ChatGPT

Adobe Photoshop UXP Plugin Development (v26.7) – Guide Overview and Key Capabilities

UXP Guide Documentation for Photoshop Plugins

The UXP developer documentation for Photoshop (version 26.7, corresponding to the 2022 UXP guides) includes a comprehensive set of guides for plugin development. In total, there are roughly **35 documents** under the UXP plugin development section. These cover everything from getting started tutorials and manifest files to advanced topics like distribution, toolchain, and debugging [1]. (For example, the guides span topics such as *Quickstart tutorials*, *Manifest v5*, *Code Samples*, *Hybrid Plugins (UXP & C++)*, *File Access*, *Network I/O*, *"How Do I..." snippets*, *Theme Awareness*, *Debugging*, and more [2].) This extensive documentation ensures developers can find guidance on every aspect of building Photoshop UXP plugins.

Building a Dockable Panel UI

Panel Entrypoint: UXP allows plugins to create **dockable panel** interfaces (non-blocking UI) that behave like native Photoshop panels. In the plugin's `manifest.json`, you define an entrypoint of type `"panel"` with an identifier, label, icons, and size constraints. For example: `"entrypoints": [{ "type": "panel", "id": "panelName", "label": { "default": "Panel Name" }, "minimumSize": {...}, "preferredDockedSize": {...}, "icons": [...] }]` [3]. This makes the plugin appear as a panel that the user can dock, resize, open, or close at will – providing a seamless UX in the Photoshop workspace [4].

UI Framework: Inside the panel, you build the UI using HTML/CSS and JavaScript (the UXP runtime provides a subset of web capabilities). Adobe recommends using the **Spectrum UXP components** for a consistent look and feel. Spectrum components (or Spectrum CSS) come with Photoshop's theming support, so your panel will automatically adapt to light/dark UI themes [5]. You can use standard form elements (buttons, checkboxes, dropdowns, etc.), or even frameworks like React (the guides include a "Working with React" tutorial). The **UXP Developer Tool** can create a starter panel plugin and live-reload it for development [6].

Proposed Panel UI Structure: For this plugin's functionality, a logical panel layout might be:

- *Header:* Plugin title or icon (as set in the manifest) and maybe a refresh or settings gear (often via a fly-out menu [7] for extra options).
- *Script Selection:* A dropdown or list box to **load and select a script file** from a directory. For example, an HTML `<select>` or list of filenames that the plugin has detected (more on file access below).
- *Layer Selection:* A scrollable section listing **group layers (folders)** in the active document, each with a checkbox. This lets the user mark which layer groups to target. (For good UX, you could display the layer group names, and possibly indent child groups if needed. UXP doesn't have a built-in tree control, so a simple list with hierarchical naming or prefix would do, or multiple lists if necessary.) Each item's checkbox can be toggled to select/unselect that layer group for processing.

- *Actions:* Buttons for “Run Script on Selected Layers” and perhaps “Batch Process Files”. For example, a **Run** button that executes the chosen script on the selected layers of the current document, and another button or toggle to **Process Multiple Files** which triggers batch automation across files (if implemented).
- *Status/Log Area:* (Optional) A small text area or status line to show feedback (e.g. “Completed processing layer X” or errors). This isn’t mandatory, but can improve clarity for the user during long operations.

Using this structure, the panel remains dockable and non-modal, meaning the user can interact with Photoshop (e.g. adjust the canvas or layer selection) while the panel is open [8]. This is ideal for a plugin that works with layer selections. Ensure the panel UI is responsive (UXP panels can be resized), and consider grouping or disabling controls contextually (for instance, disable the Run button if no script is selected or if no layers are checked).

File Access – Loading Script Files

Plugin Packaged Scripts: If the plugin comes with its own automation scripts, they can reside in the plugin’s installation folder (the “plugin home” directory). Files in the plugin’s home are **read-only** at runtime but can be freely accessed by your code [9]. You might organize them in a subfolder (e.g. a `scripts/` directory). To load such a script, you can simply use the JavaScript module system: UXP supports `require()` for including JS files. For example, if you have a script file `myScript.js` bundled with your plugin, you can do: `const myScript = require("./scripts/myScript.js");` and then call its exported functions. (The documentation provides a snippet for how to structure modules and use `require` to include them [10].) This is a straightforward way to load known script files – you can then populate your UI’s dropdown with a static list of these script names.

User-Provided Scripts (File System Access): If you want to allow users to select a directory of script files (for instance, if the plugin should run user-authored `.jsx` or `.js` files), UXP provides a secure file system API via `require('uxp').storage.localFileSystem`. Direct file system access is sandboxed for security, but the plugin can prompt the user to pick files or folders. For example, you can call `await localFileSystem.getFolder()` to show a folder picker dialog [11]. If the user selects a folder (say containing script files), your code receives a `Folder` object. From this, you can read its entries: `const entries = await folder.getEntries();` which gives an array of `Entry` objects (files or subfolders). You would then filter these for the script files you care about (e.g. by extension or naming convention). Those file names can populate your UI list. When the user chooses one, you might store the corresponding `Entry` object for later execution.

Keep in mind that **user permission is required** to access arbitrary files. UXP’s design is such that a plugin must call a picker function (like `getFolder()` or `getFileForOpening`) – it cannot just read an arbitrary path on disk without consent. Once the user grants access through the picker, UXP returns a token or handle you can use to work with that file/folder [12]. For example, `getFolder()` returns a `Folder` handle that is valid for the session. If you need to persist access (e.g. remember the folder next time the plugin runs), you can convert it to a persistent token, but for simplicity you might just ask each time or keep it for the plugin session.

Listing and UI Integration: After obtaining the folder and filtering entries, you likely will have an array of `File` entries for scripts. You can then display these in the UI (perhaps as a list of file names). The user’s

selection in the dropdown/list should correspond to one of these `File` objects behind the scenes. It's a good practice to store the selected file entry (or its path) in your plugin's state when the user picks it. Also consider providing a **refresh** mechanism – if the contents of the directory change or if the user wants to pick a different folder, your UI should handle that (e.g. a "Choose Folder" button or a refresh icon next to the script list).

Layer Selection – Selecting Group Layers in the Document

Accessing Layers via Photoshop API: Adobe's UXP Photoshop API provides a DOM-like interface through `require('photoshop').app`. You can get the currently active document with `app.activeDocument`, and from there access its `layers` collection. The `Document.layers` collection contains the **top-level layers** of the document (both individual layers and layer groups) [13]. This collection behaves like an array – for example, you can iterate `app.activeDocument.layers.forEach(layer => ...)` to inspect each layer's name or type. Each **Layer** object has properties like `name`, `id`, and `kind`. In particular, `layer.kind` tells you the layer type (e.g. normal pixel layer, text layer, **group**, etc.). For group folders, `layer.kind` will equal `LayerKind.GROUP` (a constant in `photoshop.constants`) [14]. Moreover, a group Layer object will have its own `layer.layers` property which is a Layers collection of its children. This means you can recurse into group layers to get sub-layers if needed. (A non-group layer will have `layer.layers == null` or simply not be iterable.)

To **list group layers** for user selection, you might restrict to top-level groups initially for simplicity. For each `layer` in `app.activeDocument.layers`: check if `layer.kind === constants.LayerKind.GROUP`. If so, include it in your list. If you also want to include nested groups, you can traverse recursively (the documentation and community examples show patterns for this – e.g., a recursive function that goes into `layer.layers` when a group is encountered [14]). The end result could be a flat list of all group layers (or you could maintain a hierarchy in the UI if that's preferable). Each layer has a `name` property which you can display in the UI. Keep track of the actual Layer object (or at least some identifier like the layer ID) for each list entry so you know which layer the user has chosen.

UI for Layer Selection: As mentioned, using a list of checkboxes next to layer names is a user-friendly way to allow multi-selection. You can dynamically create a checkbox for each group layer name when the panel opens or when the document changes. (Note: you may want to update the list if the user opens a different document or if layers change – listening to Photoshop events for document switch or layer changes is more advanced, but at minimum you could add a "Refresh Layers" button to re-scan the document's groups.) When a user checks or unchecks an item, simply update your internal selection state. This might be an array of selected Layer objects or their indices.

Programmatic Selection vs. Plugin UI Selection: It's important to distinguish between the user selecting layers **within Photoshop's Layers panel** versus selecting them in your **plugin's UI**. The approach above uses the plugin's own interface (checkboxes) to mark target layers. This does not necessarily change which layers are "active" in Photoshop itself; it's an independent selection for the plugin's purposes. This is often fine – you don't need to actually change Photoshop's layer selection. However, UXP does allow you to programmatically select layers in Photoshop if needed. Each Layer object has a boolean property `layer.selected` that you can set to `true` or `false` [15]. Setting `myLayer.selected = true` will add that layer to Photoshop's selection (and you may want to set others to false if you intend to exclusively select one layer [16]). For multiple selection, you could set multiple layers' `.selected = true`. In practice,

you might not need to do this at all – it's usually better to operate on layers via the API directly – but it's good to know this capability exists (for example, if you wanted your plugin to highlight layers as the user clicks on them in the plugin UI). In summary, the plugin can gather the target layers through the Photoshop API and allow the user to toggle which ones to include or exclude, without disrupting the Photoshop UI's own layer selection unless you explicitly want to.

Executing Scripts/Actions on Selected Layers

Once you have a set of target layers (the group layers the user checked off) and a chosen script or action, the core of your plugin is to **execute the automation on each selected layer**. There are a few key parts to this:

1. Photoshop DOM Operations: The Photoshop API exposes many DOM-like methods and properties to manipulate layers. Depending on what your "script" needs to do, you might call these methods. For example, you can change layer properties (opacity, blend mode, etc.), or call functions like `layer.scale()`, `layer.rotate()`, `layer.flatten()`, `layer.duplicate()`, etc., if available. In the documentation's examples, they show operations like scaling layers by a certain percentage [17]. If your script files contain actual JavaScript code (functions) that use the Photoshop API, you can invoke those. For instance, if a script file exports a function `applyEffects(layer)`, you would call `applyEffects(layer)` for each selected layer. Because you required the script module earlier, you have those functions available in your plugin code.

2. BatchPlay for Actions or Advanced Operations: Not everything is covered by simple DOM methods. For more complex or newer features, or to execute a recorded Photoshop Action, you might use the **Action API** via `require('photoshop').action.batchPlay()`. BatchPlay lets you send low-level action descriptors (the same kind of commands that underlie Photoshop's scripting and actions). For example, to run a specific Action from the Actions panel, you could batchPlay a "select" or "play" command with the action name and action set. (This requires knowing the Action's internal identifiers or names.) Similarly, if your script needs to perform something not yet in the DOM (for example, a filter or an adjustment), you might call it via batchPlay. BatchPlay returns promises and can execute multiple steps in one go. This is an advanced capability, but it's available if needed. (As an example, the documentation shows using `batchPlay` to select layers by name [18] – the same idea can be used to trigger menu commands or actions by their IDs.)

3. Modal vs Non-Modal Execution: A critical consideration when changing a Photoshop document is to use **modal execution** properly. Photoshop's UXP API is "promise-based" and many operations are asynchronous. To avoid race conditions and to allow Photoshop to manage undo states, UXP introduced the concept of running operations in an **executeAsModal** block. In practice, this means wrapping your sequence of edits in a function and calling `require('photoshop').core.executeAsModal(yourFunction)`. Within that function, you can perform DOM changes to the document. Photoshop will treat them as a single transaction (in terms of undo and performance) and will not allow other UI actions to interfere while it runs [19]. For example, you might implement a function `processSelectedLayers()` that, given the list of layers and the chosen script, iterates through each layer and applies the script's logic. You would call:

```
await require('photoshop').core.executeAsModal(processSelectedLayers);
```

Inside `processSelectedLayers(executionContext) { … }`, you loop over the layers. During this execution, Photoshop is in a modal state (the user can't interact with the document until it finishes, which is usually quick). This approach is recommended for any operation that changes the document's structure or layer pixels [20] – it yields better performance and stability. (If your script actions are read-only – e.g. just reading layer names or counting layers – you wouldn't need modal, but most automation does make changes.)

4. Looping Through Layers: In the modal function (or outside if not using modal), you will **iterate over each selected layer**. For each layer, perform the desired action. For instance, if the user selected three group layers and a script "Export Group as PNG", you would for each layer call your export function. A pseudo-code might look like:

```
for (const layer of selectedLayers) {
    // If needed, target the layer in Photoshop (not always required to do
explicitly)
    // Run the script's main function on the layer
    await myScript.run(layer);  // assume myScript.run performs the desired
actions on that layer
}
```

The UXP API allows standard looping constructs. The documentation's examples demonstrate iterating over layers – e.g., scaling layers whose name matches a criteria [17]. In our case, the criterion is that the layer is in the selected list. Ensure you use `await` for any asynchronous operation in the loop (most Photoshop API calls that modify the document are async). This will sequence them one after the other. If the operations are independent and you wanted to attempt parallel, you *could* spawn promises, but generally it's safer to do them sequentially for simplicity and to avoid overwhelming Photoshop.

After looping, you may want to give the user feedback – e.g., "Done! Processed 3 layers." This could be a simple `console.log` (which appears in the Developer Tool console) or a message in the panel UI.

Batch Processing Multiple PSD Files

The plugin's functionality can extend from working on one open document to **processing multiple Photoshop files** in a batch. This involves file handling and looping at the document level:

1. Selecting Files or Folders: Provide a way for the user to specify the set of PSD files. Common approaches: a "Select Folder" button (to process all PSDs in a folder) or a "Select Files" dialog. Using `localFileSystem.getFileForOpening({ allowMultiple: true, types: ... })` will let the user pick multiple files at once [21]. Alternatively, `getFolder()` (discussed earlier) can get a directory, and then you filter `folder.getEntries()` for `.psd` files. Once you have a list of file entries (or file paths), these represent the documents to process.

2. Opening Documents: To process a file, it needs to be open in Photoshop. The UXP Photoshop API allows you to open files by path or by using the file handle. You can use `app.open()` with a file path string [22] – for example, `await app.open("/path/to/file.psd")` returns a Document object. Even better, since

you might have a `File` entry (from the file picker) already, you can pass that to `app.open()` directly. (If you pass a UXP File entry object, Photoshop will automatically create a session token for it to open, per documentation [23].) So in code, you might do:

```
for (const fileEntry of fileList) {
    const doc = await app.open(fileEntry);  // opens the PSD, returns Document
    // ... process ...
}
```

This will sequentially open each file. *Tip:* If the files are large or numerous, you might want to open one, process, then close it before moving to the next, to avoid having many large files open at once.

3. Detecting Layers by Name: The question mentions *detecting layer names* across files – presumably to apply the script only to certain layers in each document (e.g. a batch operation where you always target layers named "Logo" in every file). There are two ways you might approach layer targeting in batch mode:

- **Use the same selections as the current document:** If the user has selected some group layers in the UI (from the currently open document), you could take their names (or relative positions) as the pattern. Then for each opened document in the batch, find the layers with those names. For example, if the user checked `Header` and `Footer` groups in Document A, for Document B you'd do `docB.layers.getByName("Header")` (if the API supports `getByName`) or iterate layers to find a layer named "Header" [13] [24]. The Photoshop DOM **does** provide a `layers.getByName(name)` method to retrieve a layer by its name [24]. This can simplify lookup (keeping in mind if duplicate names exist, getByName might return the first match). Once you identify the corresponding layers in the new document, you can run the same script on them.

- **Use a predefined name or criteria:** If the batch operation is not based on what the user selected in a single document, it could simply be hard-coded (e.g., "for each file, find the 'Watermark' layer group and run script X on it"). In that case, you already know the target layer names or other criteria. You would still open each document, then search within `doc.layers` (and sub-layers if needed) for layers matching those names.

In either approach, **the mechanics are**: open document -> get its relevant Layer objects -> apply script to those layers -> optionally save/close document.

4. Applying Scripts to Each Document's Layers: This is essentially the same as the single-document case, but nested one level up. You will loop over documents, and inside that loop, possibly loop over layers of interest. It's wise to utilize `executeAsModal` here as well. For example, you might wrap the entire processing of one document in an executeAsModal call (especially if you plan to do multiple layer edits within). This could ensure all changes in that document are one undo step. You could even encompass the *whole* batch in one modal execution, but that may not be ideal if you are opening/closing files – better to handle each file individually in case something goes wrong (so you don't lock Photoshop for too long in one giant modal operation). For instance:

```
for (const file of filesToProcess) {
  await app.open(file);
  const doc = app.activeDocument;
  await core.executeAsModal(async () => {
    // find target layers in doc
    let targets = findLayersByName(doc, targetNames);
    for (const layer of targets) {
      await runScriptOnLayer(layer);
    }
    await doc.save();  // save changes
  });
  await doc.close();  // close the file after saving
}
```

In the above pseudo-code, `findLayersByName` would implement the name-matching logic (using something like `doc.layers.getByName` or manual search), and `runScriptOnLayer` encapsulates the automation for a single layer (could be the same function used in single-doc mode). We also call `doc.save()` and `doc.close()` for each file – the DOM provides `document.save()` (which will save to the existing path without prompting in most cases) [25] and methods to close documents with or without saving [26] [27]. Ensure you handle unsaved changes appropriately – you can specify save options when closing to suppress prompts [26].

5. User Experience for Batch Operations: When implementing batch mode, communicate with the user. If processing many files, consider showing a progress indicator or at least a message ("Processing file 3 of 10..."). Because UXP panel UIs do not update during a modal operation (the UI thread is paused while the modal task runs), you might need to break out of modal or use asynchronous techniques for progress. One simple approach: process files one by one *without* wrapping the entire batch in a single modal call – Photoshop will allow some UI refresh between modal calls. You could even update a text label in the panel after each file (since after each executeAsModal returns, your panel code can run to update the DOM). Another approach is to use the `executeAsModal` with the `executionContext.reportProgress()` function (if available) to periodically report status. But for many plugins, a simpler update-per-file is sufficient.

Finally, after batch processing, you might pop up an alert or message saying "Batch complete" – the Photoshop API provides a simple alert method (`require('photoshop').core.showAlert({message: '...'});`) [28] which is a quick way to notify the user.

Key Modules and Considerations for Extensibility

To summarize, the key UXP APIs and components involved in this plugin would be:

- **UXP Manifest & Entry Points:** Define a Panel type entry in `manifest.json` (possibly alongside a command if you want a menu item, but not necessary for a pure panel). Include panel metadata like icon assets for a professional touch [29]. Keep the manifest up-to-date (Photoshop 26.7 might require

manifest version 5 – check *Photoshop manifest* guide for any Photoshop-specific manifest additions [30]).

- **Photoshop DOM API (`photoshop` module):** Provides `app`, `app.activeDocument`, `Document` and `Layer` objects, layer collections, etc. Use this for all layer and document operations (listing layers, checking `layer.kind`, setting `layer.selected`, calling methods like `layer.scale()`, `document.save()`, etc.) [31] [17]. The `photoshop.constants` sub-module gives you enumerations like `LayerKind.GROUP` to identify groups [14]. Also `photoshop.core` for `executeAsModal` and alerts.

- **UXP Storage APIs (`uxp.storage` module):** Use `localFileSystem` for file/folder pickers and file I/O. Relevant functions: `getFolder()` to choose a directory [11], `Folder.getEntries()` to list files, `File.read()` or requiring modules for loading script content, and possibly `getFileForOpening()` for file selection dialogs [21]. Also note methods like `getTemporaryFolder()`, `getDataFolder()`, `getPluginFolder()` which you might use if you need to store data or read plugin-packaged files [32] – e.g., `getPluginFolder()` returns your plugin's installation folder (home) for reading packaged assets.

- **UI Components:** No separate module needed if using HTML, but to maintain consistency consider using Spectrum Web Components (e.g. `<sp-button>`, `<sp-checkbox>`, `<sp-dropdown>`). These are included in UXP and just need the Spectrum CSS/JS to be referenced (the UXP docs have a Spectrum reference link [33]). They ensure your plugin respects Photoshop's theming and design guidelines out of the box.

- **Event Handling:** Your panel UI will use standard DOM event listeners for button clicks, checkbox toggles, etc. UXP supports a subset of DOM; for example, you can use `document.getElementById()` in your HTML panel and attach `.onclick` or `addEventListener('change', …)` for form elements. Keep your UI responsive by not blocking the main thread – long operations should be awaited (as described with executeAsModal).

Extensibility Considerations: To ensure the plugin is extensible and maintainable, structure your code in modular functions: one for loading scripts, one for refreshing layer list, one for performing actions on a layer, one for batch processing, etc. This not only makes it easier to update each piece, but also allows re-use (for instance, the same `runScriptOnLayer` function can be used whether you're processing one document or many).

Allowing the plugin to adapt to different scripts and layer names is a key part of extensibility. You might design a simple **interface for script files**, e.g., each script module exports a standard function like `apply(layer)` or `run(layer)` so that your plugin knows how to call it uniformly. That way, adding a new script file to the folder automatically makes it available in the UI and it will execute correctly as long as it follows the expected pattern. This is much cleaner than handling each script's logic inside the plugin code.

User Interaction & Clarity: Make sure to guide the user through the workflow. For example, if the plugin requires selecting a folder of scripts first, handle that before enabling layer selection or the run button. You could default to a known location (like a subfolder in plugin home) but still provide an option to change it. Label your buttons clearly ("Run Script on Layers" vs "Batch Process Files…"). If an error occurs (say a script

tries to operate on a layer that doesn't exist in a batch file), handle it gracefully – possibly skip that file with a warning message rather than crash the whole process. Using `console.error` can help log issues to the Developer Tool console for debugging [34].

Finally, test the plugin with different scenarios: no layers selected (should probably warn or do nothing safely), no script selected, huge documents, etc., to ensure it remains robust. Thanks to the comprehensive UXP documentation and APIs, you have the building blocks to implement all these features. By combining a well-designed panel UI with the powerful Photoshop DOM/actions and secure file access, your plugin can provide a clear, extensible interface for automating layer-specific tasks across one or many files in Photoshop.

Sources: The information above is based on Adobe's UXP Photoshop developer guides and API references for version 26.7, including guidance on panel UI design [4] [3], file I/O APIs [11] [12], layer access and selection via the DOM [13] [15], as well as code examples for iterating over layers and documents [17] [22]. These official documents outline the necessary components and best practices for building a plugin with the described functionality.

---

[1] [6] [7] [20] [30] [33] Getting Started—UXP for Adobe Photoshop
https://developer.adobe.com/photoshop/uxp/2022/guides/

[2] [9] [10] [12] [28] [34] How Do I...
https://developer.adobe.com/photoshop/uxp/2022/guides/how-to/

[3] [29] Manifest File Structure - UXP Manifest
https://developer.adobe.com/photoshop/uxp/2022/guides/uxp_guide/uxp-misc/manifest-v4/

[4] [5] [8] Design - Designing for Photoshop
https://developer.adobe.com/photoshop/uxp/2022/design/ux-patterns/Designingforphotoshop/

[11] [21] [23] [32] developer.adobe.com
https://developer.adobe.com/photoshop/uxp/2022/uxp-api/reference-js/Modules/uxp/Persistent%20File%20Storage/FileSystemProvider/

[13] [24] Layers
https://developer.adobe.com/photoshop/uxp/2022/ps_reference/classes/layers/

[14] Get layer list from layers inside of a group? - UXP Plugin API - Adobe Creative Cloud Developer Forums
https://forums.creativeclouddeveloper.com/t/get-layer-list-from-layers-inside-of-a-group/5002

[15] [16] [18] How to select a layer? - UXP Plugin API - Adobe Creative Cloud Developer Forums
https://forums.creativeclouddeveloper.com/t/how-to-select-a-layer/3020

[17] [19] [31] Photoshop API—UXP for Adobe Photoshop
https://developer.adobe.com/photoshop/uxp/2022/ps_reference/

[22] [25] [26] [27] Document
https://developer.adobe.com/photoshop/uxp/2022/ps_reference/classes/document/