Develop an Azure AI Voice Live voice agent

In this exercise, you complete a Flask-based Python web app based that enables real-time voice interactions with an agent. You add the code to initialize the session, and handle session events. You use a deployment script that: deploys the AI model; creates an image of the app in Azure Container Registry (ACR) using ACR tasks; and then creates an Azure App Service instance that pulls the the image. To test the app you will need an audio device with microphone and speaker capabilities.

While this exercise is based on Python, you can develop similar applications other language-specific SDKs; including:

- Azure VoiceLive client library for .NET

Tasks performed in this exercise:

- Download the base files for the app
- Add code to complete the web app
- Review the overall code base
- Update and run the deployment script
- View and test the application

This exercise takes approximately **30** minutes to complete.

Launch the Azure Cloud Shell and download the files

In this section of the exercise you download the a zipped file containing the base files for the app.

1. In your browser navigate to the Azure portal https://portal.azure.com; signing in with your Azure credentials if prompted.

2. Use the **[>_]** button to the right of the search bar at the top of the page to create a new cloud shell in the Azure portal, selecting a ***Bash*** environment. The cloud shell provides a command line interface in a pane at the bottom of the Azure portal.

   > **Note**: If you have previously created a cloud shell that uses a *PowerShell* environment, switch it to ***Bash***.

3. In the cloud shell toolbar, in the **Settings** menu, select **Go to Classic version** (this is required to use the code editor).

4. Run the following command in the **Bash** shell to download and unzip the exercise files. The second command will also change to the directory for the exercise files.

Code Copy

```
wget https://github.com/MicrosoftLearning/mslearn-ai-language/raw/refs/heads/main/downloads/python/voice-live-web.zip
```

Code Copy

```
unzip voice-live-web.zip && cd voice-live-web
```

Add code to complete the web app

Now that the exercise files are downloaded, the the next step is to add code to complete the application. The following steps are performed in the cloud shell.

> **Tip:** Resize the cloud shell to display more information, and code, by dragging the top border. You can also use the minimize and maximize buttons to switch between the cloud shell and the main portal interface.

Run the following command to change into the *src* directory before you continue with the exercise.

Code Copy

```
cd src
```

Add code to implement the voice live assistant

In this section you add code to implement the voice live assistant. The **__init__** method initializes the voice assistant by storing the Azure VoiceLive connection parameters (endpoint, credentials, model, voice, and system instructions) and setting up runtime state variables to manage the connection lifecycle and handle user interruptions during conversations. The **start** method imports the necessary Azure VoiceLive SDK components that will be used to establish the WebSocket connection and configure the real-time voice session.

1. Run the following command to open the *flask_app.py* file for editing.

   Code Copy

   ```
   code flask_app.py
   ```

2. Search for the **# BEGIN VOICE LIVE ASSISTANT IMPLEMENTATION - ALIGN CODE WITH COMMENT** comment in the code. Copy the code below and enter it just below the comment. Be sure to check the indentation.

   Code Copy

```
def __init__(
    self,
    endpoint: str,
    credential,
    model: str,
    voice: str,
    instructions: str,
    state_callback=None,
):
    # Store Azure Voice Live connection and configuration parameters
    self.endpoint = endpoint
    self.credential = credential
    self.model = model
    self.voice = voice
    self.instructions = instructions

    # Initialize runtime state - connection established in start()
    self.connection = None
    self._response_cancelled = False  # Used to handle user interruptions
    self._stopping = False  # Signals graceful shutdown
    self.state_callback = state_callback or (lambda *_: None)

async def start(self):
    # Import Voice Live SDK components needed for establishing connection and configuring
session
    from azure.ai.voicelive.aio import connect  # type: ignore
    from azure.ai.voicelive.models import (
        RequestSession,
        ServerVad,
        AzureStandardVoice,
        Modality,
        InputAudioFormat,
        OutputAudioFormat,
    )  # type: ignore
```

3. Enter **ctrl+s** to save your changes and keep the editor open for the next section.

Add code to implement the voice live assistant

In this section you add code to configure the voice live session. This specifies the modalities (audio-only is not supported by the API), the system instructions that define the assistant's behavior, the Azure TTS voice for responses, the audio format for both input and output streams, and Server-side Voice Activity Detection (VAD) which specifies how the model detects when users start and stop speaking.

1. Search for the **# BEGIN CONFIGURE VOICE LIVE SESSION - ALIGN CODE WITH COMMENT** comment in the code. Copy the code below and enter it just below the comment. Be sure to check the indentation.

Code Copy

```
# Configure VoiceLive session with audio/text modalities and voice activity detection
session_config = RequestSession(
    modalities=[Modality.TEXT, Modality.AUDIO],
    instructions=self.instructions,
    voice=voice_cfg,
    input_audio_format=InputAudioFormat.PCM16,
    output_audio_format=OutputAudioFormat.PCM16,
    turn_detection=ServerVad(threshold=0.5, prefix_padding_ms=300, silence_duration_ms=500),
)
await conn.session.update(session=session_config)
```

2. Enter **ctrl+s** to save your changes and keep the editor open for the next section.

Add code to handle session events

In this section you add code to add event handlers for the voice live session. The event handlers respond to key VoiceLive session events during the conversation lifecycle: **_handle_session_updated** signals when the session is ready for user input, **_handle_speech_started** detects when the user begins speaking and implements interruption logic by stopping any ongoing assistant audio playback and canceling in-progress responses to allow natural conversation flow, and **_handle_speech_stopped** handles when the user has finished speaking and the assistant begins processing the input.

1. Search for the **# BEGIN HANDLE SESSION EVENTS - ALIGN CODE WITH COMMENT** comment in the code. Copy the code below and enter it just below the comment, be sure to check the indentation.

Code Copy

```
async def _handle_event(self, event, conn, verbose=False):
    """Handle Voice Live events with clear separation by event type."""
    # Import event types for processing different Voice Live server events
    from azure.ai.voicelive.models import ServerEventType

    event_type = event.type
    if verbose:
        _broadcast({"type": "log", "level": "debug", "event_type": str(event_type)})

    # Route Voice Live server events to appropriate handlers
    if event_type == ServerEventType.SESSION_UPDATED:
        await self._handle_session_updated()
    elif event_type == ServerEventType.INPUT_AUDIO_BUFFER_SPEECH_STARTED:
        await self._handle_speech_started(conn)
    elif event_type == ServerEventType.INPUT_AUDIO_BUFFER_SPEECH_STOPPED:
        await self._handle_speech_stopped()
    elif event_type == ServerEventType.RESPONSE_AUDIO_DELTA:
        await self._handle_audio_delta(event)
    elif event_type == ServerEventType.RESPONSE_AUDIO_DONE:
        await self._handle_audio_done()
    elif event_type == ServerEventType.RESPONSE_DONE:
        # Reset cancellation flag but don't change state - _handle_audio_done already did
        self._response_cancelled = False
    elif event_type == ServerEventType.ERROR:
        await self._handle_error(event)

async def _handle_session_updated(self):
    """Session is ready for conversation."""
    self.state_callback("ready", "Session ready. You can start speaking now.")

async def _handle_speech_started(self, conn):
    """User started speaking - handle interruption if needed."""
    self.state_callback("listening", "Listening… speak now")

    try:
        # Stop any ongoing audio playback on the client side
        _broadcast({"type": "control", "action": "stop_playback"})

        # If assistant is currently speaking or processing, cancel the response to allow
interruption
        current_state = assistant_state.get("state")
        if current_state in {"assistant_speaking", "processing"}:
            self._response_cancelled = True
            await conn.response.cancel()
            _broadcast({"type": "log", "level": "debug",
                        "msg": f"Interrupted assistant during {current_state}"})
        else:
            _broadcast({"type": "log", "level": "debug",
                        "msg": f"User speaking during {current_state} - no cancellation
needed"})
    except Exception as e:
        _broadcast({"type": "log", "level": "debug",
                    "msg": f"Exception in speech handler: {e}"})

async def _handle_speech_stopped(self):
    """User stopped speaking - processing input."""
    self.state_callback("processing", "Processing your input…")
```

```
async def _handle_audio_delta(self, event):
    """Stream assistant audio to clients."""
    if self._response_cancelled:
        return  # Skip cancelled responses

    # Update state when assistant starts speaking
    if assistant_state.get("state") != "assistant_speaking":
        self.state_callback("assistant_speaking", "Assistant speaking…")

    # Extract and broadcast Voice Live audio delta as base64 to WebSocket clients
    audio_data = getattr(event, "delta", None)
    if audio_data:
        audio_b64 = base64.b64encode(audio_data).decode("utf-8")
        _broadcast({"type": "audio", "audio": audio_b64})

async def _handle_audio_done(self):
    """Assistant finished speaking."""
    self._response_cancelled = False
    self.state_callback("ready", "Assistant finished. You can speak again.")

async def _handle_error(self, event):
    """Handle Voice Live errors."""
    error = getattr(event, "error", None)
    message = getattr(error, "message", "Unknown error") if error else "Unknown error"
    self.state_callback("error", f"Error: {message}")

def request_stop(self):
    self._stopping = True
```

2. Enter **ctrl+s** to save your changes and keep the editor open for the next section.

Review the code in the app

So far, you've added code to the app to implement the agent and handle agent events. Take a few minutes to review the full code and comments to get a better understanding of how the app is handling client state and operations.

1. When you're finished enter **ctrl+q** to exit out of the editor.

Update and run the deployment script

In this section you make a small change to the **azdeploy.sh** deployment script and then run the deployment.

Update the deployment script

There are only two values you should change at the top of the **azdeploy.sh** deployment script.

- The **rg** value specifies the resource group to contain the deployment. You can accept the default value, or enter your own value if you need to deploy to a specific resource group.
- The **location** value sets the region for the deployment. The *gpt-4o* model used in the exercise can be deployed to other regions, but there can be limits in any particular region. If the deployment fails in your chosen region, try **eastus2** or **swedencentral**.

Code Copy

```
rg="rg-voicelive" # Replace with your resource group
location="eastus2" # Or a location near you
```

1. Run the following commands in the Cloud Shell to begin editing the deployment script.

```
cd ~/voice-live-web
```

```
code azdeploy.sh
```

2. Update the values for **rg** and **location** to meet your needs and then enter **ctrl+s** to save your changes and **ctrl+q** to exit the editor.

Run the deployment script

The deployment script deploys the AI model and creates the necessary resources in Azure to run a containerized app in App Service.

1. Run the following command in the Cloud Shell to begin deploying the Azure resources and the application.

```
bash azdeploy.sh
```

2. Select **option 1** for the initial deployment.

   The deployment should complete in 5-10 minutes. During the deployment you might be prompted for the following information/actions:

   - If you are prompted to authenticate to Azure follow the directions presented to you.
   - If you are prompted to select a subscription use the arrow keys to highlight your subscription and press **Enter**.
   - You will likely see some warnings during deployment and these can be ignored.
   - If the deployment fails during the AI model deployment change the region in the deployment script and try again.
   - Regions in Azure can get busy at times and interrupt the timing of the deployments. If the deployment fails after the model deployment re-run the deployment script.

View and test the app

When the deployment completes a "Deployment complete!" message will be in the shell along with a link to the web app. You can select that link, or navigate to the App Service resource and launch the app from there. It can take a few minutes for the application to load.

1. Select the **Start session** button to connect to the model.
2. You will be prompted to give the application access to you audio devices.
3. Begin talking to the model when the app prompts you to start speaking.

Troubleshooting:

- If the app reports missing environment variables, restart the application in App Service.
- If you see excessive *audio chunk* messages in the log shown in the application select **Stop session** and then start the session again.
- If the app fails to function at all, double-check you added all of the code and for proper indentation. If you need to make any changes re-run the deployment and select **option 2** to only update the image.

Clean up resources

Run the following command in the Cloud Shell to remove all of the resources deployed for this exercise. You will be prompted to confirm the resource deletion.

Code Copy

```
azd down --purge
```