Pexip Infinity Connect Plugin Deployment Guide

About this guide

This guide explains how to use the **Pexip client plugin API** to create plugins that extend the functionality of the Pexip Infinity Connect client applications.

This document covers:

Introduction

About Infinity Connect plugins

The next-generation Pexip Infinity Connect client applications support extension via a JavaScript plugin mechanism. This mechanism allows the administrator to extend the functionality of the application with additional menus, buttons and behaviors.

Plugins can be used to add buttons to the control menus, toolbars, and context menus that are available to meeting participants using the Infinity Connect applications. The application provides the plugin with information about the current meeting and/or selected participant. The plugin can also be used to access chat messages and functions exposed via the PexRTC JavaScript client API.

Plugins are able to perform simple actions such as sending requests via PexRTC (e.g. chat messages), sending queries to the client API directly (e.g. changing layouts), and performing queries or sending requests to external services (such as IFTTT) to gather information or perform other operations.

Example plugins

Examples are given throughout this document based on existing sample plugins available from the Pexip Branding Portal. To access these plugins:

1. From within the Branding Portal, select the **Customizations** option. Either create a new customization or select an existing customization, and then from the panel on the left select **Plugins** .
2. Select the plugins you wish to use.
3. From the **Dashboard**, build a branding package using the customization that includes the selected plugins.
4. Download the branding package ZIP file. Within this file, there will be a **webapp2** folder that contains the **manifest.json** and **settings.json** files, and a **plugins** subfolder containing further subfolders for each of the plugins.

For full instructions, see Creating a branding package via the Pexip branding portal.

Out of scope

This document does not go into details regarding the layout and design of plugin user interfaces.

Prerequisites

You should be proficient with JavaScript and also proficient in working with REST APIs. In particular, the examples here assume you are familiar with the basic operation of the Pexip REST API for clients as documented in Pexip client REST API v2.

You should also be familiar with the basic concepts of WebRTC as outlined in https://webrtc.org/faq/.

You should already have a fully operational Pexip Infinity deployment that includes Infinity Connect clients, and be familiar with their use.

You should also be familiar with the basic concepts of the Pexip platform and the services it offers (see Introduction to Pexip Infinity and Pexip Infinity services).

API changes and caveats

The **Pexip client plugin API** is still in development and, as such, features may be changed, added or removed. Whilst we will try to maintain compatibility with existing plugins and structures, we cannot guarantee the API will remain the same.

Pexip cannot provide detailed development / design assistance for plugins. If your Infinity Connect application experiences crashes or instabilities with plugins deployed, you will be asked to remove all plugins from the deployment (to ensure they are not the cause of the issue) before the normal support process will continue.

A word on security

Because there is potentially sensitive information traveling over the client API, all client connections are performed over an HTTPS connection. Valid certificates should be installed for your platform (see Certificates overview), and you should not use self-signed certificates or make attempts to bypass any OS level checks (e.g. Application Transport Security on the iOS platform). Since you are in complete control of each HTTPS connection you could even perform things like certificate pinning but this is beyond the scope of this document.

You should also heed industry standard practices when parsing information received either from user input or from any incoming connection, and also when storing / logging information (e.g. passwords or tokens), to ensure your application does not compromise the users' security in any way.

Plugin structure

A plugin comprises two main files:

- the plugin package file, which is a JSON file that describes the plugin and how it operates (see Plugin package file), and
- the source of the plugin, which is a plain JavaScript file (see Plugin source file).

Other files, such as image files used for buttons and icons, may also be included.

The following two files must also be provided in the root of the **webapp2** directory in order to make the plugin available to users:

- **settings.json** - see Editing settings.json
- **manifest.json** - see Editing manifest.json

Deploying the plugin

Plugins are deployed as part of the platform's application customizations, via the **Infinity Connect Web Application Customization** page on the Pexip Infinity Management Node Administrator interface (**Services > Web App Customization**.

To deploy a plugin you download, edit and upload an existing branding package, either by:

- downloading the default branding package from the Management Node, adding your plugin files and the manifest.json file, editing the settings.json file, and then zipping and uploading the revised package, or
- using the Pexip Branding Portal to create a branding package that incorporates one of the existing sample plugins along with the above two files, using the sample plugins as a basis for your own plugin files, and uploading the revised zip file.

For more information, see Manually configuring the branding files.

In order for your plugins to be recognized, you'll need to let the Infinity Connect application know:

1. that its settings have been changed, by Editing manifest.json, and
2. what the new settings are, by Editing settings.json.

Editing settings.json

To make the plugin available for use, you reference it in the **settings.json** file for the main application. The **settings.json** file can contain a list of available plugins.

- To define which plugins are available within the application, modify the list of `plugins` at the bottom of **settings.json** to include the ID and source of the package file.
- To load the plugin (i.e. enable it) by default, set `"enabled": true`. Users can load and unload any of the available plugins themselves.

The following extract from our example **settings.json** file shows how the spotlight plugin is made available and enabled by default:

```
"plugins": [
  {
    "id": "spotlight-plugin-1.0",
    "srcURL": "plugins/spotlight/spotlight.plugin.package.json",
    "enabled": true
  }
]
```

Example extract from settings.json showing spotlight plugin activation

Editing manifest.json

The **manifest.json** file triggers the application to upload the custom configuration described in the **settings.json** file.

Each time there is an addition or change to any of the files in the branding package (including the **settings.json** file), you must change the `brandingID` field in the **manifest.json** file. It doesn't matter if this number is higher or lower; as long as it has changed from the previous version of the file, it will trigger the application to upload the updated branding package - including the plugins.

The **manifest.json** file must be saved in the root of the **webapp2** directory. This file will already exist if you have used the Pexip Branding Portal to create an example plugin for editing, but if you have downloaded the default branding package from the Management Node, you must create it yourself. An example is shown below:

```
{
  "brandingID":"1521829718679",
  "isSettingsBranded":true,
  "isWatermarkBranded":false,
  "isStylesBranded":false
}
```

Example manifest.json file

After the plugin has been deployed successfully, the **Plugin** option will appear in users' **Settings** page; clicking on that option will show them a list of available plugins.

Plugin package file

Overview

The plugin package file is a JSON file that describes the plugin and how it operates. It is referenced in **settings.json**.

Below is an example of the spotlight plugin package file:

```
{
  "id": "spotlight-plugin-1.0",
  "name": "spotlight",
  "description": "Applies a spotlight to a participant from the roster",
  "version": 1.0,
  "srcURL": "spotlight.plugin.js",
  "allowUnload": true,
  "platforms": ["web", "electron", "ios", "android"],
  "participantRoles": ["chair"],
  "menuItems": {
    "participants": [{
      "icon": null,
      "label": null,
      "action": "spotlightParticipant"
    }]
  }
}
```

Example spotlight plugin package file

Content

The fields required in the package file are as follows:

Field	Description
id	A unique id that will be used to reference the plugin in various places.
name	The name for the plugin, which will appear in the application under the **Settings > Plugin** menu.
description	The text that describes what the plugin does, which will appear in the application under the **Settings > Plugin** menu.
version	An internal version number for the plugin.
srcURL	The location of the Plugin source file.
allowUnload	Determines whether the user can unload (disable) this plugin: • ***true***: the user can disable the plugin. • ***false***: the user cannot disable the plugin.
platforms	A list of platforms for which this plugin will be enabled: • ***web***: enabled for the web app. • ***electron***: enabled for the desktop client. • ***ios***: enabled for the mobile client for iOS. • ***android***: enabled for the mobile client for Android.
participantRoles	A list of roles that can use this plugin: • ***chair***: available to Hosts. • ***guest***: available to Guests.
menuItems	Determines how and where the plugin will appear in the user interface: • ***conference***: appears as a menu item in the conference control menu. • ***toolbar***: appears as a button in the floating conference toolbar. • ***participants***: appears as a menu item in the context menu for a participant. ℹ You can only specify one of the above options. Each of the three options above in turn has three fields: • **icon**: the location of the image to use for the icon. Can be `null`. For more information, see Menu item icons. • **label**: the textual label for the item. Can be `null`. For more information, see Menu item labels • **action**: the name of the function to call from the Plugin source file. For example, `"action": "spotlightParticipant"` in the spotlight package file calls `function spotlightParticipant(participant)` in the spotlight source file.

Menu item icons

We recommend using an SVG image for your icon files. When creating the image, you **must** include an `id` field in the source of the SVG, and also reference this `id` in the icon field in the package.

For example, the **grid.svg** file for the layout plugin contains the `id` `"grid"`:

```
<svg
  id="grid"
  fill="currentColor"
  viewBox="0 0 24 24"
  xmlns="http://www.w3.org/2000/svg">
```

which is then referenced in the `icon` field of the layout package file as follows:

```
   "menuItems": {
       "conference": [{
           "icon": "grid.svg#grid",
           "label": "Change conference layout",
           "action": "changeConferenceLayout"
       }]
   }
```

If you wish to dynamically update this icon or create the icon based on state, you should set the `icon` field to `null` (as in the spotlight example) and modify it from the plugin — see Plugin state.

Menu item labels

The text in the `label` field will be visible to users (for example, the text that is shown in the conference control menu, or for toolbar items the text that appears when hovering over the button). If you wish to dynamically update the label based on state or other information, you should set the `label` field to `null` (as in the spotlight example) and modify it from the plugin — see Plugin state.

Plugin source file

Content

Your plugin source must be a plain JavaScript file. You must place the body of your source inside an IIFE (Immediately Invoked Function Expression) so as to not pollute the global name-space.

There are four main functions you must include:

1. **load**: this performs any actions on loading of the plugin, e.g. setting menu labels based on state.
2. **unload**: this performs any cleanup actions when the plugin is disabled.
3. **<doSomething>**: this is the main body of your code. The name should be reflected in the package json and also in the **registerPlugin** function call (see below).

   For example, `"action": "spotlightParticipant"` in the spotlight package file calls `function spotlightParticipant(participant)` in the spotlight source file.
4. **PEX.pluginAPI.registerPlugin**: this must be called in order for the plugin to be loaded into the application. It should include the ID of the plugin (as defined in the Plugin package file) and the names of the three functions listed above.

Following is an example based on the spotlight plugin:

```
// Use IIFE (Immediately Invoked Function Expression) to wrap the plugin to not pollute global namespace with whatever is
defined inside here
(function () {
  var state$ = PEX.pluginAPI.createNewState({});

  // called by the PluginService when plugin is loaded
  function load(participants$, conferenceDetails$) {
    participants$.subscribe(participants => {
      var state;
      participants.map(participant => {
        if (participant.spotlight === 0) {
          state = Object.assign({}, state, {
            [participant.uuid]: {
              icon: 'star.svg#star',
              label: 'Spotlight'
            }
          });
        } else {
          state = Object.assign({}, state, {
            [participant.uuid]: {
              icon: 'star-fill.svg#star-fill',
              label: 'Unspotlight'
            }
          });
        }
      });

      if (state) {
        state$.next(state);
      }
    });
  }

  // item click functions
  function spotlightParticipant(participant) {
    if (participant.spotlight === 0) {
      PEX.pluginAPI.spotlightOnParticipant(participant.uuid);
    } else {
      PEX.pluginAPI.spotlightOffParticipant(participant.uuid);
    }
  }

  // unload / cleanup function
  function unload() {
    // clean up any globals or other cruft before being removed
    console.log('unload spotlight-plugin');
  }

  // Register our plugin with the PluginService - make sure id matches your package.json
  PEX.pluginAPI.registerPlugin({
    id: 'spotlight-plugin-1.0',
    load: load,
    unload: unload,
    spotlightParticipant: spotlightParticipant,
    state$: state$,
  });

})(); // End IIFE
```

Example spotlight plugin source file

User interfaces

Depending on your plugin, there may be no requirement for a user interface (for example if it just listens to events as in window.PEX.actions$).

However, if you do require the ability to display a dialog then you should use the **openTemplateDialog** call (for more information, see openTemplateDialog). Styling can be achieved either using inline CSS, or by using some of the standard styles available in the Pexip style sheet, which can be viewed via **https://<conferencing_node>/webapp2/app.css**.

For example, `class="dialog-button green-action-button"` would provide you with the same green action button used throughout the application. You can also style your input boxes using the existing Pexip CSS styles.

Plugin API

pluginAPI functions

Available on the `window.PEX.pluginAPI` object. Currently limited to the following functions, these will be expanded in future releases.

registerPlugin

Called by your plugin to register with the application. See the example spotlight plugin for an example of use.

createNewState

Creates states — see Plugin state.

openTemplateDialog

This provides a mechanism to create a dialog with a title and a body.

The body should contain your template. If you are performing simple operations or showing basic information, you can simply create inline JavaScript in this template (for example, `onClick` methods on buttons as in the example layout plugin). If you want to perform complex operations, it is better perform your actions in a `.then`, which will run after the promise to open the dialog has resolved.

ℹ The syntax for calling dialogs changed between v21 and v22. For information on migrating your existing plugins, see Migrating from v21 to v22.

```
function changeConferenceLayout(conferenceDetails) {
       PEX.pluginAPI.openTemplateDialog({
               title: 'Layout control',
               body: `<div style="display: flex; flex-wrap: wrap; justify-content: center;">
                       <!-- actors_overlay_text can be set to auto or off on each of the layouts.-->
                       <button class="dialog-button blue-action-button" onclick="window.PEX.pluginAPI.transformConferenceLayout
({layout:'4:0'})">Equal</button>
                       <button class="dialog-button blue-action-button" onclick="window.PEX.pluginAPI.transformConferenceLayout
({layout:'1:0'})">1+0</button>
                       <button class="dialog-button blue-action-button" onclick="window.PEX.pluginAPI.transformConferenceLayout
({layout:'1:7'})">1+7</button>
                       <button class="dialog-button blue-action-button" onclick="window.PEX.pluginAPI.transformConferenceLayout
({layout:'1:21'})">1+21</button>
                       <button class="dialog-button blue-action-button" onclick="window.PEX.pluginAPI.transformConferenceLayout
({layout:'2:21'})">2+21</button>
                       <button class="dialog-button blue-action-button" onclick="window.PEX.pluginAPI.transformConferenceLayout
({})">Default</button>
               </div>`
       });
}
```

Example from layout plugin showing use of onClick methods in the openTemplateDialog function

spotlightOnParticipant

Takes a participant UUID and sets the spotlight. For more information, see setParticipantSpotlight.

spotlightOffParticipant

Takes a participant UUID and removes the spotlight. For more information, see setParticipantSpotlight.

dialOut

Initiates a call from the current conference to a participant. Takes an `alias`, `protocol` and `role`; provides a participant UUID if successful. For more information, see dialOut.

disconnectParticipant

Takes a participant UUID and disconnects them. See disconnectParticipant.

changeConferenceLayout

This function is being deprecated - you should use transformConferenceLayout instead. For more information, please contact your Pexip authorized support representative.

transformConferenceLayout

Takes a JSON dictionary containing the layout required. This function extends the capabilities of the **changeConferenceLayout** function. It makes use of the **transform_layout** function in the Pexip client REST API to change the conference layout, control streaming content, and to enable/disable indicators and overlay text.

sendChatMessage

Takes a message to send into the conference.

Observable

dialogRef.viewInit$

Subscribe to this if you want to perform actions (e.g. add event listeners) to your dialog. At this point, the DOM has fully loaded and you'll have access to all the elements. See the example below for usage.

dialogRef.close$

Subscribing to this will allow you to perform actions when the dialog has been closed.

```
function doSomething(conferenceDetails) {
    PEX.pluginAPI.openTemplateDialog({
    title: 'My Title',
    body: '<button id="myButtonId">Do Something</button>'
    }).subscribe(dialogRef => {

    dialogRef.viewInit$.subscribe(() => {
        console.log('viewinit complete, all elements displaying to user, go ahead');
    });
    dialogRef.close$.subscribe(() => {
        console.log('dialog closed');
    });
    }
)};
```

Example showing use of dialogRef.viewInit$.subscribe and dialogRef.close$.subscribe

window.PEX.actions$

You can subscribe to actions and then process the action using **ofType**. You can then act on the event or message as in the example shown below.

```
window.PEX.actions$.ofType(window.PEX.actions.RECEIVE_CHAT_MESSAGE)
  .subscribe((action) => {
  console.log('action', action);
    if (action.payload.payload.startsWith('The thing I am looking for')) {
    console.log('found the thing I am looking for in the chat message');
    } else {
      console.log('normal message, ignore');
    }
  });
```

Example showing use of window.PEX.actions$.ofType

Options for `(window.PEX.actions.<ACTION>)` are:

SEND_CHAT_MESSAGE

Triggered when the application user sends a message from the conference.

RECEIVE_CHAT_MESSAGE

Triggered when a message is received into the conference from another participant.

SELECT_EVENT

Triggered when the application user selects an event in the timeline.

PARTICIPANT_CONNECT_SUCCESS

Triggered when another participant connects to the conference.

PARTICIPANT_DISCONNECT_SUCCESS

Triggered when another participant disconnects from the conference.

Plugin state

State can be used to toggle icons and menu names based on information received from the conference or participant. You can alter the text/icon by updating the state.

The example below is from the spotlight plugin. It looks at the participant status to see if the spotlight has been set, and sets the menu icon and label accordingly:

```
// called by the PluginService when plugin is loaded
function load(participants$, conferenceDetails$) {
  participants$.subscribe(participants => {
    var state;
    participants.map(participant => {
      if (participant.spotlight === 0) {
        state = Object.assign({}, state, {
          [participant.uuid]: {
            icon: 'star.svg#star',
            label: 'Spotlight'
          }
        });
      } else {
        state = Object.assign({}, state, {
          [participant.uuid]: {
            icon: 'star-fill.svg#star-fill',
            label: 'Unspotlight'
          }
        });
      }
    });

    if (state) {
      state$.next(state);
    }
  });
}
```

Example from spotlight plugin using plugin state

Troubleshooting common issues

Access blocked to certain sites - CSP issues

If you are accessing external sites / API from your plugin you may run into CSP access issues. CSP can be enabled/disabled on the Conferencing Nodes and there are also basic CSP rules on the Pexip reverse proxy. You may also run into issues with your browser not allowing, for example, mixed secure/insecure connections.

CSP is designed to protect you from various attacks and security issues so careful consideration should be made before changing any CSP rules deployed or turning off CSP.

Application Oops

Because plugins are pulled into the core of the application, errors are treated as application crashes and you will see the standard "Oops" page. You will normally see a back-trace in the browser console letting you know what went wrong. With this in mind, it is advisable to follow good development practices and perform thorough testing before deploying plugins to users, especially if the plugin is to be loaded automatically as this will lead to an instability on every launch of the app.

Migrating from v21 to v22

In v21, the syntax for opening dialogs used Observables, so in order to create a new dialog you had to do something like the following:

```
PEX.pluginAPI
    .openTemplateDialog({
        title: 'Title',
        body: `<div>
                body
                <button id="plugin-button">Btn</button>
               </body>`
    })
    .subscribe(dialogRef => {
        // wait until the dialog dom has been fully initialized
        dialogRef.viewInit$.subscribe(() => {
            //all the code basically goes here

            //And you can subscribe to close event etc
            dialogRef.close$.subscribe(() => {

            });
        });
    });
```

In v22 we moved this mechanism to Promises, so the current implementation looks like this:

```
PEX.pluginAPI
    .openTemplateDialog({
        title: 'Title',
        body: `<div>
                body
                <button id="plugin-button">Btn</button>
               </body>`
    })
    .then(dialogRef => {
        //already after viewInit from previous example so we can execute code directly, don't need to do any additional
stuff here

        //But we can still subscribe to the close event
        dialogRef.close$.subscribe(() => {
            //
        });

        // or use the dialogRef.close function
        document
            .getElementById('plugin-button')
            .addEventListener('click', event => dialogRef.close());
    });
```

If you don't want to add any additional logic after the dialog is open you can skip `then` and it will still work:

```
PEX.pluginAPI
    .openTemplateDialog({
        title: 'Title',
        body: `<div>body</body>`
    });
```

In summary, to migrate your plugin from v21 to v22 you must:

- remove any appeareances of `dialogRef.viewInit$`
- change from `openTemplateDialog(...).subscribe()` to `openTemplateDialog(...).then()`.