Build a Production-Ready Chrome Extension Productivity Suite with Manifest V3

Back

Mandatory Task

Domain

Skills

Report Issue

Build a multi-feature Chrome extension using Manifest V3 to enhance user productivity. You will learn to work with core Chrome APIs

in the software industry.

Description

Background

Chrome extensions are small software programs that customize the browsing experience. They enable users to tailor Chrome functionality and behavior to individual needs or preferences. With the introduction of Manifest V3, the extension platform has undergone significant changes, emphasizing security, performance, and privacy. Key concepts include a service worker replacing background pages, a new `scripting` API for executing code in tabs, and stricter permission models.

A typical production-ready extension consists of several components: a manifest file (`manifest.json`) that declares metadata and permissions, a popup UI for user interaction, an options page for configuration, content scripts to interact with web pages, and a service worker for background logic and event handling. Communication between these components is managed through a message passing system. State is often managed using the `chrome.storage` API, which offers both local and synced storage options.

Implementation Details

Step 1: Project Setup & Manifest V3 Configuration

Initialize your project with a modern frontend framework like React or Vue for the UI components. Create the `manifest.json` file at the root. This is the heart of your extension. It must be configured for Manifest V3 and declare all necessary permissions and components.

```
{
  "manifest_version": 3,
  "name": "Productivity Suite",
  "version": "1.0",
  "description": "A multi-feature productivity extension.",
  "permissions": [
    "storage",
    "tabs",
```

```
    "scripting",
```



```
    "<all_urls>"
  ],
  "background": {
    "service_worker": "background.js"
  },
  "action": {
    "default_popup": "popup.html"
  },
  "options_page": "options.html",
  "chrome_url_overrides": {
    "newtab": "newtab.html"
  },
  "commands": {
    "_execute_action": {
      "suggested_key": {
        "default": "Ctrl+Shift+P",
        "mac": "Command+Shift+P"
      },
      "description": "Open Popup"
    }
  }
}
```

Step 2: Build UI Components

Develop the user interfaces for the popup, options page, and the new tab override page. The popup (`popup.html`) will be the main interaction hub, while the options page (`options.html`) will handle settings like the website blocklist. The new tab page (`newtab.html`) will feature custom widgets.

Step 3: Implement Core Features

Implement the logic for each feature. This involves writing JavaScript for the service worker (`background.js`), content scripts, and your UI components. Use the `chrome.tabs` , `chrome.storage` , and `chrome.scripting` APIs extensively.

- **Tab Management:** Logic to query current tabs, group them, and save/restore sessions to `chrome.storage.local`.

- **Note-Taking:** Simple markdown-enabled notes stored in `chrome.storage.local`.

Step 4: Add Advanced Integrations

- **Context Menu:** In your service worker, use `chrome.contextMenus.create` to add a right-click menu item. This should trigger an event that your service worker listens for.
- **Keyboard Shortcuts:** Define commands in your `manifest.json`. Listen for these commands using `chrome.commands.onCommand.addListener` in your service worker.
- **Data Export:** Implement a function to retrieve all data from `chrome.storage` and download it as a JSON file.

Submission Artifacts Checklist

Your final submission must be a `.zip` archive containing the following:

- / (root)
  - manifest.json # The extension manifest file (req-manifest-v3-setup)
  - popup.html # The popup UI file
  - options.html # The options page UI file
  - newtab.html # The custom new tab page UI file
  - background.js # The service worker script
  - /src/ # Source code for your UI (React, Vue, etc.)
  - /dist/ # Bundled/compiled code ready to be loaded by Chrome
  - package.json # Project dependencies and scripts
  - webpack.config.js # Or other bundler configuration
  - README.md # Detailed setup and usage instructions
  - .env.example # If any environment variables are used

Implementation Guidelines

- **State Management:** For a complex extension like this, consider a state management library (like Redux, Zustand, or Pinia) for your UI

popup, options page, content scripts, and service worker. Avoid direct DOM manipulation of other components' pages.

- **Performance:** Be mindful of the performance impact of your extension. Use `chrome.storage.sync` only for small configuration data, and `chrome.storage.local` for larger data like notes or tab sessions. Debounce events where necessary, especially those tied to user input.
- **Security:** Be cautious with permissions. Only request the permissions you absolutely need. When injecting scripts or styles, use the `chrome.scripting` API instead of older, less secure methods. Sanitize any user input that is displayed in your UI to prevent XSS attacks.
- **Error Handling:** Gracefully handle errors from Chrome APIs. Many of these APIs are asynchronous and return Promises or use callbacks. Wrap API calls in `try...catch` blocks or check `chrome.runtime.lastError` in callbacks.

FAQ

- **Q: Can I use a different frontend framework than React?**
  - A: Yes, you are free to use any modern JavaScript framework like Vue, Svelte, or even vanilla JavaScript, as long as you meet all the functional requirements, including those testable via `data-testid` attributes (req-popup-ui, req-options-page-ui).
- **Q: How do I test the extension during development?**
  - A: You can load your extension in an unpacked format. Go to `chrome://extensions`, enable "Developer mode", and click "Load unpacked". Select your project's build/distribution directory.
- **Q: What's the difference between `storage.sync` and `storage.local`?**
  - A: `storage.sync` automatically syncs data across all browsers a user is logged into, but has stricter storage limits. It's ideal for user settings (req-settings-sync). `storage.local` is for larger data specific to a single machine and has higher limits.
- **Q: My service worker seems to stop running. Is this normal?**
  - A: Yes, in Manifest V3, service workers are event-based and will be terminated after a period of inactivity to save resources. You must design your logic around event listeners (`chrome.tabs.onUpdated`, `chrome.commands.onCommand`, etc.) rather than relying on a persistent global state.
- **Q: How do I debug the popup or service worker?**
  - A: To debug the popup, right-click on the extension icon and select "Inspect popup". To debug the service worker, go to `chrome://extensions`, find your extension, and click the "service worker" link to open its dedicated DevTools console.
- **Q: How should the website blocker work?**
  - A: The service worker should listen for tab updates using `chrome.tabs.onUpdated`. If the URL matches a blocked site, it should programmatically inject a content script using `chrome.scripting.executeScript` that redirects the page or replaces its content

(req-website-blocker-logic).

Core Requirements

1. The extension must be configured using Manifest V3 and declare all necessary permissions in `manifest.json`.

   File: `manifest.json`

   Required Properties:

   - `manifest_version`: Must be `3`.
   - `permissions`: Must include `"storage"`, `"tabs"`, `"scripting"`, and `"contextMenus"`.
   - `host_permissions`: Must include `"<all_urls>"` to allow content scripts on any page.
   - `background.service_worker`: Must be defined.
   - `action.default_popup`: Must be defined.
   - `options_page`: Must be defined.

   Verification:

   1. Parse the `manifest.json` file.
   2. Verify `manifest_version` is exactly `3`.
   3. Verify the `permissions` array contains all four required strings.
   4. Verify `host_permissions` is correctly configured.
   5. Verify paths are defined for the service worker, popup, and options page.

---

2. The extension popup must provide the main user interface with clear navigation between features.

   File: `popup.html` (and associated scripts)

Required Elements with `data-testid` attributes:

- A container listing saved sessions: `data-testid="sessions-list"`
- A textarea for taking notes: `data-testid="notes-textarea"`
- A button to save the notes: `data-testid="save-notes-btn"`
- A link/button to open the options page: `data-testid="open-options-btn"`

Verification:

1. Use a browser automation tool to open the extension popup.
2. Verify that all elements with the specified `data-testid` attributes exist in the popup's DOM.

---

3. A dedicated options page must allow users to configure the website blocker.

File: `options.html` (and associated scripts)

Required Elements with `data-testid` attributes:

- An input field for adding a new hostname to block: `data-testid="block-hostname-input"`
- A button to submit the new hostname: `data-testid="add-block-btn"`
- A container that lists all currently blocked hostnames: `data-testid="blocked-sites-list"`
- A button to export all user data: `data-testid="export-data-btn"`

Verification:

1. Use a browser automation tool to navigate to the extension's options page.
2. Verify that all elements with the specified `data-testid` attributes exist in the DOM.

---

4. Users must be able to save all tabs in the current window as a named session.

1. User provides a name for the session (e.g., via a `prompt` or an input field).
2. User clicks the element with `data-testid="save-session-btn"`.
3. The extension queries for all tabs in the current window.
4. The session, including the name and an array of tab URLs, is saved to `chrome.storage.local`.

Verification:

1. Open a browser window with at least 3 distinct tabs (e.g., google.com, github.com, example.com).
2. Programmatically open the popup and trigger a click on `data-testid="save-session-btn"` (providing a name like 'test-session').
3. Execute a script in the extension's context to call `chrome.storage.local.get('sessions')`.
4. Verify the returned object contains a 'test-session' key with an array of the correct tab URLs.

---

5. Users must be able to restore a saved tab session, opening all its tabs in a new window.

Feature: Restore Tab Session

Behavior:

1. The popup UI lists saved sessions. Each session should have a restore button, e.g., `data-testid="restore-session-test-session"`.
2. When the user clicks the restore button, the extension creates a new browser window.
3. All URLs from the saved session are opened as new tabs in that new window.

Verification:

1. First, save a session named 'test-session' with known URLs (see req-tab-session-save).
2. Close the original tabs.
3. Open the popup, click the element `data-testid="restore-session-test-session"`.

4. Verify that a new window is created and that `chrome.tabs.query` confirms that all the saved URLs are now open.

6. Users must be able to add hostnames to a blocklist via the options page.

> **Feature:** Add Hostname to Blocklist
>
> **Behavior:**
>
> 1. On the options page, the user types a hostname (e.g., `"facebook.com"`) into the input with `data-testid="block-hostname-input"`.
> 2. The user clicks the button with `data-testid="add-block-btn"`.
> 3. The hostname is added to an array of blocked sites stored in `chrome.storage.sync`.
>
> **Verification:**
>
> 1. Navigate to the options page.
> 2. Programmatically enter `"evil-corp.com"` into the input and click the add button.
> 3. Execute `chrome.storage.sync.get('blockedSites')`.
> 4. Verify the returned `blockedSites` array now includes `"evil-corp.com"`.

---

7. The extension must block navigation to sites on the blocklist.

> **Feature:** Active Website Blocking
>
> **Behavior:**
>
> 1. A site (e.g., `"evil-corp.com"`) is on the blocklist.
> 2. When the user attempts to navigate to any page on that host (e.g., `https://www.evil-corp.com/login`),
> 3. The extension prevents the page from loading and either redirects to a local `blocked.html` page within the extension or injects a content script to replace the page body with a "blocked" message.

Verification:

2. Use a browser automation tool to navigate to `http://example.com`.
3. Verify that the final page URL is NOT `http://example.com`. It should be an internal extension page (`chrome-extension://...`) OR the page content must contain a specific element indicating it's blocked, e.g., `<h1 data-testid="blocked-message">Page Blocked</h1>`.

---

8. The popup must include a simple note-taking feature that persists data.

Feature: Persistent Notes

Behavior:

1. User types text into the textarea with `data-testid="notes-textarea"`.
2. User clicks the save button with `data-testid="save-notes-btn"`.
3. The text content is saved to `chrome.storage.local`.
4. When the popup is closed and reopened, the saved text is loaded back into the textarea.

Verification:

1. Open the popup, enter "Test note content" into the textarea, and click the save button.
2. Close and reopen the popup.
3. Verify the value of the textarea `data-testid="notes-textarea"` is still "Test note content".
4. Execute `chrome.storage.local.get('notes')` and verify the stored value matches.

---

9. The extension must override the default Chrome new tab page with a custom page.

Feature: New Tab Override

Behavior:

2. When the user opens a new tab, this local HTML file is displayed instead of Chrome's default page.

Verification:

1. Check for the `chrome_url_overrides` key in `manifest.json` as specified.
2. Use a browser automation tool to open a new tab.
3. Verify the URL of the new tab starts with `chrome-extension://` followed by the extension's ID and the path to the new tab HTML file.

---

10. The custom new tab page must display at least two widgets with data from storage.

File: `newtab.html` (and associated scripts)

Behavior:

1. The new tab page must display the user's notes.
2. The new tab page must display the list of saved tab sessions.

Required Elements with `data-testid` attributes:

- A container for notes: `data-testid="widget-notes"`
- A container for saved sessions: `data-testid="widget-sessions"`

Verification:

1. First, save some notes and at least one tab session.
2. Open a new tab.
3. Verify the element `data-testid="widget-notes"` exists and contains the saved note text.
4. Verify the element `data-testid="widget-sessions"` exists and contains the name of the saved session.

API Usage: `chrome.storage.sync`

Behavior:

- The list of blocked websites, as defined in `req-website-blocker-add`, must be stored using `chrome.storage.sync` to ensure settings persist across a user's logged-in browsers.

Verification:

- This is verified as part of the test for `req-website-blocker-add`. The verification script will specifically check the `sync` storage area for the `blockedSites` array.

12. The user must be able to export all their data as a single JSON file from the options page.

Feature: Export Data

Behavior:

1. User clicks the button `data-testid="export-data-btn"` on the options page.
2. A file download is initiated for a file named `productivity_suite_export.json`.
3. The JSON file contains all data from `chrome.storage`, including notes, tab sessions, and the blocklist.

JSON Schema:

```
{
  "sessions": { "type": "object" },
  "notes": { "type": "string" },
  "blockedSites": { "type": "array", "items": { "type": "string" } }
}
```

Verification:

2. On the options page, click `data-testid="export-data-btn"`.
3. Intercept the downloaded file.
4. Verify the file is named `productivity_suite_export.json`.
5. Parse the JSON content and validate its structure against the specified schema, ensuring the keys `sessions`, `notes`, and `blockedSites` are present.

---

13. The extension must register at least two keyboard shortcuts for quick actions.

File: `manifest.json`

Behavior:

- The manifest must define at least two commands under the `"commands"` key. One must be the default `_execute_action` (for opening the popup), and another must be a custom command (e.g., `"save-session"`).

Required `manifest.json` Structure:

```
{
  "commands": {
    "_execute_action": { ... },
    "custom-command-name": {
      "suggested_key": { "default": "Ctrl+Shift+S" },
      "description": "A custom action"
    }
  }
}
```

partnr