Agents

Integrating agents with my application

Embedding the agent into the web chat

IBM watsonx Orchestrate allows you to embed intelligent agents directly into your web applications using the Embedded Chat feature. This integration supports secure communication, flexible UI customization, and advanced event handling to deliver rich conversational experiences.

Why embed agents?

Embedding agents into your application provides:

- Real-time interaction: Users engage with agents directly within your app.
- Custom UI integration: Match the agent's look and feel to your brand.
- Secure communication: RSA encryption and JWT authentication protect sensitive data.
- Context-aware automation: Agents can access encrypted context variables to personalize responses.
- Scalable deployment: Embed across multiple environments using consistent configuration.

Prerequisites

Environments: draft vs. live

Draft: Used for testing. Embed snippets target the draft variant.

Live: Available after deployment. Embed snippets default to the live variant in production.

Use the CLI command to generate the embed script:

```
orchestrate channels webchat embed --agent-name=test_agent1
```

This outputs a `<script>` tag with configuration for your environment.

When targeting the local environment, the command uses your agent's draft variant. On a production instance, it defaults to using your agent's live (deployed) variant. The following is an example of the command's output:

OUTPUT

```
<script>
  window.wxOConfiguration = {
    orchestrationID: "your-orgID_orchestrationID", // Adds control
    hostURL: "https://dl.watson-orchestrate.ibm.com", // or region
    rootElementID: "root",
    showLauncher: false,
    deploymentPlatform: "ibmcloud", // Required for IBM Cloud embe
    crn: "your-org-crn", // Required for IBM Cloud embed, can be s
    chatOptions: {
      agentId: "your-agent-id",
      agentEnvironmentId: "your-agent-env-id",
    },
    layout: {
      form: "fullscreen-overlay", // Options: float | custom | ful
      showOrchestrateHeader: true, // Optional: shows top agent he
      width: "600px", // Optional: honored when form is float only
      height: "600px", // Optional: honored when form is float onl
    },
  };

  setTimeout(function () {
    const script = document.createElement("script");
    script.src = `${window.wxOConfiguration.hostURL}/wxochat/wxoLo
    script.addEventListener("load", function () {
      wxoLoader.init();
    });
    document.head.appendChild(script);
  }, 0);
</script>
```

··· Collapse

Application requirements

Your application must:

- Include a server (local or cloud).

Have an HTML element with ID `root`.

Place the embed script inside the `<body>` tag.

The page that embeds the web chat must use HTML's strict mode (in clude the `<!DOCTYPE html>` tag).

Example:

```
<!DOCTYPE html>
<body>
  <div id="root"></div>
  <script src="path-to-embed-script.js"></script>
</body>
```

Security configuration

Security is enabled by default, but must be explicitly configured. The embedded chat will not function until security is properly configured

Security Architecture

The embedded chat uses RSA public-key cryptography to secure communication. The configuration involves two key pairs:

1. IBM Key Pair

   Generated by: watsonx Orchestrate service.

   Public key: Shared with your application. Your application uses this key to encrypt the `user_payload` section of the JWT sent to watsonx Orchestrate.

   Private key: Stored securely by watsonx Orchestrate.cUsed to decrypt the `user_payload` section of the JWT.

2. Client Key Pair

   Generated by: You (or a security configuration tool).

Public key: Shared with watsonx Orchestrate. Used to verify that JWTs originate from your application.

Private key: Remains with you and must be stored securely. Used to sign JWTs sent to watsonx Orchestrate.

3. JWT Authentication

   When security is enabled, your application must:

   Generate a JWT signed with your private key (Client Key Pair).

   Include the JWT in all requests to the Embedded Chat API. watsonx Orchestrate validates the token using your public key.

When security is enabled:

All requests to the Embedded Chat API must include a valid JWT token

The token must be signed with your private key

The watsonx Orchestrate service validates the token using your public key

This prevents unauthorized access to your watsonx Orchestrate instance

When security is disabled:

Requests to the Embedded Chat API do not require authentication

Anyone with access to your web application where chat is Embedded can access your watsonx Orchestrate instance. In addition, your Watson Orchestrate instance allows anonymous authentication to a limited set of Apis, which is required to get your embed chat to work for anonymous users.

This option should only be used for specific use cases where anonymous chat access is required.

Ensure your watsonx Orchestrate instance in this case, does not provide access to sensitive data or access to tools configured with functional credentials that access sensitive data.

Enabling security

1 Prerequisites

IBM watsonx Orchestrate instance

API Key with administrative privileges

Service Instance URL from your watsonx Orchestrate instance

On macOS and Linux: OpenSSL installed on your system (for key generation)

Python Installed on your system (for key extraction from APIs)

2 Get the watsonx Orchestrate security tool

Copy the following automated script to configure security:

wxO-embed-chat-security-tool.sh

```
#!/bin/bash
# IBM watsonx Orchestrate - Embedded Chat Security Configurat
# This script works on both Windows (PowerShell) and Unix-bas

# Detect OS and execute appropriate script
if [ -n "$BASH_VERSION" ]; then
    # Running in Bash (Unix/Linux/Mac)
```

··· See all 1392 lines

3 Change the script's permissions

On Unix-based systems (macOS and Linux), change the permissions to run the script:

```
chmod +x wxO-embed-chat-security-tool.sh
```

4 Run the script

Run the script and follow the instructions to enable or disable security:

```
./wxO-embed-chat-security-tool.sh
```

After you configure security:

1. The tool generates an IBM key pair via the API
2. The tool generates a client key pair using OpenSSL
3. Both public keys are configured in the service
4. Security is enabled

All keys are saved in the `wxo_security_config` directory:

`ibm_public_key.pem` : IBM's public key in PEM format

`ibm_public_key.txt` : IBM's public key in single-line format

`client_private_key.pem` : Your private key (keep it secure!)

`client_public_key.pem` : Your public key in PEM format

`client_public_key.txt` : Your public key in single-line format

Anonymous Access

You may disable security for anonymous access, but only if:

No sensitive data is exposed.

No tools with functional credentials are accessible.

Context variables

To use context variables:

1. Enable `context_access_enabled: true` in your agent definition.

2. Add variables like `channel` to the `context_variables` in your agent definition file.
3. Reimport the agent.
4. Include context variables in the JWT payload.

```yaml
spec_version: v1
style: react
name: hello_agent
llm: watsonx/meta-llama/llama-3-1-70b-instruct
description:  'Agent description'
instructions: |
  You are a helpful agent that must answer user questions in a clea
collaborators: []
tools: []
context_access_enabled: true
context_variables:
  - channel     # Use it to get access to context variables on the
```

You can add context variables to a JWT token using a JavaScript script. The following script shows how to include context variables inside a JWT token in your server:

createJWT

```
const fs = require('fs');
const RSA = require('node-rsa');
const crypto = require('crypto');
const jwtLib = require('jsonwebtoken');
const express = require('express');
const path = require('path');
const { v4: uuid } = require('uuid');

const router = express.Router();

// This is your private key that you will keep on your server. Th
// key into the appropriate field on the Security tab of the web
// This public key is used to validate the signature on the jwt.
const PRIVATE_KEY = fs.readFileSync(path.join(__dirname, 'wxo_sec

//The code below will use this key to encrypt the user payload in
const PUBLIC_KEY = fs.readFileSync(path.join(__dirname, 'wxo_secu


// A time period of 45 days in milliseconds.
const TIME_45_DAYS = 1000 * 60 * 60 * 24 * 45;

/**
 * Generates a signed JWT. The JWT used here will always be assig
 * the user is authenticated and we have session info, then info
 * Always use the anonymous user ID even if the user is authentic
 * a session is not allowed.
 */
function createJWTString(anonymousUserID, sessionInfo,context) {
  // This is the content of the JWT. You would normally look up t
  const jwtContent = {
    // This is the subject of the JWT which will be the ID of the
    //
    // This user ID will be available under integrations.channel.
    // system_integrations.channel.private.user.id in actions.
    sub: anonymousUserID,
    // This object is optional and contains any data you wish to
    // encrypted using the public key so it will not be visible t
```

```
    user_payload: {
      custom_message: 'Encrypted message',
      name: 'Anonymous',
    },
    context
  };

  // If the user is authenticated, then add the user's real info
  if (sessionInfo) {
    jwtContent.user_payload.name = sessionInfo.userName;
    jwtContent.user_payload.custom_user_id = sessionInfo.customUs
  }

  const dataString = JSON.stringify(jwtContent.user_payload);

  // Encrypt the data
  const encryptedBuffer = crypto.publicEncrypt(
    {
      key: PUBLIC_KEY,
      padding: crypto.constants.RSA_PKCS1_OAEP_PADDING,
      oaepHash: 'sha256'  // Specify OAEP padding with SHA256
    },
    Buffer.from(dataString, 'utf-8')
  );

  // Convert encrypted data to base64
  jwtContent.user_payload = encryptedBuffer.toString('base64');
  console.log(jwtContent.user_payload)


  // Now sign the jwt content to make the actual jwt. We are givi
  // to demonstrate the web chat capability of fetching a new tok
  // you would likely want to set this to a much higher value or
  const jwtString = jwtLib.sign(jwtContent, PRIVATE_KEY, {
    algorithm: 'RS256',
    expiresIn: '10000000s',
  });

  return jwtString;
}
```

```
/**
 * Gets or sets the anonymous user ID cookie. This will also ensu
 * day expiration time.
 */
function getOrSetAnonymousID(request, response) {
  let anonymousID = request.cookies['ANONYMOUS-USER-ID'];
  if (!anonymousID) {
    // If we don't already have an anonymous user ID, then create
    // but for the sake of this example we are going to shorten i
    anonymousID = `anon-${uuid().substr(0, 5)}`;
  }

  // Here we set the value of the cookie and give it an expiratio
  // have an ID to make sure that we update the expiration date t
  response.cookie('ANONYMOUS-USER-ID', anonymousID, {
    expires: new Date(Date.now() + TIME_45_DAYS),
    httpOnly: true,
  });

  return anonymousID;
}

/**
 * Returns the session info for an authenticated user.
 */
function getSessionInfo(request) {
  // Normally the cookie would contain a session token that we wo
  // like a database. But for the sake of simplicity in this exam
  // info.
  const sessionInfo = request.cookies.SESSION_INFO;
  if (sessionInfo) {
    return JSON.parse(sessionInfo);
  }
  return null;
}

/**
 * Handles the createJWT request.
 */
```

```
function createJWT(request, response) {
  const anonymousUserID = getOrSetAnonymousID(request, response);
  const sessionInfo = getSessionInfo(request);

  const context = {
    dev_id: 23424,
    dev_name: "Name",
    is_active: true
  }

  response.send(createJWTString(anonymousUserID, sessionInfo,cont
}

router.get('/', createJWT);
```

Collapse

You can check the examples for watsonx Assistant web chat that are mostly compatible with the watsonx Orchestrate embedded chat to see how to use this code example

After generating the JWT token, pass it to the embedded web chat. The following example shows how to do that:

JavaScript

```
<script>
    function getUserId() {
        let embed_user_id = getCookie('embed_user_id');
        if (!embed_user_id) {
            embed_user_id = Math.trunc(Math.random() * 1000000);
            setCookie('embed_user_id', embed_user_id);
        }
        return embed_user_id;
    }

    function getCookie(name) {
        console.log('getCookie');
        const value = `; ${document.cookie}`;
        const parts = value.split(`; ${name}=`);
        if (parts.length === 2) return parts.pop().split(';').shi
    }

    function setCookie(name, value) {
        document.cookie = `${name}=${value}; path=/`;
    }
    function preSendHandler(event) {
        if (event?.message?.content) {
            event.message.content = event.message.content.toUpper
        }
    }

    function sendHandler(event) {
        console.log('send event', event);
    }

    function feedbackHandler(event) {
        console.log('feedback', event);
    }

    function preReceiveHandler(event) {
        event?.content?.map((element) => {
            element.type = 'date';
        });
```

```
}

function receiveHandler(event) {
    console.log('received event', event);
}

function userDefinedResponseHandler(event) {
    console.log('userDefinedResponse event', event);
    event.hostElement.innerHTML = `
                    <cds-code-snippet>
                        node -v Lorem ipsum dolor sit amet, c
                        veritatis voluptate id incidunt moles
                        alias, architecto hic, dicta fugit? D
                        laboriosam!
                    </cds-code-snippet>
                    <br><br>
                    <div style="background-color:orange;color
                        <p>${event.contentItem?.template || '
                    </div>`;
}

function onChatLoad(instance) {
    instance.on('chatstarted', (instance) => {
        window.wxoChatInstance = instance;
    });
    instance.on('pre:send', preSendHandler);
    instance.on('send', sendHandler);
    instance.on('pre:receive', preReceiveHandler);
    instance.on('receive', receiveHandler);
    instance.on('feedback', feedbackHandler);
    instance.on('userDefinedResponse', userDefinedResponseHan
}
async function getIdentityToken() {
    // This will make a call to your server to request a new
    const result = await fetch(
        "http://localhost:3000/createJWT?user_id=" + getUserI
    );
    window.wxOConfiguration.token = await result.text();
}
```

```
window.wxOConfiguration = {
    orchestrationID: "20250430-0912-2925-309a-35c6bef54760_20
    hostURL: "https://us-south.watson-orchestrate.cloud.ibm.c
    rootElementID: "root",
    deploymentPlatform: "ibmcloud",
    crn: "crn:v1:bluemix:public:watsonx-orchestrate:us-south:
    chatOptions: {
        agentId: "852431a8-32dd-4925-8cc3-9ea3d3162726",
        agentEnvironmentId: "5d769a04-9445-4768-a687-710d6e9a
        onLoad: onChatLoad
    },
};
getIdentityToken().then(() => {
    const script = document.createElement("script");
    script.src = `${window.wxOConfiguration.hostURL}/wxochat/
    script.addEventListener("load", function () {
        wxoLoader.init();
    });
    document.head.appendChild(script);
});
```

Collapse

Customizing the embedded chat

Header

Header is an optional property in `wxOConfiguration` that controls whether header actions appear.

Parameter	Type	Description
`header.showResetButton`	boolean	Displays the Reset Chat button in the header when set to true. Defaul is `true` .
`header.showAiDisclaimer`	boolean	Displays the AI disclaimer icon/button in the header when set to true. Default is `true` .

Language

Parameter	Type	Description
`defaultLocale`	string	Defines the default language supported by the chat. Supported values: `de` , `en` , `es` , `fr` , `it` , `ja` , and `pt-BR`

Styles

You can customize embedded web chats to create a unique chat interface that better fits your webpage.

To apply custom styles, add a `style` component inside the `window.wxOConfiguration` object in your web chat script. In this component, you can configure the following elements:

Header is an optional property of `wxOConfiguration` that controls the visibility of header actions.

Parameter	Type	Description
`headerColor`	string	Set a six-digit hex code that defines the chat header color.
`userMessageBackgroundColor`	string	Set a six-digit hex code that defines the user message bubble color.
`primaryColor`	string	Set a six-digit hex code that defines the interactive elements color.
`showBackgroundGradient`	boolean	Displays the background gradient when set to true. Default is `true` .

The following is an example of how to customize the embedded web chat using the `style` component inside `window.wxOConfiguration` :

Basic customization Customize header and style

```
<script>
window.wxOConfiguration = {
  orchestrationID: "my-tenant-id",
  hostURL: "my-host-url",
  rootElementID: "root",            // fullscreen-overlay only
  showLauncher: false,              // fullscreen-overlay only, fa
  chatOptions: {
    agentId: "12345_test_agent1",           // required
    agentEnvironmentId: "my-agent-env-id"   // required
  },
  header: {
    showResetButton: true,   // optional; defaults to true
    showAiDisclaimer: true   // optional; defaults to true
  },
   style: {
   headerColor: '', //6-digit hex value or empty for default
   userMessageBackgroundColor: '', //6-digit hex value or empty fo
   primaryColor: '', //6-digit hex value or empty for default
   showBackgroundGradient: true, // optional; defaults to true
  },
};

setTimeout(function() {
  const script = document.createElement('script');
  script.src = `${window.wxOConfiguration.hostURL}/wxochat/wxoLoad
  script.addEventListener('load', () => wxoLoader.init());
  document.head.appendChild(script);
}, 0);
</script>
```

··· Collapse

Layout

The watsonx Orchestrate embed supports a flexible `layout` object to control how and where the chat UI appears.

Parameter	Type	Default	Descriptic
`rootElementID`	string	—	(`fullscr` `overlay` the conta mount ch
`showLauncher`	boolean	true	(`fullscr` `overlay` the bubbl (true) or r immediat
`layout.form`	string	float	Defines th form of y Use `ful` `overlay` the web c fullscreen additiona are requir Use `floa` display th as a floati Also confi `wid` the w `hei` of th Use `cus` define a c layout. Al the `cust` paramete custom el
`layout.width`	string	—	(float onl width (e.g '30rem').
`layout.height`	string	—	(float onl height (e. '40rem').

Parameter	Type	Default	Descriptic
`layout.showOrchestrateHeader`	boolean	true	Render th header ba hide it (fa
`layout.customElement`	HTMLElement	—	element r render int

```
<script>
window.wxOConfiguration = {
  orchestrationID: "my-tenant-id",
  hostURL: "my-host-url",
  rootElementID: "root",            // fullscreen-overlay only
  showLauncher: false,              // fullscreen-overlay only, fal

  chatOptions: {
    agentId: "12345_test_agent1",           // required
    agentEnvironmentId: "my-agent-env-id"   // required
  },

  layout: {
    form: 'float',                          // 'fullscreen-overlay
    width: '600px',                         // float only
    height: '600px',                        // float only
    showOrchestrateHeader: true,           // hide header if false
    customElement: hostElement             // custom only
  }
};

setTimeout(function() {
  const script = document.createElement('script');
  script.src = `${window.wxOConfiguration.hostURL}/wxochat/wxoLoade
  script.addEventListener('load', () => wxoLoader.init());
  document.head.appendChild(script);
}, 0);
</script>
```

Collapse

The following is an example of how to customize the layout of the embedded web chat to display it in fullscreen mode:

JavaScript

```
<script>
window.wxOConfiguration = {
  orchestrationID: "my-tenant-id",
  hostURL: "my-host-url",
  rootElementID: "root",
  showLauncher: false,
  chatOptions: {
    agentId: "test_agent1",
    agentEnvironmentId: "my-agent-draft-env-id"
  },

  layout:{
    form: 'fullscreen-overlay',
    showOrchestrateHeader: true,
  }

};

setTimeout(function() {
  const script = document.createElement('script');
  script.src = `${window.wxOConfiguration.hostURL}/wxochat/wxoLoad
  script.addEventListener('load', () => wxoLoader.init());
  document.head.appendChild(script);
}, 0);
</script>
```

Collapse

Feedback, events and instance methods

Thumbs-up and thumbs-down feedback

In the embedded chat, you need to manually enable thumbs-up and thumbs-down feedback using `pre:receive` handlers. First, subscribe to the `pre:receive` event to inject feedback options. Then, handle submitted feedback through the `feedback` event.

The following script shows how to configure feedback in the embedded chat:

JavaScript

```
<script>
function feedbackHandler(event) {
    console.log('feedback', event);
}

function preReceiveHandler(event) {
    console.log('pre-receive event', event);
    const lastItem = event?.message?.content?.[event.message.cont
    if (lastItem) {
        lastItem.message_options = {
            feedback: {
                is_on: true,
                show_positive_details: false,
                show_negative_details: true,
                // Note, these positive details are not used as l
                positive_options: {
                    categories: ['Funny', 'Helpful', 'Correct'],
                    disclaimer: "Provide content that can be shar
                },
                negative_options: {
                    categories: ['Inaccurate', 'Incomplete', 'Too
                    disclaimer: "Provide content that can be shar
                },
            },
        };
    }
}

function onChatLoad(instance) {
    instance.on('pre:receive', preReceiveHandler);
    instance.on('feedback', feedbackHandler);
}

window.wxOConfiguration = {
    ...
        };
setTimeout(function () {
            ...
```

```
        }, 0);
</script>
```

Collapse

watsonx Orchestrate does not persist feedback internally. You are responsible for storage and analysis. If you want to store feedback on your backend, you can capture feedback data and send it to your own backend API:

sendFeedbackToBackend.js

```
instance.on('feedback', (feedbackEvent) => {
  fetch('/api/store-feedback', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
      'Authorization': `Bearer ${jwtToken}`
    },
    body: JSON.stringify({
      feedback: feedbackEvent.feedback,
      messageId: feedbackEvent.messageId,
      sessionId: feedbackEvent.sessionId
    })
  });
});
```

Events

Embedded web chat supports a variety of events that allow you to trigger specific actions or customize behavior. The following tables list all supported events, grouped by category.

Customization Events

Event name	Description
`userDefinedResponse`	Triggered when a response contains an unrecognized or `user_defined` response type.

Message Events

Event name	Description
`pre:receive`	Triggered before the web chat receives a response from the agent.
`pre:send`	Triggered before the web chat sends a message to the agent.
`receive`	Triggered after the web chat receives a response from the agent.
`send`	Triggered after the web chat sends a message to the agent.
`pre:restartConversation`	Triggered before the conversation restarts. Useful for alerting the user that the chat will reset, allowing them to complete any ongoing actions (e.g., finishing a tool call).
`restartConversation`	Triggered after the conversation restarts, before a new session begins. Useful for displaying specific UI elements when a new session starts.

View Events

Event name	Description
`view:pre:change`	Triggered before the view state changes.
`view:change`	Triggered after the view state changes.

Security Events

Event name	Description
`identityTokenExpired`	Triggered when security is enabled and the JWT token expires.

Miscellaneous Events

Event name	Description
`chat:ready`	Triggered when the web chat is fully loaded and ready to receive user input. Useful for displaying a welcome message or initializing UI components.

Instance methods

While events are used to listen to the embedded chat, instance methods are used to take actions on the embedded chat. The following tables list all supported instance methods, grouped by category.

Message

Event name	Description
`doAutoScroll`	Scrolls to the most recent message in the list.
`scrollToMessage`	Scrolls the messages list to a specific message.
`send`	Sends the specified message to the agent.
`updateHistoryUserDefined`	Updates a user_defined property in message history. Need to store the `user_defined` in the `message_state` column of the messages table so that it will appear in the history
`restartConversation`	Restarts the conversation with the agent.

User Interface

Event name	Description
`changeView`	Changes the current view state.
`updateLocale`	Changes the display language. Supported values: `de`, `en`, `es`, `fr`, `it`, `ja`, and `pt-BR`.

Security/Identity

Event name	Description
`updateIdentityToken`	Replaces the current JWT with a new one for continued secure communication. This is commonly used when tokens expire or are refreshed during a session.
`destroy`	Destroys the web chat and removes it from the page.

Events

Event name	Description
`once`	Subscribes a handler function to an event so it runs only once when that event occurs. After the event fires, the handler is no longer called.
`off`	Removes a subscription to an event type.
`on`	Subscribes to a type of event.

Events and instance methods example

The following example shows how to configure events and instance methods in the embedded web chat:

JavaScript

```
<script>
    function preSendHandler(event, instance) {
        console.log('pre:send event', event);
        if (event?.message?.message?.content) {
            event.message.message.content = event.message.mes
        }
    }

    function sendHandler(event, instance) {
        console.log('send event', event);
    }

    function feedbackHandler(event, instance) {
        console.log('feedback', event);
    }

    function preReceiveHandler(event, instance) {
        console.log('pre-receive event', event);
        event?.message?.content?.map((element) => {
            if (element?.text?.includes('assistant')) {
                element.text = element.text.replace('assistan
            }
            element.type = 'user_defined';
        });

        const lastItem = event?.message?.content?.[event.mess
        if (lastItem) {
            lastItem.message_options = {
                feedback: {
                    is_on: true,
                    show_positive_details: false,
                    show_negative_details: true,

                    positive_options: {
                        categories: ['Funny', 'Helpful', 'Cor
                        disclaimer: "Provide content that can
                    },
                    negative_options: {
```

```
                    categories: ['Inaccurate', 'Incomplet
                    disclaimer: "Provide content that can
                },
            },
        };
    }
}

function receiveHandler(event, instance) {
    console.log('received event', event);
    instance.off('pre:receive', preReceiveHandler);
    instance.updateAuthToken("wrong-or-expired-token")
}

function userDefinedResponseHandler(event, instance) {
    console.log('userDefinedResponse event', event);
    event.hostElement.innerHTML = `
            <cds-code-snippet>
                node -v Lorem ipsum dolor sit amet, conse
                veritatis voluptate id incidunt molestiae
                alias, architecto hic, dicta fugit? Debit
                laboriosam!
            </cds-code-snippet>
            <br><br>
            <div style="background-color:orange;color:whi
                <p>${event.contentItem?.text || '[No mess
            </div>`;
}

function preRestartConversationHandler(event, instance) {
    console.log('pre:restartConversation event', event);
}

let calledRestartConversation = false;
function restartConversationHandler(event, instance) {
    console.log('restartConversationHandler event', event
    if (!calledRestartConversation) {
        setTimeout(() => {
            instance.send('Hello from embedded web chat s
        }, 3000);
```

```
            calledRestartConversation = true;
        }

    }

    function preThreadLoadedHandler(event, instance) {
        console.log('pre:threadLoaded event', event);
        event.messages[0].content[0].text = 'Modified prompt
    }

    async function authTokenNeededHandler(event, instance) {
        console.log('authTokenNeeded event', event);
        event.authToken = "<Refreshed Token>"
    }

    function onChatLoad(instance) {
        instance.on('chat:ready', (event, instance) => {
            console.log('chat:ready', event);
        });
        instance.once('pre:send', preSendHandler);
        instance.on('send', sendHandler);
        instance.once('pre:receive', preReceiveHandler);
        instance.on('receive', receiveHandler);
        instance.on('feedback', feedbackHandler);
        instance.on('userDefinedResponse', userDefinedRespons
        instance.on('pre:restartConversation', preRestartConv
        instance.on('restartConversation', restartConversatio
        instance.on('pre:threadLoaded', preThreadLoadedHandle
        instance.on('authTokenNeeded', authTokenNeededHandler
    }
    window.wxOConfiguration = {
        clientVersion: 'latest',
        orchestrationID: '<tenantId>',
        hostUrl: 'http://localhost:3000',
        showLauncher: true,
        rootElementId: 'root',
        chatOptions: {
            agentId: '<agentId>',
            agentEnvironmentId: '<agentEnvironmentId>',
            onLoad: onChatLoad,
```

```
        },
    };
    setTimeout(function () {
        const script = document.createElement('script');
        script.src = `${window.wxOConfiguration.hostUrl}/wxoL
        script.addEventListener('load', function () {
            wxoLoader.init();
        });
        document.head.appendChild(script);
```

Collapse

API Integration

You can also integrate your agent with external applications by using the ADK's provided agent completions API. These APIs allow agents to be shared across multiple watsonx Orchestrate instances:

Orchestrate Native Runs API: For long-duration workflows.

API Documentation:

Non streaming Runs API

Streaming Runs API

Chat Completions Compatibility Layer: OpenAI-compatible for easy integration.

API Documentation:

Chat Completions API

Copyright information

Privacy and accessibility

Privacy

Accessibility

Terms of use