Using agents in embedded webchat

IBM watsonx Orchestrate's Embedded Chat feature allows you to integrate the watsonx Orchestrate chat experience into your own web UI applications. To ensure secure communication between your application and the watsonx Orchestrate service, the Embedded Chat feature includes security mechanisms that use public-key cryptography.

Generating embedded webchat

To simplify integration with your website, the CLI includes the `orchestrate channels webchat embed` command. This command takes the name of an agent and produces a script tag that you can place in the `<head></head>` section of your page for the currently active environment.

BASH

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

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

Embedded chat security

By default, security is **enabled, but not configured** for the embedded chat. This means:

- The embedded chat will not function until security is **properly configured**
- You **must** configure **both IBM and client key pairs** for the chat to work
- Alternatively, you can **explicitly disable security** to allow anonymous access

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

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

Context variables for embedded webchat

To use context variables in embedded webchat, include them inside the JWT token.

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:

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, '../keys

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


// 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

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

JavaScript

```
<script>
    function getUserId() {
        let embed_user_id = sessionStorage.embed_user_id;
        if (!embed_user_id) {
            embed_user_id = Math.trunc(Math.random() * 1000000);
            sessionStorage.embed_user_id = embed_user_id;
        }
        return embed_user_id;
    }
    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, 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?.template || '[No
                </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:3003/createJWT?user_id=" + getUserI
    );
    window.wxOConfiguration.token = await result.text();
}
window.wxOConfiguration = {
    orchestrationID: "20250430-0912-2925-309a-35c6bef54760_20
    hostURL: "https://dl.watson-orchestrate.ibm.com",
    rootElementID: "root",
    chatOptions: {
        agentId: "852431a8-32dd-4925-8cc3-9ea3d3162726",
        agentEnvironmentId: "5d769a04-9445-4768-a687-710d6e9a
    },
    style: {
        headerColor: '#b8b890',
        userMessageBackgroundColor: '#ffa31a',
        primaryColor: '#33ff3c',
```

```
        },
        showLauncher: false,
        layout: {
            form: 'fullscreen-overlay',
            showOrchestrateHeader: true,
        }
    };
    setTimeout(function () {
        const script = document.createElement('script');
        script.src = `${window.wxOConfiguration.hostURL}/wxochat/
        script.addEventListener('load', function () {
            wxoLoader.init();
        });
        document.head.appendChild(script);
    }, 0);
    getIdentityToken().then(() => {
        const script = document.createElement("script");
        script.src = `${window.wxOConfiguration.hostUrl}/wxochat/
        script.addEventListener("load", function () {
            wxoLoader.init();
        });
        document.head.appendChild(script);
    });
```

··· Collapse

Customizing embedded webchat

Configuring 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. Default is `true` .

Parameter	Type	Description
`header.showAiDisclaime`	boolean	Displays the AI disclaimer icon/button in the header when set to true. Default is `true`.

Customizing 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:

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.
`style.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`:

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"
  },

 style: {
   headerColor: '#000000',
   userMessageBackgroundColor: '#000000',
   primaryColor: '#000000'
  },

};

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

Customizing layout

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

arameter	Type	Default	Description
`rootElementID`	string	—	(`fullscreen` `overlay` onl the container to mount cha
`showLauncher`	boolean	true	(`fullscreen` `overlay` onl the bubble la (true) or rend immediately (
`layout.form`	string	float	Defines the la form of your chat. Use `fullscr` `overlay` to the web chat fullscreen mo additional parameters a required. Use `float` display the w as a floating Also configur `width` : of the we `height` Height o web chat Use `custom` define a custo layout. Also configure the `customElem` parameter wi custom eleme
`layout.width`	string	—	(float only) Po width (e.g. '3 '30rem').

arameter	Type	Default	Description
`layout.height`	string	—	(float only) Pc height (e.g. '5 '40rem').
`layout.showOrchestrateHeader`	boolean	true	Render the st header bar (tı hide it
`layout.customElement`	HTMLElement	—	element refer render into.

JavaScript

```
<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
  },

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

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

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

Enabling thumbs-up and thumbs-down

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?.content?.[event.content.length - 1];
    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

Events reference

Embedded webchat 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:send`	Triggered before the webchat sends a message to the assistant.
`send`	Triggered after the webchat sends a message to the assistant.
`pre:receive`	Triggered before the webchat receives a response from the assistant.
`receive`	Triggered after the webchat receives a response from the assistant.

Event name	Description
`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.
`pre:threadLoaded`	Triggered when a user navigates to a chat thread in full-screen embedded chat. Useful for displaying custom responses or UI elements.

Security Events

Event name	Description
`identityTokenExpired`	Triggered when security is enabled and the JWT token expires.
`authTokenNeeded`	Triggered when the embedded chat requires a refreshed or updated authentication token.

Miscellaneous Events

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

Events example

The following example shows how to configure events in the embedded webchat:

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 webchat se
        }, 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