Services

Selecting and authenticating bot services

Bots built with RTVI define and register various services as part of their pipeline. For example, a typical voice bot will likely use Language Models (LLM), text-to-speech, and speech-to-text services to provide voice-to-voice interactions.

RTVI clients allow you to specify which of the available services to use in a session and how to configure them.

Understanding services

Your RTVI client can configure any services registered in an RTVI-powered bot using the configuration options it makes available.

Service registration within your bot might look something like this pseudocode:

```
def main(services:dict, api_keys:dict):
    rtvi_llm = RTVIService(
        name="llm",
        options=[
            RTVIServiceOption(
                name="model",
                type="string",
                handler=config_llm_model_handler),
            RTVIServiceOption(
                name="messages",
                type="array",
                handler=config_llm_messages_handler)])

    rtvi = RTVIProcessor(config=config)
    rtvi.register_service(rtvi_llm)
    def service_factory_get(service: str, name: str, api_key: str) -> Any:
        match service:
            case "openai":
                return OpenAILLMService(
                    name=name,
                    api_key=api_key)
            case "together":
                return OpenAILLMService(
                    name=name,
                    api_key=api_key,
                    base_url="https://api.together.xyz"
                )


llm = service_factory_get(services["llm"], "llm", api_keys[api_keys["llm"]]
or "")

# ... pipeline code
```

The above bot file defines a service named `llm` with two config options, `model` and `messages`, as well as their associated handlers.

Bots can define one or more services to a particular function. For example, we may want to run a pipeline that can be switched between different LLM providers e.g. OpenAI, Together, Anthropic etc.

Building a client requires knowledge of the services that have been registered and the corresponding names. This information is necessary to pass the appropriate configuration and API keys as string-matched values.

Names and providers

In the above example, we have a factory method that returns the relevant provider class for a specific service based on name string.

> Service Name - An arbitrary string that references the service in your bot file, e.g. `"llm"`
> Service Provider - A provider-specific implementation that gets included in the pipeline, e.g. `OpenAILLMService`

Selecting between services on the client

RTVI bots can be passed an optional `services` object at startup that can be used to specify which provider to use for the specified service name.

In the above above example, we can configure a voice client to use Together like so:

```
const voiceClient = new VoiceClient({
  services: {
    llm: "together",
  },
  config: [
    {
      service: "llm",
      options: [
        { name: "model", value: "meta-llama/Meta-Llama-3.1-8B-Instruct-Turbo"
},
      ],
    },
  ],
  // etc ...
});
```

Passing API keys

Service keys are secret, you should not set them on a client. To pass API keys to your RTVI bot, host a secure server-side endpoint that includes them as part of the config payload.

RTVI bots accept an `api_keys` object, mapping them to the relevant service account. Here is an example server-side route using NextJS:

```
// api/
export async function POST(request: Request) {
  const { services, config } = await request.json();

  if (!services || !config || !process.env.DAILY_BOTS_URL) {
    return new Response(`Services or config not found on request body`, {
      status: 400,
    });
  }

  const payload = {
    services,
    api_keys: {
      together: process.env.TOGETHER_API_KEY,
    },
    config: [...config],
  };

  const req = await fetch("your-bot-start-endpoint", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify(payload),
  });

  const res = await req.json();

  return Response.json(res);
}
```

You can also extend your config object here, if you wanted to provide some defaults outside of the client constructor.

In the above example, we'd set the baseUrl property of the voice client to point to this endpoint, and define which service's to use in the bot's registry like so:

```
const voiceClient = new DailyVoiceClient({
  baseUrl: "/api",
  services: {
    llm: "together",
  },
  config: [
    {
      service: "llm",
      options: [
        { name: "model", value: "meta-llama/Meta-Llama-3.1-8B-Instruct-Turbo"
},
      ],
    },
  ],
});
```

`config` references the service name that it was registered with in your bot file.
`services` specifies which provider / service account to use for a specific name-matched service in the registry (in this case, `llm`.)
`api_keys` provides a key matched to the service account.

Client Methods
Configuration
github

Configuration

Passing service configuration values to a bot.
RTVI bots expose services and service configuration options to clients.

Your client config can be set when initializing your bot or at runtime.

A typical bot config, in JSON, might look like this:

```
[
  { service: "vad", options: [{ name: "params", "value": {stop_secs: 3.0} }]
},
  {
    service: "tts",
    options: [{ name: "voice", value: "79a125e8-cd45-4c13-8a67-188112f4dd22"
}],
  },
  {
    service: "llm",
    options: [
      { name: "model", value: "meta-llama/Meta-Llama-3.1-8B-Instruct-Turbo"
},
      {
        name: "initial_messages",
        value: [
          {
            role: "system",
            content: `You are a assistant called ExampleBot. You can ask me
anything.
              Keep responses brief and legible.
              Your responses will converted to audio. Please do not include
any special characters in your response other than '!' or '?'.
              Start by briefly introducing yourself.`,
          },
        ],
      },
      { name: "run_on_config", value: true },
    ],
  },
]
```

Client-side configuration

You can pass a config into the `VoiceClient` constructor. Passing a config from the client is optional. A bot will always start with a default config if no config is passed from the client. Some

RTVI implementations may also choose to ignore configs passed from the client, for security or other reasons.

Getting the bot config

getBotConfig()

Returns a Promise that resolves with the bot's current configuration.

```
config = voiceClient.getBotConfig()
```

Updating the bot config

updateConfig()

Update the bot's config. Passing a partial config is fine. Only the individual options that are passed to the `updateConfig()` call will be updated. You can omit any services you don't want to make changes to. You can omit any options within a service that you don't want to change.

Returns a Promise that resolves with the full updated configuration.

`config`

`VoiceClientConfigOption[]`

VoiceClientConfigOption[] partial object with the new configuration

```
new_config = voiceClient.updateConfig([
  { service: "vad", options:[{stop_secs: 0.5}] }
])
```

Retrieving available config options

describeConfig()

Get the available config options for the bot the client is connected to.

Returns a Promise that resolve’s with the bot’s configuration description.

```
configuration_metadescription = await voiceClient.describeConfig()
```

Server-side configuration

Platforms implementing RTVI on the server side will generally provide a method for passing a config into a “start” method. It’s a good practice to use the same config format for both client-side and server-side configuration, though of course this choice is left up to the implementor of the server-side APIs.

Setting service API keys

It's important to note that API keys should never be included in configuration messages from or to clients. Clients shouldn't have access to API keys at all.

Platforms implementing RTVI should use a separate mechanism for passing API keys to a bot. A typical approach is to start a bot with a larger, "meta config" that includes API keys, a list of services the bot should instantiate, the client-visible bot configuration, and perhaps other fields like the maximum session duration.

For example:

```
const bot_start_rest_api_payload = {
  api_keys: api_keys_map_for_env
  bot_profile: "a_bot_version_and_capabilities_string",
  max_duration: duration_in_seconds
  services: [{ llm: "together", tts: "cartesia" }]
  config: config_passed_from_client
};
```

Services
Callbacks and events
github

Callbacks and events

A voice client exposes callback hooks that can be defined on the constructor.

Callbacks can be configured as part of your voice client constructor:

```
const voiceClient = new VoiceClient({
  // ...
  callbacks: {
    onBotReady: bot_ready_handler_func,
  },
});
```

State and connectivity

`onTransportStateChanged`

`state:TransportState`

Provides a `TransportState` string representing the connectivity state of the local client.

One of: `idle` | `initializing` | `initialized` | `authenticating` | `connecting` | `connected` | `ready` | `disconnected` | `error`

See transports for state explanation.

`onConnected`

Local user successfully established a connection to the transport.

`onDisconnected`

Local user disconnected from the transport, either intentionally by calling `voiceClient.disconnect()` or due to an error.

`onBotConnected`

Bot connected to the transport and is configuring.

Note: bot connectivity does not infer that its pipeline is yet ready to run. Please use `onBotReady` instead.

`onBotReady`

`botReadyData:BotReadyData`

The bot has been instantiated, its pipeline is configured, and it is receiving user media and interactions.

This method is passed a `BotReadyData` object, which contains a `config` `VoiceClientConfigOption[]` array and the RTVI `version` number.

It is recommended to hydrate your client with the passed config in case the bot alters any configuration based on its implementation.

Since the bot is remote and may be using a different version of RTVI than the client, you can use the passed `version` string to check for compatibility.

`onBotDisconnected`

Bot disconnected from the transport.

This may occur due to session expiry, a pipeline error or because the local participant left the session.

Messages and errors

`onGenericMessage`

`data:unknown`

If the client receives an unknown message type from the transport (see messages and events), it can be handled here.

`onMessageError`

`message:VoiceMessage`

Response error when an action fails or an unknown message type is sent from the client.

`onError`

`message:VoiceMessage`

Error signalled by the bot. This could be due to a malformed config update or an unknown action dispatch or the ability to complete the requested action.

Configuration

`onConfigUpdated`

`config:VoiceClientConfigOption[]`

Sent when the bots configuration changes. This is most likely in response to a user `updateConfig` action, but can occur within a bots pipeline.

It’s recommended to hydate the client config with the passed config.

`onConfigDescribe`

`configDescription:unknown`

A list of available configuration options for each service. Sent in response to a user `describeConfig()` call.

```
"config": [
  {
      "service": "llm",
      "options": [
            { "name": "model", "type": "string" },
            { "name": "messages", "type": "array" },
            ...
      ]
  },
  {
      "service": "tts",
      "options": [
            { "name": "voice", "type": "string" },
            ...
      ]
  },
  ...
]
```

Media and devices

`onAvailableMicsUpdated`

`mics:MediaDeviceInfo[]`

Lists available local media microphone devices. Triggered when a new device becomes available or in response to `voiceClient.initDevices()`.

`onAvailableCamsUpdated`

`cams:MediaDeviceInfo[]`

Lists available local media camera devices. Triggered when a new device becomes available or in response to `voiceClient.initDevices()`.

`onMicUpdated`

`mic:MediaDeviceInfo`

User selected a new microphone as their selected/active device.

`onCamUpdated`

`cam:MediaDeviceInfo`

User selected a new camera as their selected/active device.

Audio and Voice Activity

`onTrackStarted`

`track: MediaStreamTrack, participant:Participant`

Media track from a local or remote participant was started and playable. Can be either an audio or video track.

`onTrackStopped`

`track: MediaStreamTrack, participant:Participant`

Media track from a local or remote participant was stopped and no longer playable.

`onLocalAudioLevel`

`level:number`

Local audio gain level (0 to 1).

`onRemoteAudioLevel`

`level: number, participant: Participant`

Remote audio gain level (0 to 1).

Note: if more than one participant is connected to the transport, the `participant` property details the associated peer.

`onBotStartedSpeaking`

The bot started speaking/sending speech audio.

`onBotStoppedSpeaking`

The bot stopped speaking/sending speech audio.

`onUserStartedSpeaking`

The local user started speaking.

This method is more reliable than using audio gain and is the result of the bot's VAD (voice activity detection) model. This provides a more accurate result in noisy environments.

`onUserStoppedSpeaking`

The local user stopped speaking, indicated by the VAD model.

Transcription

`onUserStoppedSpeaking`

`transcript:Transcription`

Transcribed local user input (both partial and final)

`onBotTranscript`

`transcript:string`

Transcribed bot input. Sentence aggregated.

Other

`onMetrics`

`default: "data:PipecatMetrics"`

Pipeline data provided by Pipecat. Please see Pipecat documentation for more information.

Events

RTVI defines the following standard events that map to messages/actions from the bot to the client.

```
MessageError = "messageError",
Error = "error",

Connected = "connected",
Disconnected = "disconnected",
TransportStateChanged = "transportStateChanged",

ConfigUpdated = "configUpdated",
ConfigDescribe = "configDescribe",
ActionsAvailable = "actionsAvailable",
ParticipantConnected = "participantConnected",
ParticipantLeft = "participantLeft",
TrackStarted = "trackStarted",
TrackedStopped = "trackStopped",

AvailableCamsUpdated = "availableCamsUpdated",
AvailableMicsUpdated = "availableMicsUpdated",
CamUpdated = "camUpdated",
MicUpdated = "micUpdated",

BotConnected = "botConnected",
BotReady = "botReady",
BotDisconnected = "botDisconnected",
BotStartedSpeaking = "botStartedSpeaking",
BotStoppedSpeaking = "botStoppedSpeaking",
RemoteAudioLevel = "remoteAudioLevel",

UserStartedSpeaking = "userStartedSpeaking",
UserStoppedSpeaking = "userStoppedSpeaking",
LocalAudioLevel = "localAudioLevel",

Metrics = "metrics",
UserTranscript = "userTranscript",
BotTranscript = "botTranscript",

LLMFunctionCall = "llmFunctionCall",
LLMFunctionCallStart = "llmFunctionCallStart",
LLMJsonCompletion = "llmJsonCompletion",
```

Handlers for these can be bound on the client like so:

```
import { VoiceEvent } from "realtime-ai";

function handleBotReady() {
  console.log("Bot is ready!");
}

// Bind an event handler
voiceClient.on(VoiceEvent.BotReady, handleBotReady);
// or
voiceClient.on("botReady", handleBotReady);
```

```
// Unbind an event handler
voiceClient.off(VoiceEvent.BotReady, handleBotReady);
```

Configuration
Action
API

Callbacks and events

A voice client exposes callback hooks that can be defined on the constructor.

Callbacks can be configured as part of your voice client constructor:

```
const voiceClient = new VoiceClient({
  // ...
  callbacks: {
    onBotReady: bot_ready_handler_func,
  },
});
```

State and connectivity

`onTransportStateChanged`

`state:TransportState`

Provides a `TransportState` string representing the connectivity state of the local client.

One of: `idle` | `initializing` | `initialized` | `authenticating` | `connecting` | `connected` | `ready` | `disconnected` | `error`

See transports for state explanation.

`onConnected`

Local user successfully established a connection to the transport.

`onDisconnected`

Local user disconnected from the transport, either intentionally by calling `voiceClient.disconnect()` or due to an error.

`onBotConnected`

Bot connected to the transport and is configuring.

Note: bot connectivity does not infer that its pipeline is yet ready to run. Please use `onBotReady` instead.

`onBotReady`

`botReadyData:BotReadyData`

The bot has been instantiated, its pipeline is configured, and it is receiving user media and interactions.

This method is passed a `BotReadyData` object, which contains a `config` `VoiceClientConfigOption[]` array and the RTVI `version` number.

It is recommended to hydrate your client with the passed config in case the bot alters any configuration based on its implementation.

Since the bot is remote and may be using a different version of RTVI than the client, you can use the passed `version` string to check for compatibility.

`onBotDisconnected`

Bot disconnected from the transport.

This may occur due to session expiry, a pipeline error or because the local participant left the session.

Messages and errors

`onGenericMessage`

`data:unknown`

If the client receives an unknown message type from the transport (see messages and events), it can be handled here.

`onMessageError`

`message:VoiceMessage`

Response error when an action fails or an unknown message type is sent from the client.

`onError`

`message:VoiceMessage`

Error signalled by the bot. This could be due to a malformed config update or an unknown action dispatch or the ability to complete the requested action.

Configuration

`onConfigUpdated`

`config:VoiceClientConfigOption[]`

Sent when the bots configuration changes. This is most likely in response to a user `updateConfig` action, but can occur within a bots pipeline.

It's recommended to hydate the client config with the passed config.

`onConfigDescribe`

`configDescription:unknown`

A list of available configuration options for each service. Sent in response to a user `describeConfig()` call.

```
"config": [
  {
      "service": "llm",
      "options": [
            { "name": "model", "type": "string" },
            { "name": "messages", "type": "array" },
            ...
      ]
  },
  {
      "service": "tts",
      "options": [
            { "name": "voice", "type": "string" },
            ...
      ]
  },
  ...
]
```

Media and devices

`onAvailableMicsUpdated`

`mics:MediaDeviceInfo[]`

Lists available local media microphone devices. Triggered when a new device becomes available or in response to `voiceClient.initDevices()`.

`onAvailableCamsUpdated`

`cams:MediaDeviceInfo[]`

Lists available local media camera devices. Triggered when a new device becomes available or in response to `voiceClient.initDevices()`.

`onMicUpdated`

`mic:MediaDeviceInfo`

User selected a new microphone as their selected/active device.

`onCamUpdated`

`cam:MediaDeviceInfo`

User selected a new camera as their selected/active device.

Audio and Voice Activity

`onTrackStarted`

`track: MediaStreamTrack, participant:Participant`

Media track from a local or remote participant was started and playable. Can be either an audio or video track.

`onTrackStopped`

`track: MediaStreamTrack, participant:Participant`

Media track from a local or remote participant was stopped and no longer playable.

`onLocalAudioLevel`

`level:number`

Local audio gain level (0 to 1).

`onRemoteAudioLevel`

`level: number, participant: Participant`

Remote audio gain level (0 to 1).

Note: if more than one participant is connected to the transport, the `participant` property details the associated peer.

`onBotStartedSpeaking`

The bot started speaking/sending speech audio.

`onBotStoppedSpeaking`

The bot stopped speaking/sending speech audio.

`onUserStartedSpeaking`

The local user started speaking.

This method is more reliable than using audio gain and is the result of the bot's VAD (voice activity detection) model. This provides a more accurate result in noisy environments.

`onUserStoppedSpeaking`

The local user stopped speaking, indicated by the VAD model.

Transcription

`onUserStoppedSpeaking`

`transcript:Transcription`

Transcribed local user input (both partial and final)

`onBotTranscript`

`transcript:string`

Transcribed bot input. Sentence aggregated.

Other

`onMetrics`

`default: "data:PipecatMetrics"`

Pipeline data provided by Pipecat. Please see Pipecat documentation for more information.

Events

RTVI defines the following standard events that map to messages/actions from the bot to the client.

```
MessageError = "messageError",
Error = "error",

Connected = "connected",
Disconnected = "disconnected",
TransportStateChanged = "transportStateChanged",

ConfigUpdated = "configUpdated",
ConfigDescribe = "configDescribe",
ActionsAvailable = "actionsAvailable",

ParticipantConnected = "participantConnected",
ParticipantLeft = "participantLeft",
TrackStarted = "trackStarted",
TrackedStopped = "trackStopped",

AvailableCamsUpdated = "availableCamsUpdated",
AvailableMicsUpdated = "availableMicsUpdated",
CamUpdated = "camUpdated",
MicUpdated = "micUpdated",

BotConnected = "botConnected",
BotReady = "botReady",
BotDisconnected = "botDisconnected",
BotStartedSpeaking = "botStartedSpeaking",
BotStoppedSpeaking = "botStoppedSpeaking",
RemoteAudioLevel = "remoteAudioLevel",

UserStartedSpeaking = "userStartedSpeaking",
UserStoppedSpeaking = "userStoppedSpeaking",
LocalAudioLevel = "localAudioLevel",

Metrics = "metrics",
UserTranscript = "userTranscript",
BotTranscript = "botTranscript",

LLMFunctionCall = "llmFunctionCall",
LLMFunctionCallStart = "llmFunctionCallStart",
LLMJsonCompletion = "llmJsonCompletion",
```

Handlers for these can be bound on the client like so:

```
import { VoiceEvent } from "realtime-ai";

function handleBotReady() {
  console.log("Bot is ready!");
}

// Bind an event handler
voiceClient.on(VoiceEvent.BotReady, handleBotReady);
// or
voiceClient.on("botReady", handleBotReady);

// Unbind an event handler
voiceClient.off(VoiceEvent.BotReady, handleBotReady);
```

Configuration