Next.JS

> Learn how to create a web application that enables voice conversations with ElevenLabs AI agents

This tutorial will guide you through creating a web client that can interact with a Conversational AI agent. You'll learn how to implement real-time voice conversations, allowing users to speak with an AI agent that can listen, understand, and respond naturally using voice synthesis.

What You'll Need

1. An ElevenLabs agent created following [this guide](/docs/conversational-ai/quickstart)
2. `npm` installed on your local system.
3. We'll use Typescript for this tutorial, but you can use Javascript if you prefer.

<Note>
  Looking for a complete example? Check out our [Next.js demo on
  GitHub](https://github.com/elevenlabs/elevenlabs-examples/tree/main/examples/conversational-ai/nextjs).
</Note>

<Frame background="subtle">

</Frame>

Setup

<Steps>
  <Step title="Create a new Next.js project">
    Open a terminal window and run the following command:

    ```bash
    npm create next-app my-conversational-agent
    ```

    It will ask you some questions about how to build your project. We'll follow the default suggestions for this tutorial.
  </Step>

  <Step title="Navigate to project directory">
    ```shell
    cd my-conversational-agent
    ```
  </Step>

  <Step title="Install the ElevenLabs dependency">
    ```shell
    npm install @11labs/react
    ```
  </Step>

  <Step title="Test the setup">
    Run the following command to start the development server and open the provided URL in your browser:

    ```shell
    npm run dev
    ```

    <Frame background="subtle">

    </Frame>
  </Step>

````
</Steps>

## Implement Conversational AI

<Steps>
  <Step title="Create the conversation component">
    Create a new file `app/components/conversation.tsx`:

    ```tsx app/components/conversation.tsx
    'use client';

    import { useConversation } from '@11labs/react';
    import { useCallback } from 'react';

    export function Conversation() {
      const conversation = useConversation({
        onConnect: () => console.log('Connected'),
        onDisconnect: () => console.log('Disconnected'),
        onMessage: (message) => console.log('Message:', message),
        onError: (error) => console.error('Error:', error),
      });


      const startConversation = useCallback(async () => {
        try {
          // Request microphone permission
          await navigator.mediaDevices.getUserMedia({ audio: true });

          // Start the conversation with your agent
          await conversation.startSession({
            agentId: 'YOUR_AGENT_ID', // Replace with your agent ID
          });

        } catch (error) {
          console.error('Failed to start conversation:', error);
        }
      }, [conversation]);

      const stopConversation = useCallback(async () => {
        await conversation.endSession();
      }, [conversation]);

      return (
        <div className="flex flex-col items-center gap-4">
          <div className="flex gap-2">
            <button
              onClick={startConversation}
              disabled={conversation.status === 'connected'}
              className="px-4 py-2 bg-blue-500 text-white rounded disabled:bg-gray-300"
            >
              Start Conversation
            </button>
            <button
              onClick={stopConversation}
              disabled={conversation.status !== 'connected'}
              className="px-4 py-2 bg-red-500 text-white rounded disabled:bg-gray-300"
            >
              Stop Conversation
            </button>
          </div>

          <div className="flex flex-col items-center">
            <p>Status: {conversation.status}</p>
            <p>Agent is {conversation.isSpeaking ? 'speaking' : 'listening'}</p>
          </div>
````
````

```
        </div>
      );
    }
    ```
  </Step>

  <Step title="Update the main page">
    Replace the contents of `app/page.tsx` with:

    ```tsx app/page.tsx
    import { Conversation } from './components/conversation';

    export default function Home() {
      return (
        <main className="flex min-h-screen flex-col items-center justify-between p-24">
          <div className="z-10 max-w-5xl w-full items-center justify-between font-mono text-
sm">
            <h1 className="text-4xl font-bold mb-8 text-center">
              ElevenLabs Conversational AI
            </h1>
            <Conversation />
          </div>
        </main>
      );
    }
    ```
  </Step>
</Steps>

<Accordion title="(Optional) Authenticate the agents with a signed URL">
  <Note>
    This authentication step is only required for private agents. If you're using a public
agent, you
    can skip this section and directly use the `agentId` in the `startSession` call.
  </Note>

  If you're using a private agent that requires authentication, you'll need to generate
  a signed URL from your server. This section explains how to set this up.

  ### What You'll Need

  1. An ElevenLabs account and API key. Sign up [here](https://www.elevenlabs.io/sign-up).

  <Steps>
    <Step title="Create environment variables">
      Create a `.env.local` file in your project root:

      ```yaml .env.local
      ELEVENLABS_API_KEY=your-api-key-here
      NEXT_PUBLIC_AGENT_ID=your-agent-id-here
      ```

      <Warning>
        1. Make sure to add `.env.local` to your `.gitignore` file to prevent accidentally
committing sensitive credentials to version control.
        2. Never expose your API key in the client-side code. Always keep it secure on the
server.
      </Warning>
    </Step>

    <Step title="Create an API route">
      Create a new file `app/api/get-signed-url/route.ts`:

      ```tsx app/api/get-signed-url/route.ts
      import { NextResponse } from 'next/server';
```

```
      export async function GET() {
        try {
          const response = await fetch(
            `https://api.elevenlabs.io/v1/convai/conversation/get_signed_url?
agent_id=${process.env.NEXT_PUBLIC_AGENT_ID}`,
            {
              headers: {
                'xi-api-key': process.env.ELEVENLABS_API_KEY!,
              },
            }
          );

          if (!response.ok) {
            throw new Error('Failed to get signed URL');
          }

          const data = await response.json();
          return NextResponse.json({ signedUrl: data.signed_url });
        } catch (error) {
          return NextResponse.json(
            { error: 'Failed to generate signed URL' },
            { status: 500 }
          );
        }
      }
      ```
    </Step>

    <Step title="Update the Conversation component">
      Modify your `conversation.tsx` to fetch and use the signed URL:

      ```tsx app/components/conversation.tsx {5-12,19,23}
      // ... existing imports ...

      export function Conversation() {
        // ... existing conversation setup ...
        const getSignedUrl = async (): Promise<string> => {
          const response = await fetch("/api/get-signed-url");
          if (!response.ok) {
            throw new Error(`Failed to get signed url: ${response.statusText}`);
          }
          const { signedUrl } = await response.json();
          return signedUrl;
        };

        const startConversation = useCallback(async () => {
          try {
            // Request microphone permission
            await navigator.mediaDevices.getUserMedia({ audio: true });

            const signedUrl = await getSignedUrl();

            // Start the conversation with your signed url
            await conversation.startSession({
              signedUrl,
            });

          } catch (error) {
            console.error('Failed to start conversation:', error);
          }
        }, [conversation]);

        // ... rest of the component ...
      }
```

```
      ```

      <Warning>
        Signed URLs expire after a short period. However, any conversations initiated before
expiration will continue uninterrupted. In a production environment, implement proper error
handling and URL refresh logic for starting new conversations.
      </Warning>
    </Step>
  </Steps>
</Accordion>

Next Steps

Now that you have a basic implementation, you can:

1. Add visual feedback for voice activity
2. Implement error handling and retry logic
3. Add a chat history display
4. Customize the UI to match your brand

<Info>
  For more advanced features and customization options, check out the
  [@11labs/react](https://www.npmjs.com/package/@11labs/react) package.
</Info>
```
```