Pexip Infinity Connect Plugin Deployment Guide

About this guide

The **Pexip client plugin API** enables you to create plugins that extend the functionality of the Pexip Infinity Connect client applications.

This topic covers:

About Infinity Connect plugins

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

The application provides the plugin with information about the current meeting and selected participants, and the plugin can be used to add buttons to the control menus, toolbars, and context menus that are available to those participants.

The plugin API includes a set of functions that act as a wrapper around requests to the PexRTC JavaScript client API. Your plugins can also send queries directly to the Pexip client REST API, or send requests to external services (such as IFTTT) to gather information or perform other operations.

Prerequisites

Before developing your own plugins you must:

- be proficient with JavaScript and 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.
- be familiar with the basic concepts of WebRTC.
- have a fully operational Pexip Infinity deployment and be familiar with the concepts of the Pexip platform and the Infinity Connect clients.

Getting started

A good way to get started with developing your own plugins is to use the Pexip Branding Portal to create a branding package and to include in that package the set of example plugins offered in the portal. You can then use those examples as a model for your own plugins in combination with the information provided in this guide.

Security considerations

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). As you are in complete control of each HTTPS connection you could perform things like certificate pinning but this is beyond the scope of this guide.

You should 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.

API changes and caveats

The Pexip client plugin API is in continuous development and, as such, features may be changed, added or removed. While 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.

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

Deploying your plugins

Plugins are deployed by creating and uploading a customized Infinity Connect web app that includes your plugin files:

1. Create an Infinity Connect branding/customization package. There are two main ways to get started in doing this:
   - Use the Pexip Branding Portal to create a branding package. The branding portal also contains some example plugins that you can include and use as a model for your own plugins (and which are referred to in this guide).
   - Download the default branding package from the Management Node Administrator interface (via **Services > Web App Customization**).

   Note that the **settings.json** and **manifest.json** files are not included in the default branding ZIP file that is downloaded from the Management Node (and thus you have to create them manually), but they are included in ZIP files generated by the branding portal.
2. Create your plugin files and add them to the branding package as described in Plugin files and structure.
3. Refer to your plugin files in the **plugins** block in the **settings.json** file in the branding package (as described below).
4. Update the **manifest.json** file in the branding package so that **isSettingsBranded** is **true** and that **brandingID** has been modified to a new value (as described below).
   Each time you upload a branding package to Pexip Infinity it must contain a different value for the **brandingID** to ensure that the customization changes are picked up by the Infinity Connect apps.
5. Zip up the revised branding package and upload it to Pexip Infinity via the Administrator interface (**Services > Web App Customization**).

After the plugin has been deployed successfully, the **Plugin** option will appear in each Infinity Connect user's **Settings** page, and clicking on that option will show them a list of available plugins.

For more information about creating and uploading a branding package, see Manually configuring the branding files.

Specifying your plugins in settings.json

The **settings.json** file, which must be saved in the root of the **webapp2** directory, defines which plugins are available within the application. Each plugin must be referenced in the plugins block of the **settings.json** file in your branding package, and is defined by the following fields:

Field	Description
id	A unique id that is used to reference the plugin in the plugin package file and the PEX.pluginAPI.registerPlugin function in the plugin source file.
srcURL	The location of the plugin package file.
enabled	Determines whether the plugin is enabled (loaded) by default: • ***true***: the plugin is enabled (loaded) by default. • ***false***: the plugin is disabled (not loaded) by default. Users can load and unload any of the available plugins themselves if `allowUnload` in the plugin package file is set to true.

The following example extract from a **settings.json** file shows two plugins — a recording plugin that is referenced but is not enabled by default, and a screenshot plugin that is enabled by default:

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

Enabling application branding in the manifest.json file

The **manifest.json** file, which must be saved in the root of the **webapp2** directory, indicates which aspects of the Infinity Connect app have been customized, and it also triggers the app into uploading the latest custom configuration.

- You must set **isSettingsBranded** to **true** for your plugins to be included.
- You must specify a different value for the **brandingID** every time you upload a new customization to the Management Node. The **brandingID** value has to be different from the previously uploaded value for the new settings to take effect. In the **manifest.json** file generated by the Pexip branding portal, the **brandingID** is set to a timestamp, but any type of numerical value is valid.

Here is an example **manifest.json** file:

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

Plugin files and structure

A plugin consists of the following files:

- a package file, which is a JSON file that describes the plugin and how it operates
- a source file, which is a plain JavaScript file that contains your plugin's functions
- optional supporting files such as image files used for buttons and icons

Example plugins

The Pexip Branding Portal contains some example plugins that you can include and use as a model for your own plugins. Many of the examples in this guide are based on the sample plugins from the branding portal. To access the source code for these example 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 want 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 is a **webapp2** folder that contains a **plugins** subfolder containing further subfolders for each of the selected plugins.

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

Plugin package file

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

This is an example of the recording plugin package file:

```
{
  "id": "recording-plugin-1.0",
  "name": "recording",
  "description": "Record the conference through adding a streaming participant",
  "version": 1.0,
  "srcURL": "recording.plugin.js",
  "allowUnload": true,
  "platforms": ["web", "electron", "ios", "android"],
  "participantRoles": ["chair"],
  "menuItems": {
    "toolbar": [{
      "icon": null,
      "label": null,
      "action": "recordConference"
    }]
  }
}
```

The fields required in the package file are as follows:

Field	Description
id	A unique id that is used to reference the plugin in the **settings.json** file and the PEX.pluginAPI.registerPlugin function in the plugin source file.
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 on which to enable the plugin: • ***web***: the web app. • ***electron***: the desktop client. • ***ios***: the mobile client for iOS. • ***android***: the mobile client for Android.
participantRoles	A list of roles that can use this plugin: • ***chair***: available to Hosts. • ***guest***: available to Guests.

Field	Description
menuItems	Determines how and where the plugin appears 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. ⓘ You can only specify one of the above options. You then specify the following fields for nominated menu option: • **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": "recordConference"` in the recording package file calls `function recordConference(participants)` in the recording source file.

Menu item icons

Only SVG images are currently supported 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** when you refer to the icon.

For example, the **crop.svg** file for the screenshot plugin contains `id="crop"`:

```
<svg xmlns="http://www.w3.org/2000/svg" id="crop" viewBox="0 0 24 24" ...
```

which is then referenced in the **icon** field of the screenshot package file as follows:

```
"menuItems": {
    "conference": [{
        "icon": "crop.svg#crop",
        "label": "take screenshot of videos",
        "action": "openScreenshotDialog"
    }]
}
```

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

Menu item labels

The text in the `label` field will be visible to users (either in the conference control menu, or as the hover tip for toolbar items). If you want to dynamically update the label based on state or other information, you should set the `label` field to `null` (as in the recording example) and modify it from within the plugin — see Plugin state.

Plugin source file

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 namespace. It is referenced by the **srcURL** in the plugin package file.

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.

   When the load function is called, it always receives participant and conference details. See Participant and conference properties for full details.
2. **unload**: this performs any cleanup actions when the plugin is disabled.
3. **<your function>** : this is the main body of your code. This function name should be reflected in the **package file** and also in the **registerPlugin** function call (#4 below).

   The function's parameters include the participant and/or conference properties, depending on context (where the plugin appears in the user interface). If it is in the toolbar, then participant and conference properties are available; from the participant roster menu the current participant properties are available; and from the conference control menu the conference properties are available.

For example, `"action": "recordConference"` in the recording package file calls `function recordConference(participants)` in the recording source file.

4. **PEX.pluginAPI.registerPlugin**: this registers the plugin with the PluginService. It should include:
   - the **id** of the plugin (as defined in the Plugin package file)
   - the names of the three functions listed above (i.e. **load**, **unload**, and **<your function>**)
   - **state$** (only required if you are using state to toggle icons or menu names)

   i.e. it follows this model:

```
PEX.pluginAPI.registerPlugin({
    id: '<the id of your plugin>',
    load: load,
    unload: unload,
    <your function>: <your function>,
    state$: state$
});
```

Your source file can also contain additional functions as required.

Here is an example of a complete source file based on the recording plugin that demonstrates the four main functions listed above:

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

    // Init function called by the PluginService when plugin is loaded
    function load(participants$, conferenceDetails$) {
        participants$.subscribe(participants => {
            var state;
            if (
                participants.filter(
                    participant =>
                        participant.uuid === uuid && participant.isStreaming
                ).length > 0
            ) {
                state = {
                    icon: 'stopRecording.svg#stopRecording',
                    label: 'Stop recording'
                };
            } else {
                state = {
                    icon: 'startRecording.svg#startRecording',
                    label: 'Start recording'
                };
            }

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

    // context menu item functions
    function recordConference(participants) {
        if (!connecting) {
            if (
                participants.filter(p => p.uuid === uuid && p.isStreaming)
                    .length > 0
            ) {
                stopRecording(uuid);
            } else {
                startRecording();
            }
        }
    }

    function startRecording() {
        PEX.pluginAPI
            .openTemplateDialog({
                title: 'Record alias',
```

```
                body: `<form>
                      <input id="simpleRecordPluginAlias" class="pex-text-input" placeholder="Type the
alias details here" autofocus />
                      <button class="dialog-button buttons green-action-button" style="margin-top: 40px"
id="recordAliasButton">Start</button>
                    </form>`
            })
            .then(dialogRef => {
                    if (localStorage.pexSimpleRecordPluginAlias) {
                        document.getElementById(
                            'simpleRecordPluginAlias'
                        ).value = localStorage.pexSimpleRecordPluginAlias;
                    }

                    document
                        .getElementById('recordAliasButton')
                        .addEventListener('click', event => {
                            event.preventDefault();
                            event.stopPropagation();

                            const value = document.getElementById(
                                'simpleRecordPluginAlias'
                            ).value;
                            if (!value) {
                                return;
                            }

                            localStorage.pexSimpleRecordPluginAlias = value;
                            var alias = localStorage.pexSimpleRecordPluginAlias;
                            var protocol = 'auto';

                            // const matches = localStorage.pexSimpleRecordPluginAlias.match(/^([^:]+):
(.+)/);
                            // if (matches && ['sip', 'rtmp', 'mssip', 'h323'].indexOf(matches[1]) > -1)
{
                            //     alias = matches[2];
                            //     protocol = matches[1];
                            // }

                            connecting = true;

                            window.PEX.pluginAPI.dialOut(
                                alias,
                                protocol,
                                'guest',
                                value => {
                                    if (value.result.length === 0) {
                                        connecting = false;
                                        document.getElementById(
                                            'simpleRecordPluginAlias'
                                        ).style.border = '2px solid red';
                                        document.getElementById(
                                            'simpleRecordPluginAlias'
                                        ).value = '';
                                        document.getElementById(
                                            'simpleRecordPluginAlias'
                                        ).placeholder = 'check alias and retry';
                                        localStorage.pexSimpleRecordPluginAlias =
                                            '';
                                    } else {
                                        connecting = false;
                                        uuid = value.result[0];
                                        dialogRef.close();
                                    }
                                },
                                {
                                    streaming: true
                                }
                            );
                        });
            });
        // }
    }
```

```
    function stopRecording(uuid) {
        window.PEX.pluginAPI.disconnectParticipant(uuid);
    }

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

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

Participant and conference properties

When the plugin's **load** function is called, it always receives participant and conference details from the PluginService e.g. `function load(participants$, conferenceDetails$)`.

Note that no app properties, such as the selected language, are passed to the plugin.

Conference properties

The following conference properties are available:

Field	Type	Description
locked	boolean	Whether the conference is locked.
guestsMuted	boolean	Whether all Guests are muted.
chatEnabled	boolean	Whether chat messages are enabled.
mediaType	string	The conference's media capabilities: • **video**: the call has video capability. • **audioonly**: the call is audio-only. • **none**: control-only participants are connected.
started	boolean	Whether the conference has started.

Participant properties

The following participant properties are available:

Field	Type	Description
uuid	string	The UUID of this participant, to use with other operations.
name	string	The display name of the participant.
role	string	The level of privileges the participant has in the conference: • "chair": the participant has Host privileges • "guest": the participant has Guest privileges
uri	string	The URI of the participant.
startTime	number	A Unix timestamp of when this participant joined (UTC).

Field	Type	Description
protocol	string	The call protocol. Values: "api", "webrtc", "sip", "rtmp", "h323" or "mssip". (Note that the protocol is always reported as "api" when an Infinity Connect client dials in to Pexip Infinity.)
isExternal	boolean	Boolean indicating if it is an external participant, e.g. coming in from a Skype for Business / Lync meeting.
presenting	boolean	Indicates if the participant is the current presenter.
muted	boolean	Indicates if the participant is administratively muted.
canControl	boolean	Set to true when the participant is a Host and it is a VMR or Virtual Auditorium service.
canDisconnect	boolean	Indicates if the participant can be disconnected.
canTransfer	boolean	Indicates if the participant can be transferred into another VMR.
canMute	boolean	Indicates if the participant can be muted.
serviceType	string	The service type: • "connecting": for a dial-out participant that has not been answered • "waiting_room": if waiting to be allowed to join a locked conference • "ivr": if on the PIN entry screen • "conference": if in a VMR • "lecture" if in a Virtual Auditorium • "gateway": if it is a gateway call • "test_call": if it is a Test Call Service
callType	string	The type of call: "video", "audio" or "api".
spotlight	number	A Unix timestamp of when this participant was spotlighted, if spotlight is used.
isStreaming	boolean	Indicates whether this is a streaming/recording participant.
vad	number	Audio speaking indication. 0 = not speaking, 100 = speaking.
stageIndex	number	The index of the participant on the "stage". 0 is most recent speaker, 1 is the next most recent etc.
rxPresentation	boolean	Indicates if the participant is administratively allowed to receive presentation.
feccSupported	boolean	Indicates if this participant can be sent FECC messages.
buzzTime	number	A Unix timestamp of when this participant raised their hand, otherwise zero.

User interfaces

Your plugin may not necessarily require a user interface (for example if it just listens to events as in window.PEX.actions$).

However, if you need to display a dialog then you should use the openTemplateDialog call. 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 functions

The plugin API includes a set of functions that are available on the `window.PEX.pluginAPI` object that act as a wrapper around requests to the PexRTC JavaScript client API.

Currently, only a subset of the PexRTC functions are available through the `window.PEX.pluginAPI` object. However, you can use the sendRequest plugin API function to send queries directly to the Pexip client REST API to perform other operations.

registerPlugin

Called by your plugin to register with the application. See the recording plugin for an example.

createNewState

Creates states — see Plugin state.

sendRequest

This function lets you send conference and participant control commands into the conference via the Pexip client REST API. This provides access to a range of functions that are not supported directly via the plugin API.

To perform conference-level commands, such as locking/unlocking the conference or muting all guests, your requests should follow these examples:

`PEX.pluginAPI.sendRequest('/lock');` (locks the conference)
`PEX.pluginAPI.sendRequest('/muteguest');` (mutes all Guests)

To perform participant-level commands you have to pass the **/participants/<participant_uuid>/<request>** part of the request in your call to sendRequest. For example, to mute an individual participant you need something like this:

```
var muteparticipant = "/participants/" + participant.uuid + "/mute";
PEX.pluginAPI.sendRequest(muteparticipant);
```

If the command requires a playload then you include the payload as a second argument. For example, to send DTMF tones of 1234 to a participant:

```
sendRequest('/participants/<participant_uuid>/dtmf', {digits: "1234"})
```

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 create inline JavaScript in this template (using, for example, `onClick` methods on buttons). 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.

The syntax for calling dialogs changed between v21 and v22. For more information, see Migrating plugins from v21 to v22 or later.

This example shows the use of `onClick` methods in the openTemplateDialog function:

```
PEX.pluginAPI.openTemplateDialog({
  title: 'Layout control',
  body: `<div style="flex-wrap: wrap; justify-content: center;">
    <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
({})">Default</button>
  </div>`
});
```

This example shows how to perform your actions in a `.then`, and how to subscribe to `dialogRef.close$` if required:

```
function doSomething(conferenceDetails) {
  PEX.pluginAPI.openTemplateDialog({
  title: 'Title',
  body: '<button id="myButtonId">Do Something</button>'
  })
  .then(dialogRef => {
    //Do something
    console.log('doing some stuff');

    //Subscribe to the close event, if required
    dialogRef.close$.subscribe(() => {
      console.log('dialog closed');
    });
```

```
  }
)};
```

spotlightOnParticipant

Takes a participant UUID and sets the spotlight e.g. `PEX.pluginAPI.spotlightOnParticipant(participant.uuid);`

spotlightOffParticipant

Takes a participant UUID and removes the spotlight e.g. `PEX.pluginAPI.spotlightOffParticipant(participant.uuid);`

dialOut

Initiates a call from the current conference to a participant. Takes an `alias`, `protocol` and `role`, and provides a participant UUID if successful. See the branding portal's recording plugin for a usage example.

disconnectParticipant

Takes a participant UUID and disconnects them e.g. `PEX.pluginAPI.disconnectParticipant(participant.uuid);`.

changeConferenceLayout

This function is being deprecated — you should use transformConferenceLayout instead.

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 PexRTC client 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 e.g. `PEX.pluginAPI.sendChatMessage('Here is some chat');`.

Observables / promises

Dialogs use promises and actions$ are observables.

dialogRef.close$

Subscribing to this allows you to perform actions when the dialog has been closed, for example:

```
dialogRef.close$.subscribe(() => {
  console.log('dialog closed');
});
```

See openTemplateDialog above for a complete example of how to manage dialogs.

Note that in v21 the syntax for opening dialogs used observables instead of promises; see Migrating plugins from v21 to v22 or later for more information.

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');
    }
  });
```

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

<ACTION>	Trigger
SEND_CHAT_MESSAGE	The application user sends a message from the conference.
RECEIVE_CHAT_MESSAGE	A message is received into the conference from another participant.
SELECT_EVENT	The application user selects an event in the timeline.
PARTICIPANT_CONNECT_SUCCESS	Another participant connects to the conference.
PARTICIPANT_DISCONNECT_SUCCESS	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.

If using state, you must include it in the **registerPlugin** function.

The example below is from the recording plugin. It looks at the participant status to see if any participant has its **isStreaming** property set, and assigns the icon and label accordingly to either stop or start recording:

```
// Init function called by the PluginService when plugin is loaded
function load(participants$, conferenceDetails$) {
    participants$.subscribe(participants => {
        var state;
        if (
            participants.filter(
                participant =>
                    participant.uuid === uuid && participant.isStreaming
            ).length > 0
        ) {
            state = {
                icon: 'stopRecording.svg#stopRecording',
                label: 'Stop recording'
            };
        } else {
            state = {
                icon: 'startRecording.svg#startRecording',
                label: 'Start recording'
            };
        }

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

It also includes **state$** in its **registerPlugin** function:

```
PEX.pluginAPI.registerPlugin({
  id: 'recording-plugin-1.0',
  load: load,
  unload: unload,
  recordConference: recordConference,
  state$: state$
});
```

Troubleshooting common issues

This section contains a few common issues you may encounter when developing your plugin.

Plugins, or updates to your plugins, are not appearing in the Infinity Connect client

A failure to see your plugin, or changes to your plugin, typically occurs when:

- There is an incorrect JSON structure in your **settings.json** file, particularly if you have multiple plugins specified.
- You have not updated the **brandingID** in the **manifest.json** file and so the branding package is not being applied to the apps.

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, you should follow good development practices and perform thorough testing before deploying plugins to users, especially if the plugin is to be loaded automatically as this could lead to an instability on every launch of the app.

Migrating plugins from v21 to v22 or later

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

```
PEX.pluginAPI
    .openTemplateDialog({
        title: 'Title',
        body: `<div>
                <button id="plugin-button">Btn</button>
               </div>`
    })
    .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 this mechanism moved to Promises, so the current implementation looks like this:

```
PEX.pluginAPI
    .openTemplateDialog({
        title: 'Title',
        body: `<div>
                <button id="plugin-button">Btn</button>
               </div>`
    })
    .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>...</div>`
    });
```

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

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

Note that v21-style Observables for opening dialogs are still supported in v22 or later, but we strongly recommend that you use / migrate to Promises as soon as practicable.