Get Started
Learn how to build Daily Bots by example

1. Sign up to get your API Key
To get started, you'll need to sign up for a Daily Bots account and enter your credit card information to get your API Key.

Sign Up
Get your API key to get started now!

2. Set up and run the example project
As described in the Architecture guide, you need a Client Application and a Client Server to build a bot. We've wrapped up both of those in a single Next.js app to make it easy to get started. You can clone the repo by running git clone https://github.com/daily-demos/daily-bots-web-demo.git, or visit the repo to learn more.

Once you've cloned the repo, you can run npm install or yarn inside the repo's folder to install all of the dependencies you'll need.

Configuring your environment
Next, you'll need to configure your environment. Find the file named env.example and rename it to .env.local. The file currently has four keys in it:

NEXT_PUBLIC_BASE_URL: This controls how the 'Client Application' part of the app finds the 'Client Server' part. You should leave this as-is for now. The /api path is pointing to /app/api/route.ts, which we'll explore later.
DAILY_API_KEY: Put the API key you got in step 1 here.
OPENAI_API_KEY: If you want to use OpenAI models, you'll need to supply your own key. You can set it here, and the Client Server will use it when you run your bot.
DAILY_BOTS_URL: This tells your app where to find the hosted backend. You should set this to https://api.daily.co/v1/bots/start.
With those values set, you should be ready to run your bot. Just run yarn dev or npm run dev in a terminal. If everything works, you should be able to visit http://localhost:3000 to see your bot. Go ahead and have a quick chat to verify it works!

3. Explore the example project
There are a few important parts of the example project that you'll want to get familiar with.

Bot configuration: /rtvi.config.ts

rtvi.config.ts

```
export const defaultBotProfile = "voice_2024_10";

export const defaultServices = {
  stt: "deepgram",
  llm: "anthropic",
  tts: "cartesia",
};

export const defaultServiceOptions = {
  service_options: {
    deepgram: {
      model: "nova-2-general",
      language: "en",
    },
    anthropic: {
      model: "claude-3-5-sonnet-latest",
    },
  },
};

export const defaultConfig = [
  {
    service: "tts",
    options: [{ name: "voice", value: "d46abd1d-2d02-43e8-819f-51fb652c1c61" }],
  },
  {
    service: "llm",
    options: [
      {
        name: "initial_messages",
        value: [
          {
            // anthropic: user; openai: system

            role: "system",
            content: "You are a helpful voice bot.",
          },
        ],
      },
      { name: "run_on_config", value: true },
    ],
  },
```

```
];

// These are your app's endpoints, which are used to initiate the /bots/start
// API call or initiate actions
export const defaultEndpoints = {
  connect: "/connect",
  actions: "/actions",
};
```

Daily Bots are extremely configurable. There's a lot you can change about a bot while it's running, but most of the important configuration happens when a bot session starts. It's a good idea to manage as much of that as possible in the rtvi.config.ts file in the root of the project repo. Many of the available options are already in that file, but you can refer to the API Reference and RTVI docs for more available configuration.

Bot initialization and the voiceClient: /app/page.tsx
/app/page.tsx

```
const voiceClient = new RTVIClient({
  params: {
    baseUrl: process.env.NEXT_PUBLIC_BASE_URL || "/api",
    endpoints: defaultEndpoints,
    config: defaultConfig,
  },
  transport: new DailyTransport(),
  enableMic: true,
  enableCam: false,
  timeout: BOT_READY_TIMEOUT,
});

voiceClient.registerHelper(
  "llm",
  new LLMHelper({
    callbacks: {},
  })
) as LLMHelper;
```

The /app/page.tsx file is the main Next.js file for the demo app. It initializes a RTVIClient object that you'll use throughout your app. page.tsx is also where you'll want to register any additional helpers you need.

In the session: /components/Session/index.tsx

/components/Session/index.tsx is the component that controls an active bot session, and it's usually the place you'll want to do a with the voiceClient object. In the example app, you'll see a few things like metrics collection happening in useRTVIClientEvent hooks.

/components/Session/index.tsx

```
useRTVIClientEvent(
  RTVIEvent.Metrics,
  useCallback((metrics) => {
    metrics?.ttfb?.map((m: { processor: string; value: number }) => {
      stats_aggregator.addStat([m.processor, "ttfb", m.value, Date.now()]);
    });
  }, [])
);
```

The Client Server: /app/api/route.ts
/app/api/route.ts

```
const payload = {
  bot_profile: defaultBotProfile,
  service_options: defaultServiceOptions,
  services,
  api_keys: {
    together: process.env.TOGETHER_API_KEY,
    cartesia: process.env.CARTESIA_API_KEY,
  },
  config: [...config],
};
```

When a user wants to talk to a bot, your Client Application will make a request to a Client Server, which is a server process you control. That server then makes a POST request to the Daily Bots API to start a bot. In that request, you need to specify which Bot Profile you want to use. You can also include API keys for services that are supported by RTVI and Pipecat but not built into Daily Bots, such as OpenAI. In the example above, the Client Server is including keys for Together and Cartesia, but that isn't strictly necessary, because both of those service have built-in support in Daily Bots.

4. Check out the other demos!
That should be enough for you to start finding your way around the example app. If you want to see some other interesting use cases, you can check out these hosted examples. The source code for these examples is also available in branches in the RTVI examples repo.

Talk To The Bot

A great starting place showing off how to interact with a bot and turn all the possible knobs. Built using React and NextJS.

Weather Reporter
Demonstrates using LLM Tooling features to build out more complex interactions with the bot. See the cb/function-calling branch for the source code.

Vision
Demonstrates using our vision bot profile to interact and send video frames to the bot for even more context. Check out the khk/vision-for-launch branch for the code.

Introduction
Build Your First Daily Bot
x
youtube
linkedin

RTVIClient

The client-side component that speaks to the Daily Bot

Overview

The `RTVIClient` is the component you will primarily interface with for interacting with your Daily Bot. This client is part of the RTVI suite of client libraries, all of which follow the same API design detailed below. The purpose of this component is to:

- Provides a `connect()` method that handshakes / authenticates with your bot service
- Configures your bot services
- Manages your media connections
- Provides methods, callbacks and events for interfacing with your bot

The `RTVIClient` works in conjunction with the `DailyTransport` from the `realtime-ai-daily` library to enable voice and video communication with the bot. A `DailyTransport` is required for Daily Bots.

This page provides details on the most common parameters, methods, and callbacks you will use with the `RTVIClient` along with specific API expectations required for Daily Bots. For full reference material and installation instructions, visit the docs for your corresponding library.

Source Docs

React

RTVI React SDK

JavaScript

RTVI Javascript SDK

Swift

RTVI iOS SDK

Kotlin

RTVI Android SDK

API Reference

Constructor Parameters

`baseUrl`

`string`

`required`

Handshake URL to your hosted endpoint that triggers authentication, transport session creation and bot instantiation.

The `RTVIClient` will send a JSON `POST` request to this URL and pass the local configuration (`config`) as a body param.

The `RTVIClient` expects this endpoint to return the response from the Daily Bots endpoint to establish the connection. Doing so will then established the connection automatically.

Example:

```
{
  params: {
    baseUrl: "/api/vi";
  }
}
```

`endpoints`

`Object <{ [key: string]: string`

`required`

These are your local app's endpoints that your client will use to connect to the `/bots/start` endpoint or initiate bot actions. For example, if you're running a Next.js application, you may have a route.ts that is used to call the `/bots/start` endpoint at `/api/v1/start-bot/route.ts`. Given that example, your `baseUrl` would be `/api/v1` and your `connect` endpoint would be `start-bot`. The same pattern applies for the `actions` endpoint.

Example:

```
{
```

```
  params: {
    baseUrl: "/api/v1",
    endpoints: {
      connect: "start-bot",
      actions: "bot-actions"
    }
  }
}
```

config

`Array <RTVIClientConfigOption[]>`

Pipeline configuration object for your registered services. Must contain a valid `RTVIClientConfig` array.

Client config is passed to the bot at startup, and can be overriden in your server-side endpoint (where sensitive information can be provided, such as API keys.)

See configuration.

Example:

```
{
  params: {
    config: [
      {
        service: "tts",
        options: [
          {
            name: "voice",
            value: "79a125e8-cd45-4c13-8a67-188112f4dd22",
          },
        ],
      },
      {
        service: "llm",
        options: [
          {
            name: "model",
            value: "claude-3-5-sonnet-latest",
          },
```

```
          {
            name: "initial_messages",
            value: [
              {
                role: "user",
                content: [
                  {
                    type: "text",
                    text: "You are a pirate.",
                  },
                ],
              },
            ],
          },
          {
            name: "run_on_config",
            value: true,
          },
        ],
      },
    ],
  }
}
```

`requestData`

`Object`

Pass through custom request body parameters to your defined `connect` endpoint.

Example:

```
{
  params: {
    requestData: {
      services: {
        tts: "cartesia",
        llm: "anthropic"
      }
    }
  }
}
```

Note: `services` is intended to be set on the server-side. If you have a need to pass them from the client-side, you can use `params.requestData`.

`callbacks`

`{ callback:()=>void }`

An array of callback functions. See callbacks.

`enableMic`

`boolean`

`default: "true"`

Enable user's local microphone device.

`enableCamera`

`boolean`

`default: "false"`

Enable user's local webcam device.

`headers`

`Object <{ [key: string]: string }>`

Custom HTTP headers to include in the initial `connect` web request to the `baseUrl`.

Example Code

JavaScript

React

Swift

Kotlin

```
import { RTVIClient } from "realtime-ai";
import { DailyTransport } from "realtime-ai-daily";

const voiceClient = new RTVIClient({
  params: {
    baseUrl: `PATH_TO_YOUR_CONNECT_ENDPOINT`,
```

```
endpoints: {
  connect: "YOUR_CONNECT_ROUTE",
  actions: "YOUR_ACTIONS_ROUTE",
},
config: [
  {
    service: "tts",
    options: [
      {
        name: "voice",
        value: "79a125e8-cd45-4c13-8a67-188112f4dd22",
      },
    ],
  },
  {
    service: "llm",
    options: [
      {
        name: "model",
        value: "claude-3-5-sonnet-latest",
      },
      {
        name: "initial_messages",
        value: [
          {
            role: "user",
            content: [
              {
                type: "text",
                text: "You are a pirate.",
              },
            ],
          },
        ],
      },
      {
        name: "run_on_config",
        value: true,
      },
    ],
  },
],
requestData: {
```

```
      ...anyCustomBodyParams,
      services: {
        tts: "cartesia",
        llm: "anthropic",
      },
    },
  },
  transport: new DailyTransport(),
  enableMic: true,
  enableCam: false,
  callbacks: {
    onBotReady: () => {
      console.log("Bot is ready!");
    },
  },
});

try {
  await voiceClient.connect();
} catch (e) {
  console.log(e.message || "Unknown error occured");
  voiceClient.disconnect();
}
```

Configuration
x
youtube
linkedin

Configuration

Defining the services and their configurations for Daily Bots

The RTVIClient takes a list of services along with the JSON configuration used to set up those services. As an open source component, the services and their configurations is vastly flexible. However, Daily Bots has a pre-defined set of services that can be used and keys to configure them.

The full set of configuration options is detailed in the RTVI API Reference. Also, different services used in your Daily Bots have different capablities. Check out the supported services to learn about available services. Then, view each service page for a full list of available configuration options.

Services

Each bot profile expects a specific set of services to be configured. The most common configuration is to have three services, speech-to-text (STT), LLM, and text-to-speech (TTS), which can be configured as follows:

```
{
  "services": {
    "stt": "<stt service name>",
    "tts": "<tts service name>",
    "llm": "<llm service name>"
  }
}
```

For the list of accepted service name values, see the supported services page. Also, see the bot profiles for a list of available profiles.

Configuration

The configuration object is where you set various options for your services. This should be sent as a part of your RTVIClient constructor but can also be updated at any time using updateConfig().

The config is a list because order matters; configurations are applied sequentially. For example, to configure a TTS service with a new voice and an LLM service with new prompting, specify the TTS first to ensure the voice is applied before the prompting. This ordering principle also applies to service options, ensuring deterministic outcomes for all RTVI implementations.

For Daily Bots you must provide a configuration options for both your llm and tts service. Optionally, you can also specify an stt service, which provides control over model and language parameters. The general format for the configuration object is as follows:

```
{
  "config": [
    {
      "service": <service>, // "tts" || "llm" || "stt"
      "options": [
        {
          "name": "<option name>",
          "value": "<option value>"
        }
      ]
    },
    {
      "service": <service>, // "tts" || "llm" || "stt"
```

```
      "options": [
        {
          "name": "<option name>",
          "value": "<option value>"
        }
      ]
    },
    {
      "service": <service>, // "tts" || "llm || "stt"
      "options": [
        {
          "name": "<option name>",
          "value": "<option value>"
        }
      ]
    }
  ]
}
```

Speech-to-text services

Speech-to-text (STT) services are responsible for transcribing user text, which is then typically passed to the LLM service.

Find available TTS services on the supported services page.

STT configuration

The main considerations for STT services are the model and language parameters. The default language is set to English, but you can modify this by setting the language value. Commonly the model and language parameters have dependencies, so be sure to select model and language values that are supported by the service you are using.

Each STT service has its own set of configuration options. Visit the STT service's page for a full list of available configuration options.

Text-to-speech services

Text-to-speech (TTS) services are responsible for converting the bot's text responses into speech.

Find available TTS services on the supported services page.

TTS configuration

The main consideration for TTS services is the voice option. Depending on the service you are using, you may also have an option for model. The values for voice and model are defined by the service you are using.

For additional configuration options like speed, emotion, stability, and more, visit the TTS service's page for a full list of available configuration options.

LLM services

LLM services are responsible for generating text responses based on the user's input. The options and format of those options are generally defined by the service and model you are using. Each option expects a "name" and "value" field, allowing the bot to dynamically apply the option to any llm and model.

Find available LLM services on the supported services page.

Core functionality

For Daily Bots, there are a few options that are required or commonly used detailed below:

model
string
required
The model you want to use for the LLM service. This is a required field. When using an integrated service without an API key, the model string must match one of the options outlined in the supported services page.

Example:

```
{
  "name": "model",
  "value": "claude-3-5-sonnet-latest"
}
```

initial_messages
Array[LLM messages]
required
The initial set of messages to prompt the LLM service with. This is a required field when providing the first configuration. For subsequent configuration updates, use the "messages" option. The format of the messages is defined by the service and model you are using, but generally contains setting a role and content.

Examples:

Anthropic

Together AI / OpenAI / Groq

```
{
  "name": "initial_messages",
  "value": [{
    "role": "user",
    "content": [{
      "type": "text",
      "text": "You are a pirate."
    }]
  }]
}
```

Anthropic Messages API
Full reference documentation for Anthropic's messages API.

Together AI Messages API
Full reference documentation for Together's message API.

Groq Messages API
Full reference documentation for Groq's message API.

OpenAI Messages API
Full reference documentation for OpenAI's message API.

messages
Array[LLM messages]
required
Same as initial_messages, but used for subsequent configuration updates. This is a required field when providing a configuration update. For the first configuration, use the "initial_messages" option.

run_on_config
bool
default: "false"
run_on_config is a boolean field that forces the bot to talk first. Without this setting, the bot will not begin speaking until the user does. This is an optional field and defaults to false.

IMPORTANT: This field typically should be listed last in your configuration to ensure the bot does not start speaking before it receives its initial messages. Otherwise, fun bot hallucinations may occur.

enable_prompt_caching
bool
Currently only works with Anthropic Claude 3.5 Sonnet and Claude 3 Haiku

Setting this field to true will enable Anthropic's prompt caching feature. This feature allows the bot to remember the last prompt it received and use it as a starting point for the next prompt. This is an optional field and defaults to false.

tools
Array[Tool Definition]
Currently only works with Anthropic and OpenAI

This field describes to the LLM all the tools it has access to and how to call them. This feature is also referred to as "function calling". The format for describing each tool is highly dependent on the service, but typically require you to give your tool a name, a description and an object detailing the set of parameters your tool expects to receive. For more information on setting up tool calling, see our full tutorial. This is an optional field and defaults to an empty array.

Examples:

Anthropic

OpenAI


```
{
  "name": "tools",
  "value": [
    {
      "name": "get_weather",
      "description":
        "Get the current weather for a location. This includes the conditions as well as the
temperature.",
      "input_schema": {
        "type": "object",
        "properties": {
          "location": {
            "type": "string",
            "description":
              "The user's location in the form 'city,state,country'. For example, if the user is in Austin,
TX, use 'austin,tx,us'.",
          },
          "format": {
            "type": "string",
```

```
            "enum": ["celsius", "fahrenheit"],
            "description":
              "The temperature unit to use. Infer this from the user's location.",
          },
        },
        "required": ["location", "format"],
      },
    },
  ],
}
```

Anthropic Tools API
Full reference documentation for Anthropic's tools use API.

OpenAI Function Calling API
Full reference documentation for OpenAI's function calling API.

Configuration options
Each LLM service has its own set of configuration options for parameters like temperature, max_tokens, top_p, and more. Visit the LLM service's page for a full list of available configuration options.

RTVIClient
Bot Profiles
x
youtube
linkedin

Cartesia

Search...
/
Homepage
Discord
Playground

Get Started
Overview
Make an API request
Use an SDK
JavaScript/TypeScript
Python
Build with Sonic

Learn how to control the speed and emotion of generated speech.

Speed and emotion controls are available through the playground and the API in the Text to Speech endpoints (Bytes, SSE, and WebSocket).

The effects of controls vary by voice and transcript. If you find that the controls cause artifacts in the generated speech, try reducing their strength or reducing the number of controls you have applied.

Playground

In the playground, you can access speed and emotion controls by clicking the “Speed/Emotion” button in the Text to Speech tab.

API
This feature is currently experimental and is subject to breaking changes.

To use controls in the API, add the __experimental_controls dictionary to the voice object in your API request:

```
"voice": {
  "mode": "id",
  "id": "VOICE_ID",
  "__experimental_controls": {
    "speed": "normal",
    "emotion": [
      "positivity:high",
      "curiosity"
    ]
  }
}
```

Speed Options
"slowest": Very slow speech
"slow": Slower than normal speech
"normal": Default speech rate
"fast": Faster than normal speech
"fastest": Very fast speech
For more granular control, you can define speed as a number within the range
[
−
1.0
,
1.0
]
[−1.0,1.0]. A value of 0 represents the default speed, while negative values slow down the speech and positive values speed it up.

Using a label
Using a number

```
"__experimental_controls": {
  "speed": "fast"
}
```

Emotion Options

The emotion parameter is an array of “tags” in the form emotion_name:level. For example, positivity:high or curiosity.

Emotion Names
anger
positivity
surprise
sadness
curiosity
Emotion Levels
Emotion controls are purely additive, they cannot reduce or remove emotions. For example, anger:low will add a small amount of anger to the voice, not make the voice less angry.

lowest
low
(omit level for moderate addition of the emotion)
high
highest
Was this page helpful?

Yes

No
Edit this page
Stream Inputs using Continuations
Up Next
Built with


```
 "bot_profile": "voice_2024_10",
  "max_duration": 300,
  "service_options": {
    "deepgram": {
      "model": "nova-2-general",
      "language": "en"
    },
    "anthropic": {
      "model": "claude-3-5-sonnet-20241022"
    }
  },
  "services": {
    "stt": "deepgram",
    "tts": "cartesia",
    "llm": "anthropic"
  },
```

```
"config": [
  {
    "service": "vad",
    "options": [
      {
        "name": "params",
        "value": {
          "stop_secs": 2
        }
      }
    ]
  },
  {
    "service": "tts",
    "options": [
      {
        "name": "voice",
        "value": YOUR_VOICE_ID
      },
      {
        "name": "model",
        "value": "sonic-english"
      }
    ]
  },
  {
    "service": "llm",
    "options": [
      {
        "name": "temperature",
        "value": 0
      },
      {
        "name": "top_p",
        "value": 0
      },
      {
        "name": "initial_messages",
        "value": [
          {
            "role": "user",
            "content": [
              {
                "type": "text",
```

```
                "text": "You are an assistant called Daily Bot. You can ask me anything. Keep responses brief and legible. Start by briefly introducing yourself.\n\nYour responses will be converted to audio. Please do not include any special characters in your response other than '!' or '?'."
              }
            ]
          }
        ]
      },
      {
        "name": "run_on_config",
        "value": true
      }
    ]
  }
 ],
 "api_keys": {
   "cartesia": YOUR_CARTESIA_API_KEY
 }
}
```


Webhooks

Building server-side functionality for your bots with webhooks

If you're building a bot that's primarily designed to be used with incoming or outgoing phone calls, you won't have an RTVI front-end to handle things like function calls. To give you more control over what happens in those sessions, Daily Bots allows you to specify webhook URLs for handling LLM function calls.

To learn more about how function calling works in Daily Bots, see this tutorial page.

Function call webhooks work in two different ways. If you want the same result functionality you get from handleFunctionCall, you can return a JSON response from your webhook. If you want more control over the execution of your bot, you can provide RTVI actions in a streaming response.

Configuring function call webhooks

You can configure a function call webhook in your `/bots/start` request using the `webhook_tools` like this:

```
const payload = {
  bot_profile: "voice_2024_10",
  max_duration: 600,
  services,
  api_keys: {
    openai: process.env.OPENAI_API_KEY,
  },
  config: [...config],
  webhook_tools: {
    get_weather: {
      url: "http://127.0.0.1:8000/weather",
      streaming: true,
    },
    get_local_time: {
      url: "http://127.0.0.1:8000/localtime",
      method: "GET",
    },
  },
};
```

You can set the following properties in each `webhook_tools` object:

url: The URL of your webhook.
method: Defaults to `POST`, but you can set it to `GET` or other methods if you want.
custom_headers: An object with key-value pairs for any custom headers to add to a webhook request (such as authentication).
streaming: Determines whether the bot should expect a simple JSON response (`false`) or a streaming response with RTVI action data (`true`). Defaults to `false`. More on this below.

JSON responses

By default (with `streaming` set to `false`), function call webhooks act just like handleFunctionCall callbacks. When the LLM decides to call a function, the bot will make a request to your webhook URL with the function arguments from the LLM. Your webhook returns a JSON object, and that object will be used as the function call result data.

Here's an example of what a non-streaming webhook handler looks like in FastAPI:

```
@app.get("/localtime")
async def population(req: FunctionCallRequest):
    tz = pytz.timezone("America/Los_Angeles")
    time = datetime.now(tz)
    return {"time": time}
```

Streaming responses

If you want more flexibility in how your function calls are handled, your webhook can stream server-sent events (SSE) containing RTVI messages (actions and config updates) back to the bot. Your SSE text stream should look like this:

```
event: action
data:{RTVI action data as JSON}

event: action
data:{RTVI action data as JSON}

event: update-config
data:{RTVI updateConfig() data as JSON}

event: close
```

Each event needs to begin with the `event:` property, followed by the type of message you're sending, such as `action`, `update-config`, or `close`. Then on the next line, send `data:` followed by JSON data. Finally,end with two newline characters (`\n\n`) to create a blank line. The last message should be `event: close` to ensure that the bot closes the connection.

You can also get the JSON response behavior by sending an `llm:function_result` action, as shown below.

Here's an example FastAPI streaming response. This code sends a config update to change the TTS language, followed by an `llm:function_result` action.

```
async def language_changer(function_name, tool_call_id, arguments):
lang = languages[arguments["language"]]
events = [
    {
        "update-config": {
            "config": [
                {
                    "service": "tts",
                    "options": [
                        {"name": "voice", "value": lang["default_voice"]},
                        {"name": "model", "value": lang["tts_model"]},
                        {"name": "language", "value": lang["value"]},
                    ],
                },
            ]
        }
    },
    {
        "action": {
            "service": "llm",
            "action": "function_result",
            "arguments": [
                {"name": "function_name", "value": function_name},
                {"name": "tool_call_id", "value": tool_call_id},
                {"name": "arguments", "value": arguments},
                {
```

```
                    "name": "result",
                    "value": {"language": arguments["language"]},
                },
            ],
        }
    },
]
for e in events:
    for k, v in e.items():
        yield f"event: {k}\ndata: {json.dumps(v)}\n\n"
yield "data:close\n\n"

@app.post("/language")
async def set_language(req: FunctionCallRequest):
    print("Language request received: req")
    return StreamingResponse(
        language_changer(req.function_name, req.tool_call_id, req.arguments),
        media_type="text/event-stream",
    )
```

Unfortunately, there isn't a great way to get information *back* from your bot when you send these messages. For example, if you send an invalid object for an `update-config` message, the config change will just silently fail. You'll need to look at your Daily Bots dashboard log for debugging.

You can use any of the documented RTVI actions in this way, but be careful if you're modifying the bot context while also using function results!

API Keys

Actions

Actions are service-specific messages that are dispatched to the bot in order to trigger certain pipeline behaviour.

Some examples of actions include:

`tts:say` - Speak a message using text-to-speech.
`tts:interrupt` - Interrupt the current text-to-speech message.

`llm:append_to_messages` - Append a context to the current LLM messages.

Under the hood, actions are blobs of data that are sent via the transport layer to the bot. The bot listens for and then processes the action to perform the necessary operations.

RTVI-enabled bots will have defined actions available for the user to invoke in their client.

Actions do not trigger events or callbacks in the client, but instead return a promise that resolves once the bot has processed the action. Actions are useful ways of extending the functionality of an RTVI bot without needing to modify the client.

Actions differ from messages in that a) they are service specific b) they do not trigger callbacks or events and c) they return a promise that resolves once the bot has processed the action.

Obtaining available actions

To obtain a list of available actions, you can use the `describeActions` method on the RTVI client.

```
const actions = await voiceClient.describeActions();
```

This will return an array of action objects that you can use to determine which actions are available.

```
{
  "label": "rtvi-ai",
  "type": "actions-available",
  "id": "UNIQUE_ID_FROM_REQUEST",
  "data": {
    "actions": [
      {
```

```
            "service": "tts",
            "action": "say",
            "arguments": [
                { "name": "text", "type": "string" },
                { "name": "interrupt", "type": "bool" },
                ...
            ]
        },
        ...
    ]
  }
}
```

Anatomy of an action

An action object has the following properties:

- `service` - The service that the action belongs to.
- `action` - The name of the action, as defined by the bot.
- `arguments` - An array of argument objects that the action accepts.

When the client dispatches an action, it will pass the action name and arguments to the bot, alongside a unique ID that is referenced on response to resolve the promise.

RTVI voice clients maintain a queue of actions that are dispatched to the bot. The action is sent as JSON data via the transport layer to the bot, which processes the action.

Once the action has been processed the bot will send a response message with the same unique ID that the client uses to reference the action queue to resolve the promise.

Dispatching an action

```
const someAction = await voiceClient.action({
  service: "tts",
  action: "say",
  arguments: [
    { name: "text", value: "Hello, world!" },
    { name: "interrupt", value: false },
  ],
});

// > Promise<VoiceMessageActionResponse>
```

An action is resolved or rejected with the following:

```
{
  "label": "rtvi-ai",
  "type": "action-response",
  "id": "UNIQUE_ID_FROM_REQUEST",
  "data": {
    "result": BOOL | NUMBER | STRING | ARRAY | OBJECT
  }
}
```

If an action is unable to be processed, the bot will return a `error-response` typed message with the same unique ID assigned by the client, triggering a rejection.

You can handle error responses in multiple ways:

- `onMessageError` callback
- `MessageError` event
- try / catching the promise

```
try {
      const someAction = await voiceClient.action({...});
} catch(e) {
      console.error(e);
}
```

Action response data

Some actions resolve with data. This data is specific to the action and is defined by the bot.

A successful action will resolve with `VoiceMessageActionResponse`:

```
{
  "label": "rtvi-ai",
  "type": "action-response",
  "id": "UNIQUE_ID_FROM_REQUEST",
  "data": {
      "result": "Hello, world!"
  }
}
```

Awaiting an action will return the `VoiceMessageActionResponse` object, which contains the `result` property.

```
try {
      const someAction = await voiceClient.action({...});
      console.log(someAction.data.result);
} catch(e) {
      console.error(e);
}
```

github