Browser Extensions Course

Flavio Copes

Updated August 3, 2026

Contents

Build a secure Manifest V3 browser extension with a popup, extension storage, content scripts, messaging, a service worker, keyboard commands, tests, and a release package.

What you'll learn: Build, test, secure, and package Page Notes, a local-first extension that saves and displays a note for the active web page.

Extension foundations

Define the project, create Manifest V3, load it unpacked, and understand its contexts.

Choose an extension problem

Decide when browser integration is necessary and define a Page Notes project with a narrow, understandable scope.

A browser extension can observe or change browser behavior beyond an ordinary website. That extra authority must solve a real browser-shaped problem.

Page Notes will save a short note for the current page URL, show it in a toolbar popup, and optionally highlight the page. It does not need browsing history, every website at install time, or a remote server.

Start from the moment normal web code becomes insufficient. A website cannot add a toolbar action, read the user's currently active tab after an extension gesture, or inject a packaged script into an unrelated page. Those are valid reasons for an extension. "It is easier to install" is not.

Define the authority budget beside the feature. Saving notes needs extension-owned storage. Reading the selected tab after the user opens the action can use `activeTab`. Highlighting needs `scripting` plus temporary access to that tab. None of these requires persistent access to every site or a backend that receives browsing data.

Also define where data stops. A full URL can contain private paths, search terms, or tokens in its query and fragment. Decide whether the note key should use the complete URL, remove the fragment, or normalize selected parameters. The safest answer depends on the product, but it must be explicit.

Write the journey from opening the toolbar action to saving and showing a note. For each step, name the extension context, permission, data read, data written, retention rule, and visible user gesture. Remove any capability that cannot be tied to a step.

Create the Manifest V3 file

Declare the extension identity, version, popup action, permissions, and packaged files in a minimal manifest.json.

Every extension has a root `manifest.json`. It tells the browser what the package contains and which capabilities it requests.

Manifest V3 is the current Chrome extensions platform. Begin with required metadata and the action popup. Add permissions only when a later lesson implements the feature that uses them.

The manifest is both configuration and a trust declaration. API permissions such as `storage` and `scripting` grant extension capabilities. Host permissions grant access to matching origins and may produce user warnings. `activeTab` is different: it grants temporary host access after a qualifying user gesture rather than persistent access at installation.

```
{
  "manifest_version": 3,
  "name": "Page Notes",
  "version": "1.0.0",
  "description": "Save a note for the current page",
```

```
  "action": { "default_popup": "popup.html" },
  "permissions": ["storage", "activeTab", "scripting"]
}
```

The example declares the end-state permissions so you can see them together. In the project, begin with only the popup, then add each permission with the feature that consumes it. This makes unused authority obvious in code review. Keep the version compatible with Chrome's one-to-four-part numeric format and increment it for every published package.

Create a clean project folder with the manifest and an empty popup page. Load it once with no permissions, then add `storage`, `activeTab`, and `scripting` one at a time. For each addition, write down the exact API and user-facing behavior it enables.

Load and reload the unpacked extension

Install the development folder locally, inspect manifest errors, and understand when a code change requires a reload.

Chrome can load a local folder as an unpacked extension, which gives a fast edit–reload–inspect loop.

Open `chrome://extensions`, enable Developer mode, choose Load unpacked, and select the folder containing the manifest. Manifest, service-worker, and content-script changes normally require reloading the extension; popup HTML may be reopened after a change.

There are several caches of reality during development. Reloading the extension installs the new package and restarts its extension contexts. A tab that already has a declaratively registered content script may still contain the old injected code until the page itself reloads. An open popup is a separate document and disappears when it loses focus.

After a manifest or worker change, reload the extension, then reload the test tab. After a popup-only change, close and reopen it. Check the extension card's Errors button before assuming a missing log means the code never ran. An invalid manifest can prevent the package from loading at all.

Keep a fixed test page and a tiny change marker, such as the extension version in a debug log, so you can prove which code generation is running. Otherwise an old content script can make a correct fix look broken.

Load and pin Page Notes. Change the popup, content script, service worker, and manifest separately, recording the minimum reload sequence for each. Deliberately break one manifest key and one content-script syntax line and locate both errors.

Understand extension contexts

Assign user interface, page access, browser events, and privileged API calls to popup, content script, and service worker contexts.

An extension is several JavaScript environments, not one page with universal access.

The popup exists only while open. A content script runs in a web page with an isolated JavaScript world but shared DOM. The extension service worker handles browser events and may stop when idle. They communicate with messages and shared extension storage.

Each context owns different work. The popup owns short-lived user interaction. The content script is the only project context that touches the host page DOM. The service worker owns browser events

such as commands and installation, but it has no DOM and must survive being terminated between events.

Isolation is not a complete trust boundary. A content script cannot directly read the page's JavaScript variables, yet both worlds can read and mutate the same DOM. Values taken from page markup are untrusted. Extension storage is shared across contexts by default, so do not place secrets there and assume the content script cannot read them.

Messages cross execution boundaries and should look like a small internal protocol. Storage crosses lifetime boundaries and should hold state needed after a popup closes or worker restarts. Neither mechanism turns one context into another.

Draw Page Notes with context lifetimes and arrows for popup → storage, popup → content script, command → service worker, and worker → content script. Annotate every arrow with its payload and mark which data originated on an untrusted page.

Check your understanding: extension foundations

Check Manifest V3, extension contexts, unpacked loading, permissions, actions, and the basic Page Notes architecture.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Popup and storage

Build the popup, identify the active page, and persist and render notes safely.

Build the popup interface

Create a small accessible form whose markup and scripts are packaged with the extension instead of loaded remotely.

The popup is an extension page. It can use HTML, CSS, modules, and extension APIs, but it disappears when focus moves away.

Add a label, textarea, save button, status region, and a separate module script. Manifest V3 packages executable code locally; do not load scripts from a CDN or use inline JavaScript.

Treat closing as normal, not exceptional. The popup can vanish while an asynchronous storage operation is running, so save immediately from the submit event and make the write itself authoritative. Do not keep the only copy of a draft or success state in popup variables.

```
<form id="note-form">
  <label for="note">Note for this page</label>
  <textarea id="note" name="note"></textarea>
  <button>Save note</button>
  <p id="status" role="status"></p>
</form>
<script type="module" src="popup.js"></script>
```

Extension pages have a restrictive content security policy. Keeping scripts in packaged files also makes the store-reviewed code match what users execute. Avoid inline event attributes such as

`onclick`; attach listeners from the module and render dynamic text through DOM properties.

Style a readable 320-pixel popup and test it entirely by keyboard. Open it, type without saving, click outside, and reopen it. Decide deliberately whether the draft is discarded, persisted, or restored, then make the interface communicate that behavior.

Identify the active page

Read the current tab URL after a user gesture and turn it into a stable key for one page-specific note.

Page Notes needs the active page URL, not broad access to the user's entire history.

The `activeTab` permission grants temporary access to the active tab after the user invokes the extension. Query the active tab from the popup and reject browser-internal URLs where the extension cannot run.

The grant lasts while the user remains on the same origin in that tab and is revoked when the tab closes or navigates to another origin. Opening the extension action, choosing its context-menu item, or invoking a declared command are qualifying gestures. It is a temporary capability, not a replacement for checking every result.

```
const [tab] = await chrome.tabs.query({ active: true, currentWindow: true })

if (!tab.url?.startsWith('http')) {
  throw new Error('Page Notes works on web pages')
}

const key = `note:${new URL(tab.url).href}`
```

Decide how URL identity works before storing data. Fragments often represent position rather than a separate document, tracking parameters may be noise, and query parameters may contain private information. A normalization function should be pure and documented so saving and loading cannot disagree.

Tabs can close or navigate between the query and the next API call. Verify `tab.id` and URL, handle rejected promises, and show an unavailable state instead of leaving a dead form. Test a normal HTTPS page, a different query string, a fragment change, `chrome://extensions`, and a tab closed while the popup is open.

Save with extension storage

Persist page notes with chrome.storage.local instead of popup memory or page-owned localStorage.

Popup variables disappear when the popup closes. Page localStorage belongs to the website origin and is the wrong owner for extension data.

`chrome.storage.local` stores extension-owned structured data and is available across extension contexts. Use an object whose property name is the page key, then await the write before showing success.

`storage.local` persists until the extension is removed and currently has a 10 MB quota unless broader storage authority is requested. That is ample for short notes, not permission to accumulate unlimited page data. Storage values are JSON-serializable, operations are asynchronous, and quota failures reject the promise.

```
form.addEventListener('submit', async event => {
  event.preventDefault()
  const value = new FormData(form).get('note').trim()
  await chrome.storage.local.set({ [key]: value })
  status.textContent = 'Saved'
})
```

By default, local storage is visible to content scripts too. Page Notes can keep note access in trusted extension contexts and send only the selected note to the active page; `setAccessLevel()` can restrict an area to trusted contexts when the design requires it. Storage is not encrypted secret storage.

Avoid a read-modify-write race over one giant notes object. Independent page keys let unrelated writes proceed without overwriting each other. Remove the key for an empty note instead of retaining meaningless records, and use `storage.onChanged` if multiple open extension pages must react.

Save two pages concurrently, close and reopen the popup, delete one note, and simulate a rejected write. Success must appear only after persistence completes, and the other page's value must remain intact.

Load and render safely

Initialize asynchronous popup state without racing user input and render stored text without interpreting it as HTML.

Storage reads are asynchronous. The user can interact before an unfinished initialization unless the interface has an explicit loading state.

Disable the form while resolving the tab and stored note, then enable it. Assign note text through `.value` or `.textContent`, never `innerHTML`, because stored content is still untrusted data.

```
form.hidden = true
const result = await chrome.storage.local.get(key)
note.value = result[key] ?? ''
form.hidden = false
note.focus()
```

Hiding the entire form removes context and can cause layout movement. A visible disabled form or stable skeleton can communicate what is loading while preserving its eventual space. On failure, keep a retry or close path; do not enable save when the page key is unknown.

Initialization also has a race with typing. If the form becomes editable before the read finishes, a late storage result can overwrite the user's input. Use one explicit state transition from loading to ready, or ignore a late result after the user has edited. Keep status text separate from the note value.

Add loading, ready, saved, unavailable, and error states. Slow the storage read, try to type during it, and test a stored note containing `<img onerror=alert(1)>`. The string must appear literally, with no network request or DOM element created from it.

Check your understanding: popup and storage

Check popup lifetime, active-tab URLs, storage.local, asynchronous initialization, safe DOM rendering, keys, and saved state.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Content scripts

Inject on demand, communicate across contexts, and make reversible page changes.

Run code in the page

Use a content script to access the current page DOM while keeping extension JavaScript isolated from page variables.

A content script runs alongside a web page and can read and change its DOM. Its JavaScript world is isolated from scripts belonging to the page.

Isolation prevents ordinary variable collisions, but both sides can affect the same DOM. Treat page content as untrusted and avoid exposing extension secrets or privileged APIs through inserted elements.

The script has only a subset of extension APIs. When it needs privileged work, it should send a narrow request to the service worker or extension page rather than injecting credentials or a broad bridge into the page. Likewise, code executed in the page's main world does not gain extension APIs.

The shared DOM is an attack and compatibility surface. The page can remove injected nodes, imitate your CSS classes, observe text you insert, and trigger ordinary DOM events. Use namespaced attributes and avoid placing private extension state in attributes or hidden elements.

Frames matter too. A script running in the top frame does not automatically describe every iframe, and broad frame injection increases authority. Page Notes only needs the top document, so keep that scope explicit.

Create `content.js` that marks the document with a namespaced data attribute without overwriting an existing value. Inspect the page console and Sources panel, then let page code remove the attribute and confirm the extension fails safely rather than treating DOM presence as trusted state.

Inject on user action

Use chrome.scripting and activeTab to run the content script only on the page where the user requests highlighting.

Page Notes does not need to run on every website automatically. The user can ask it to highlight a saved note on the active page.

Programmatic injection combines the `scripting` permission with temporary host access from `activeTab`. It avoids broad install-time host permissions and makes the action visible and intentional.

The permission pair has separate jobs: `scripting` allows programmatic injection, while `activeTab`

temporarily grants access to the invoked tab's origin. The grant ends when the tab closes or navigates to another origin. Keep the injection inside the user-initiated flow rather than storing a tab ID and assuming access remains later.

```
await chrome.scripting.executeScript({
  target: { tabId: tab.id },
  files: ['content.js']
})
```

Packaged files are easier to review and can contain reusable code. Passing `func` executes a serialized function and does not preserve its surrounding closure, so every dependency must be included explicitly. Choose one method and handle restricted pages, missing tab IDs, navigation races, and injection rejection.

Injection can run more than once. Make initialization idempotent by detecting the extension's existing root or by having one installed listener handle repeated commands. Add a Highlight button, click it repeatedly, navigate during the action, and verify there is never more than one panel or an unhandled promise rejection.

Send a message to the content script

Pass a small structured command from the popup to the page context and validate it before changing the DOM.

The popup and content script cannot call each other's functions. Messaging crosses that context boundary.

Send a message to the active tab after ensuring the content script exists. The receiver should accept only known message types and validate every field, because messages are another input boundary.

One-time messages are JSON-serialized in Chrome. Send plain data, not functions, DOM nodes, class instances, or objects whose meaning depends on prototypes. Keep payloads small and return an explicit result so the sender can distinguish "shown" from "no listener," "invalid note," or "restricted page."

```
await chrome.tabs.sendMessage(tab.id, {
  type: 'SHOW_PAGE_NOTE',
  note
})
```

`tabs.sendMessage` can reject when no matching content script exists—for example after the extension was reloaded but the tab was not. Page Notes can inject first and then send, or treat a missing receiver as an initialization signal, but it must not blindly inject after every error.

Handle `SHOW_PAGE_NOTE` with a type guard and return `{ ok: true }`. Send an unknown type, a non-string note, and a message immediately after an extension reload. The page must remain unchanged on invalid input and the popup must show a useful recovery path.

Make page changes reversible

Insert one accessible note panel without corrupting page markup, duplicating UI, or leaving changes the user cannot undo.

A content script is a guest in someone else's document. Its DOM changes should be minimal, namespaced, and reversible.

Create elements with DOM APIs, render note text with `textContent`, use unique prefixed IDs or a shadow root, and remove an existing panel before inserting another. Include a close button and restore any page state you change.

Prefer adding one extension-owned root over rewriting page HTML. A shadow root can reduce style collisions, but it does not hide content from the page and it still needs accessible names, focus styles, and sensible colors. Namespaced light-DOM CSS is simpler when isolation needs are small.

Track every side effect: inserted nodes, event listeners, classes, inline styles, scroll locking, and focus changes. Reversal means removing or restoring each one. Store original values before mutation and do not assume the page kept them unchanged while your panel was open.

Repeated show messages should update the existing panel rather than create duplicates. Closing should return focus to a meaningful page or extension control when possible, and removal by the host page should not crash later message handling.

Build a fixed note panel with a labelled close button. Show it repeatedly with changing text, close it, let page code remove it, and show it again. Compare the DOM before and after: no duplicate IDs, orphaned listeners, global style changes, or interpreted note markup should remain.

Check your understanding: content scripts

Check content-script isolation, DOM access, programmatic injection, activeTab, message validation, and reversible page changes.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Service worker and messaging

Handle browser events, commands, worker lifecycle, contracts, and context debugging.

Add the extension service worker

Register a background service worker for browser events without assuming it remains alive or keeps global state.

Manifest V3 uses an extension service worker for event-driven background behavior. The browser starts it when needed and can stop it when idle.

Declare the file under `background.service_worker`. Register event listeners synchronously at the top level so they exist when the worker starts. Persist important state in storage rather than global variables.

Think in independent events, not in one long-running process. Chrome normally terminates an idle extension worker after about 30 seconds, and it can end unexpectedly. The next command or message starts a fresh JavaScript instance where module globals have returned to their initial values. Do not add keep-alive work to fight that lifecycle.

```
{
  "background": {
    "service_worker": "service-worker.js",
    "type": "module"
```

```
  }
}
```

`runtime.onInstalled` is an extension lifecycle event used for installation and update initialization; it is not "the worker started." `runtime.onStartup` refers to the browser profile starting and also does not fire for every worker wake. Put ordinary initialization in an idempotent function called by the event that needs it.

Listeners must be registered during initial module evaluation. Registering `commands.onCommand` only after awaiting a storage read creates a window where Chrome wakes the worker for a command but no listener is ready. Read required state inside the listener instead.

Store one counter only in a global and another in `storage.session` or `storage.local`. Invoke events, wait for the worker to stop, and invoke them again. Inspect the worker from `chrome://extensions` and explain the different results without relying on a startup log.

Add a keyboard command

Declare a browser command and handle it in the service worker to open or toggle the current page note.

Keyboard commands are browser events, so they belong in the service worker rather than the short-lived popup.

Declare a command and suggested key in the manifest, then listen with `chrome.commands.onCommand`. Keyboard shortcuts can conflict, so provide a discoverable action and handle a missing assignment gracefully.

Suggested keys are requests, not guarantees. The browser or operating system may reserve the combination, and users can change or remove it from the extension shortcut settings. The command name in the event is the stable contract; the physical key is configuration.

A command invocation is a qualifying user gesture for `activeTab`, so the worker can query the active tab and inject or message it within that flow. Still reject missing IDs and restricted URLs. If no content script is present after an extension reload, initialize it deliberately before retrying the toggle once.

The worker may start only to handle this event. Read the saved note and any durable toggle state inside the listener, and make the operation idempotent. Do not assume a previous popup left the right tab or note in a global variable.

Add `toggle-page-note`, list its effective shortcut with `chrome.commands.getAll()`, and include a link or instruction for managing shortcuts. Test an assigned key, an unassigned command, a restricted page, a worker restart, and a tab that still holds an older content script.

Design message contracts

Use named message types, narrow payloads, clear responses, and error handling across popup, worker, and content script.

As an extension grows, ad hoc messages become an undocumented internal API.

Define the allowed message shapes in one module or type declaration. Include only serializable data, validate senders where trust differs, return explicit results, and handle `runtime.lastError` or rejected promises.

```
type PageNoteMessage =
  | { type: 'SHOW_PAGE_NOTE'; note: string }
  | { type: 'HIDE_PAGE_NOTE' }
  | { type: 'GET_ACTIVE_NOTE'; url: string }

type MessageResult =
  | { ok: true }
  | { ok: false; code: 'INVALID_MESSAGE' | 'UNAVAILABLE' }
```

Validate at runtime as well as with TypeScript because a sender can be old, compromised, or plain JavaScript. Check `sender.id` for external or cross-extension surfaces and `sender.tab` or URL when the receiver's authority depends on the source. Do not echo exception stacks or storage contents in errors.

Chrome messaging uses JSON serialization. For broadly compatible asynchronous listeners, call `sendResponse` and return the literal `true` so the channel stays open. Current Chrome is rolling out returned-promise support, but choosing it without a minimum-version decision can produce inconsistent behavior across installed clients.

Document the three message types with sender, receiver, payload, response, authorization rule, and failure codes. Exercise an unknown type, malformed data, no receiver, two listeners attempting to respond, and a response that cannot be serialized.

Debug each context

Open the correct DevTools target for popup, service worker, content script, and page instead of looking for every log in one console.

Each extension context has its own console, lifecycle, and source environment.

Inspect the popup by right-clicking it, the service worker from `chrome://extensions`, and content scripts from the target page's DevTools. Use the extension errors panel for manifest and runtime failures. Reload the extension after background or manifest changes.

Choose the DevTools execution context before evaluating code. The page console and isolated content-script console can both appear in one tab, while only one has extension globals. Popup DevTools keeps that popup open for inspection, which changes its usual close-on-blur lifecycle, so repeat important checks without DevTools too.

The worker inspector can keep or wake the worker and therefore hide lifecycle bugs. Record evidence, close the inspector, wait for termination, and reproduce the event from a cold worker. Use `chrome.storage` inspection and the extension Errors view to distinguish durable state from logs belonging to an old instance.

Add a small correlation ID to one user action and include it in popup, worker, and content-script logs. This turns several isolated consoles into one trace without logging the note or complete sensitive URL. Clear the logs and reproduce from a known extension version before drawing conclusions.

Trigger one deliberate error in each context plus a missing receiver after extension reload. Record the target, extension version, action ID, visible symptom, console or Errors-panel evidence, and exact reload sequence required to fix it.

Check your understanding: service worker and messaging

Check extension service-worker lifecycle, event registration, commands, messaging, storage-backed state, and context-specific debugging.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Secure, test, and publish

Review permissions and data, protect extension pages, test behavior, and package a release.

Review permissions and data

Audit every requested capability, stored value, transmitted field, and website interaction before asking users to trust the extension.

Extensions can see sensitive browsing context. A permission that is easy to add may be difficult for a user to justify.

Remove unused permissions, prefer activeTab to broad host patterns, collect no data the feature does not need, and keep page notes local unless sync is an explicit feature. Document retention and deletion behavior plainly.

Separate API permissions from host access in the audit. `storage` enables an API; `activeTab` grants temporary access after a gesture; `host_permissions` grants ongoing access to matching origins. Optional permissions can delay broader authority until a user requests the feature, but the request must happen from a user gesture and refusal must leave a usable product.

Audit flows, not only declarations. The active URL becomes a storage key, note text moves from a trusted popup into an untrusted page DOM, and extension updates can change future behavior. For each transition, identify who can read it, how long it remains, and what happens when the user deletes the note or uninstalls the extension.

Local-only is a meaningful boundary only if the code has no analytics, crash reports, remote API, or embedded third-party resource carrying that data. Verify with the Network panel and source review. If data practices change later, store disclosures and consent must change before the release.

Create a permission-and-data table covering feature, gesture, context, fields, destination, retention, deletion, warning, and denial behavior. Remove one permission and prove the unrelated journeys still work. Capture Network evidence that Page Notes transmits no note or page URL.

Protect extension pages

Keep executable code packaged locally, avoid dangerous HTML sinks, and use a restrictive content security policy.

Manifest V3 disallows remotely hosted executable code. This keeps reviewed package behavior from changing after publication.

Bundle dependencies, avoid `eval` and string-to-code APIs, keep inline scripts out, and render untrusted values as text. Do not weaken the extension page CSP to make an unsafe library work; choose code compatible with the security model.

Remote code includes more than a CDN `<script>`. Downloaded JavaScript, WebAssembly, strings evaluated as code, and libraries that fetch executable logic after review all break the packaged-code trust model. Remote JSON and APIs can provide data, but the package must decide how that data is interpreted.

Treat extension pages as privileged origins. A popup can call extension APIs, so an HTML injection there can become more serious than the same bug on an ordinary page. Prefer `.textContent`, `.value`, and explicit DOM construction. If sanitized rich content is truly required, use a reviewed packaged sanitizer and test its configuration with hostile input.

Keep web-accessible resources minimal. Declaring a file web-accessible lets matching pages request it; it does not make arbitrary extension APIs available, but it exposes assets and can reveal extension presence. Page Notes does not need to expose its popup or scripts to websites.

Search for remote executable URLs, inline handlers, `innerHTML`, `insertAdjacentHTML`, `eval`, `new Function`, dynamic script creation, and unnecessary web-accessible resources. Test hostile stored text in both popup and injected panel and inspect Network for unexpected code loads.

Test the extension

Combine pure module tests, a repeatable manual matrix, and browser automation for the critical save-and-show journey.

Not every extension API is easy to unit-test, but core logic and complete browser behavior can still be covered deliberately.

Extract URL keys and validation into pure modules. Test them with Node. Maintain a manual matrix for restricted pages, restarts, permissions, keyboard use, and upgrades. Use a browser automation profile that loads the unpacked extension for the primary journey.

Arrange tests by boundary. Pure tests cover normalization, message parsing, and state transitions. Integration tests use small fake API adapters to verify storage rejection and missing-tab behavior. Browser tests prove the packaged manifest, permissions, contexts, DOM injection, and actual extension lifecycle work together.

Lifecycle tests are essential in Manifest V3. Let the worker become idle, close and reopen the popup, reload the extension while a page remains open, navigate across origins, and upgrade from a package containing existing data. Keeping DevTools open can change worker and popup behavior, so rerun the critical path without inspectors attached.

Assert evidence rather than absence of obvious failure: one stored record, one injected root, a defined response, no unexpected network request, and the correct accessible name and focus target. Give test profiles deterministic pages and clear stored data between independent cases.

Test key normalization, hostile text rendering, concurrent saves, close and reopen, duplicate injection, toggle after worker restart, reload with a stale content script, denied or expired access, restricted pages, keyboard-only use, and an upgrade preserving existing notes. Record which layer catches each regression.

Package and publish

Prepare icons, store copy, screenshots, privacy disclosures, version updates, and a clean ZIP without development files or secrets.

Publishing is a product and trust review, not only a ZIP upload.

Use a valid increasing version, polished icons, accurate feature copy, and screenshots of real behavior. Explain permissions and data use. Build from a clean checkout, exclude tests and secrets, submit through the store, and test the installed release before promoting it.

The ZIP root must contain `manifest.json`, not another project directory. Build a deterministic allowlist of runtime files rather than zipping the repository and trying to exclude mistakes. Inspect source maps, fixtures, environment files, private keys, unused permissions, remote-code patterns, and generated dependencies before upload.

The store listing and privacy declarations are part of the release. Describe the extension's single purpose and actual behavior; screenshots must not promise features the package lacks. If code, assets, manifest, or data practices change, update the relevant package and dashboard metadata and submit the new version for review.

An accepted update is not maintenance completion. Existing users keep the previous release until the update publishes and installs, and added permissions can require user acceptance. Test the store-installed artifact in a clean profile, preserve a known-good package and release notes, monitor support and policy notices, and prepare a rollback path. Larger publishers may use staged rollout, but it does not replace pre-release testing.

Create a release checklist and reproducible ZIP whose root contains `manifest.json`. Compare its file list and hashes with the intended build output, install that exact package in a clean profile, upgrade from the prior version, and verify permissions, saved notes, store copy, privacy declarations, and rollback materials.

Check your understanding: secure, test, and publish

Check least privilege, remote code restrictions, CSP, safe data handling, extension testing, packaging, privacy disclosure, and release updates.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/browser-extensions/.