a JSON payload. Because this is a simple application, the only authentication we are going to perform on the user is that the username is not empty. If validation fails, a 401 error is returned to indicate that the user does not have access to the video call. As discussed above, a real-world application would implement a more thorough authentication mechanism here.

The token is generated using the `AccessToken` helper class from the Twilio Python Helper library. We attach a video grant for a video room called “My Room”. A more complex

application can work with more than one video room and decide which room or rooms this user can enter.

The token is returned in a JSON payload in the format:

```
1 {  
2   "token": "the-token-goes-here"  
3 }
```

Handling the connection form

Next we are going to implement the handling of the connection form in the web page. The participant will enter their name and then click the “Join call” button. Once the connection is established the same button will be used to disconnect from the call.

To manage the form button we have to attach a handler for the `click` event. The changes to *static/app.js* are shown below.

```
1 let connected = false;  
2 const usernameInput = document.getElementById('username');  
3 const button = document.getElementById('join_leave');  
4 const container = document.getElementById('container');  
5 const count = document.getElementById('count');  
6 let room;  
7  
8 function addLocalVideo() { /* no changes in this function */ };  
9  
10 function connectButtonHandler(event) {  
11   event.preventDefault();  
12   if (!connected) {  
13     let username = usernameInput.value;  
14     if (!username) {  
15       alert('Enter your name before connecting');  
16       return;  
17     }  
18     button.disabled = true;  
19     button.innerHTML = 'Connecting...';  
20     connect(username).then(() => {  
21       button.innerHTML = 'Leave call';  
22       button.disabled = false;
```



```
23     }).catch(() => {
24         alert('Connection failed. Is the backend running?');
25         button.innerHTML = 'Join call';
26         button.disabled = false;
27     });
28 }
29 else {
30     disconnect();
31     button.innerHTML = 'Join call';
32     connected = false;
33 }
34 };
35
36 addLocalVideo();
37 button.addEventListener('click', connectButtonHandler);
```

You can see that we now have a few global variables declared at the top. Four of them are for convenient access to elements in the page, such as the name entry field, the submit button in our web form and so on. The `connected` boolean tracks the state of the connection, mainly to help decide if a button click needs to connect or disconnect. The `room` variable will hold the video chat room object once we have it.

At the very bottom of the script we attach the `connectButtonHandler()` function to the click event on the form button. The function is somewhat long, but it mostly deals with validating that the user entered a name and updating how the button looks as the state of the connection changes. If you filter out the form management you can see that the actual connection and disconnection are handled by two functions `connect()` and `disconnect()` that we are going to write in the following sections.

Connecting to a video chat room

We now reach the most important (and also most complex!) part of our application. To connect a user to the video chat room the JavaScript application running in the web browser must perform two operations in sequence. First, the client needs to contact the web server and request an access token for the user, and then once the token is received, the client has to call the twilio-video library with this token to make the connection.

```
1 let connected = false;
2 const usernameInput = document.getElementById('username');
```

```

3  const button = document.getElementById('join_leave_button');
4  const container = document.getElementById('container');
5  const count = document.getElementById('count');
6  let room;
7
8  function addLocalVideo() { /* no changes in this function */ };
9
10 function connectButtonHandler(event) { /* no changes in this function */ };
11
12 function connect(username) {
13     let promise = new Promise((resolve, reject) => {
14         // get a token from the back end
15         fetch('/login', {
16             method: 'POST',
17             body: JSON.stringify({username: username})
18         }).then(res => res.json()).then(data => {
19             // join video call
20             return Twilio.Video.connect(data.token);
21         }).then(_room => {
22             room = _room;
23             room.participants.forEach(participantConnected);
24             room.on('participantConnected', participantConnected);
25             room.on('participantDisconnected', participantDisconnected);
26             connected = true;
27             updateParticipantCount();
28             resolve();
29         }).catch(() => {
30             reject();
31         });
32     });
33     return promise;
34 };

```

The `connect()` function returns a promise, to which the caller can use to attach actions to be performed once the connection is established, or also to handle errors. Internally, the promise outcome is controlled via the `resolve()` and `reject()` functions that are passed as arguments into the execution function passed in the `Promise()` constructor. You can see calls to these functions sprinkled throughout the connection logic. A call to `resolve()` will trigger the caller's success callback, while a call to `reject()` will do the same for the error callback.

The connection logic has two steps as indicated above. First we use the browser's `fetch()` function to send a request to the `/login` route in the Flask application that we created above.

This function returns a promise as well, so we use the `then(...).catch(...)` handlers to provide success and failure callbacks.

If the fetch call fails we in turn call `reject()` to fail our own promise. If the call succeeds, we decode the JSON payload into the `data` variable and then call the `connect()` function from the `twilio-video` library passing our newly acquired token.

The video connection call is also a promise, so once again we continue chaining our functions with `then(...)`. The success handler for the connection is the most interesting part, where we have been connected to the video chat room and need to arrange the `container` part of the page to reflect that.

This success callback receives a `_room` argument, which represents the video room. Since this is a useful variable, we assign `_room` to the global variable `room`, so that the rest of the application can access this room when needed.

The `room.participants` array contains the list of people already in the call. For each of these we have to add a `<div>` section that shows the video and the name. This is all encapsulated in the `participantConnected()` function, so we invoke it for each participant. We also want any future participants to be handled in the same way, so we set up a handler for the `participantConnected` event pointing to the same function. The `participantDisconnected` event is also important, as we'd want to remove any participants that leave the call, so we set up a handler for this event as well.

At this point we are fully connected, so we can indicate that in the `connected` boolean variable. The final action we take is to update the `<p>` element that shows the connection status to show the participant count. This is done in a separate function because we'll need to do this in several places. The function updates the text of the element based on the length of the `room.participants` array. Add the implementation of this function to *static/app.js*.

```
1 function updateParticipantCount() {
2     if (!connected)
3         count.innerHTML = 'Disconnected.';
4     else
5         count.innerHTML = (room.participants.size + 1) + ' participants online.'
6 };
```

Note that the `room.participants` array includes every participant except ourselves, so the total number of people in a call is always one more than the size of the list.

Connecting and disconnecting participants

You saw in the previous section that when a participant joins the call we call the `participantConnected` handler. This function needs to create a new `<div>` inside the `container` element, following the same structure we used for the `local` element that shows our own video stream.

Below you can see the implementation of the `participantConnected()` function along with the `participantDisconnected()` counterpart and a few auxiliary functions, all of which also goes in *static/app.js*.

```
1 function participantConnected(participant) {
2   let participantDiv = document.createElement('div');
3   participantDiv.setAttribute('id', participant.sid);
4   participantDiv.setAttribute('class', 'participant');
5
6   let tracksDiv = document.createElement('div');
7   participantDiv.appendChild(tracksDiv);
8
9   let labelDiv = document.createElement('div');
10  labelDiv.innerHTML = participant.identity;
11  participantDiv.appendChild(labelDiv);
12
13  container.appendChild(participantDiv);
14
15  participant.tracks.forEach(publication => {
16    if (publication.isSubscribed)
17      trackSubscribed(tracksDiv, publication.track);
18  });
19  participant.on('trackSubscribed', track => trackSubscribed(tracksDiv, track));
20  participant.on('trackUnsubscribed', trackUnsubscribed);
21
22  updateParticipantCount();
23 };
24
25 function participantDisconnected(participant) {
26   document.getElementById(participant.sid).remove();
27   updateParticipantCount();
28 }
```

```
28  };
29
30  function trackSubscribed(div, track) {
31      div.appendChild(track.attach());
32  };
33
34  function trackUnsubscribed(track) {
35      track.detach().forEach(element => element.remove());
36  };
```

The `participantConnected()` callback receives a Participant object from the `twilio-video` library. The two important properties of this object are `participant.sid` and `participant.identity`, which are a unique session identifier and name respectively. The `identity` attribute comes directly from the token we generated. Recall that we passed `identity=username` in our Python token generation function.

The HTML structure for a participant is similar to the one we used for the local video. The big difference is that we now need to create this structure dynamically using the browser's DOM API. This is the markup that we need to create for each participant:

```
1  <div id="{ participant.sid }" class="participant">
2      <div></div>  <!-- the video and audio tracks will be attached to this div -
3      <div>{ participant.name }</div>
4  </div>
```

At the start of the `participantConnected()` function you can see that we create a `participant_div`, to which we add a `tracks_div` and a `label_div` as children. We finally add the `participant_div` as a child of `container`, which is the top-level `<div>` element where we have all the participants of the call.

The second part of the function deals with attaching the video and audio tracks to the `tracks_div` element we just created. We run a loop through all the tracks the participants export, and following the basic usage shown in the library's documentation we attach those to which we are subscribed. The actual track attachment is handled in a `trackSubscribed()` auxiliary function that is defined right below.

In more advanced usages of this library a participant can dynamically add or remove tracks during a call (for example if they were to turn off their video temporarily, mute their audio, or

even start sharing their screen). Because we want to respond to all those track changes, we also create event handlers for the `trackSubscribed` and `trackUnsubscribed` events, which use the `attach()` and `detach()` methods of the track object to add and remove the HTML elements that carry the feeds.

Disconnecting from the chat room

The counterpart of the `connect()` function is `disconnect()`, which has to restore the state of the page to how it was previous to connecting. This is a lot simpler, as it mostly involves removing all the children of the `container` element except the first one, which is our local video stream.

```
1 function disconnect() {  
2     room.disconnect();  
3     while (container.lastChild.id != 'local')  
4         container.removeChild(container.lastChild);  
5     button.innerHTML = 'Join call';  
6     connected = false;  
7     updateParticipantCount();  
8 };
```

As you can see here we remove all children of the container element starting from the end and until we come upon the `<div>` with the id `local`, which is the one that we created statically in the `index.html` page. We also use the opportunity to update our `connected` global variable, change the text of the connect button and refresh the `<p>` element to show a “Disconnected” message.

Running your video chat server

If you started your web server in the early stages of this tutorial, make sure it is still running. If it isn't running, start it one more time with the following command:

```
1 (venv) $ FLASK_ENV=development flask run
```

If you use a Windows computer, use the following commands instead:

```
1 (venv) $ set FLASK_ENV=development
2 (venv) $ flask run
```

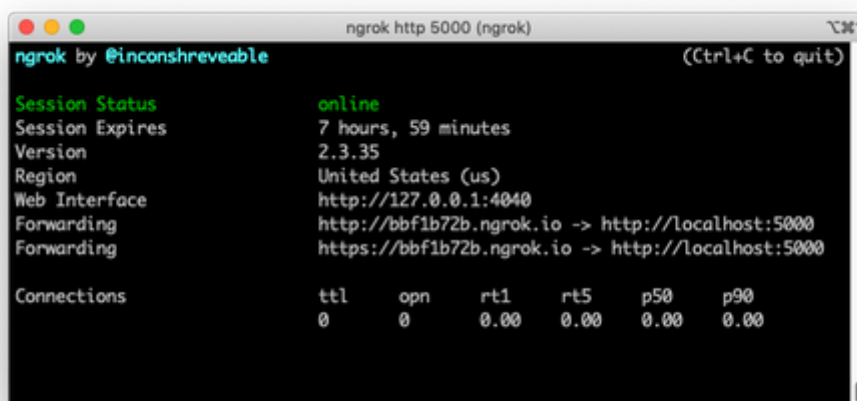
With the server running you can connect from the same computer by entering `http://localhost:5000` on the address bar of your web browser. But of course, you very likely want to also connect from a second computer or a smartphone, or maybe even invite a friend to join your video chat. This requires one more step, because the server is only running internally on your computer and is not accessible from the Internet.

There are a few different ways to expose the server to the Internet. A quick and easy way to do this is to use [ngrok](#), a handy utility that creates a tunnel between our locally running server and a public URL on the ngrok.io domain. If you don't have ngrok installed you can [download a copy for Windows, MacOS or Linux](#).

With the Flask server running, open a second terminal window and start ngrok as follows:

```
1 $ ngrok http 5000
```

Your terminal will now show something similar to this screen:



```
ngrok http 5000 (ngrok)
ngrok by @inconshreveable (Ctrl+C to quit)

Session Status      online
Session Expires     7 hours, 59 minutes
Version             2.3.35
Region              United States (us)
Web Interface        http://127.0.0.1:4040
Forwarding           http://bbf1b72b.ngrok.io -> http://localhost:5000
Forwarding           https://bbf1b72b.ngrok.io -> http://localhost:5000

Connections         ttl    opn    rt1    rt5    p50    p90
0                  0      0      0.00   0.00   0.00   0.00
```

Find the `Forwarding` lines to see what is the public URL that ngrok assigned to your server. Use the one that starts with `https://`, since many browsers do not allow unencrypted sites to access the camera and the microphone. In the example above, the public URL is `https://bbf1b72b.ngrok.io`. Yours is going to be similar, but the first component of the domain is going to be different every time you start ngrok.

While having both the Flask server and ngrok running on your computer you can use the public <https://URL> from ngrok to connect to your server from other computers and smartphones, so you are now ready to invite your friends to video chat with you!

Conclusion

I hope this was a fun and interesting tutorial. If you decide to use it as a base to build your own video chat project, you should know that the [Twilio Programmable Video API](#) has many more features that we haven't explored, including the option to:



- [record calls](#)
- [share custom data tracks](#)
- [screen share](#)
- and more.

I can't wait to see what you build!

[Miguel Grinberg](#) is a Python Developer for Technical Content at Twilio. Reach out to him at [mgrinberg \[at\] twilio \[dot\] com](mailto:mgrinberg@twilio.com) if you have a cool Python project you'd like to share on the Twilio blog!

RATE THIS POST 

AUTHORS |  [Miguel Grinberg](#)

REVIEWERS |  [Liz Moy](#)
 [Kelley Robinson](#)

Search

Build the future of communications. Start today with Twilio's APIs and services.

POSTS BY STACK

JAVA .NET RUBY PHP PYTHON SWIFT ARDUINO JAVASCRIPT

POSTS BY PRODUCT

SMS AUTHY VOICE TWILIO CLIENT MMS VIDEO TASK ROUTER FLEX SIP IOT
PROGRAMMABLE CHAT STUDIO

CATEGORIES

Code, Tutorials and Hacks

Customer Highlights

Developers Drawing The Owl

News

Stories From The Road

The Owl's Nest: Inside Twilio

TWITTER

FACEBOOK

Developer stories to your inbox.

Subscribe to the Developer Digest, a monthly dose of all things code.

Enter your email...

You may unsubscribe at any time using the unsubscribe link in the digest email. See our [privacy policy](#) for more information.

NEW!

Tutorials

Sample applications that cover common use cases in a variety of languages. Download, test drive, and tweak them yourself.

[Get started](#)

SIGN UP AND START BUILDING

Not ready yet? [Talk to an expert.](#)



ABOUT

LEGAL

COPYRIGHT © 2020 TWILIO INC.

ALL RIGHTS RESERVED.

PROTECTED BY RECAPTCHA - [PRIVACY](#) - [TERMS](#)