

Les plugins

Du manifeste au paquet signé

Référence technique du bac à sable, du registre des capacités, de la chaîne de signature et des quatre voies d'exécution, suivie d'un guide de rédaction pas à pas.

Sommaire

COMPRENDRE

1.	Modèle mental	3
2.	Le modèle d'exécution	4
3.	Anatomie d'un plugin	5
4.	Les quatre voies d'exécution	6
5.	Le protocole du pont	7
6.	Contrat et implémentation	9

ÉTENDRE

7.	Les contributions	11
8.	Où s'ouvre un panneau	12
9.	Thème, langue et instantané	13
10.	Stockage et ressources embarquées	14
11.	Le moteur R	16

GOUVERNER

12.	Capacités et consentement	18
13.	Signature, identité d'auteur et provenance	19
14.	Journal des accès	21
15.	Conflits, dépendances et compatibilité	22
16.	Le gestionnaire et les surfaces d'accès	23

RÉFÉRENCE

17.	Catalogue des méthodes hôtes	26
18.	Le manifeste, champ par champ	30
19.	Publier un plugin	31
20.	L'atelier de développement	32
21.	Quatre plugins types	33
22.	Dépannage	36
23.	Bornes et persistance	38

ÉCRIRE

24.	De zéro à un plugin qui tourne	42
25.	Choisir ses capacités	43
26.	Écrire l'interface d'un panneau	44
27.	Un plugin complet	46
28.	Liste de contrôle avant publication	49

Comprendre

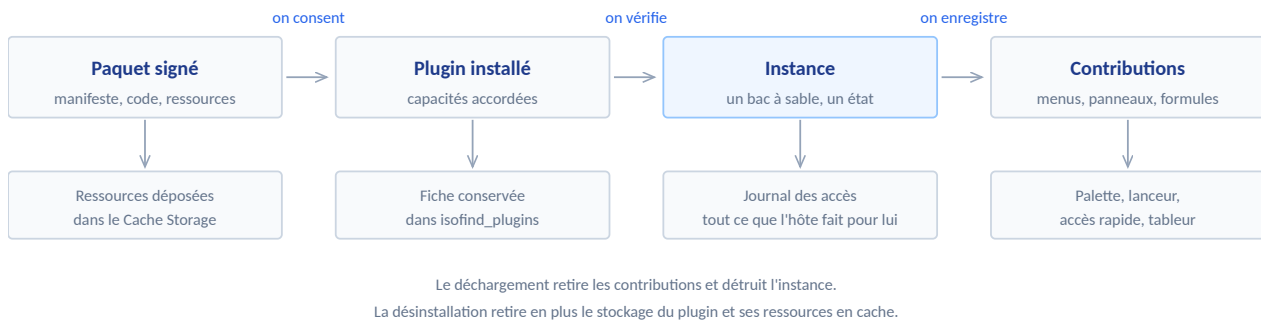
Ce qu'est un plugin, ce qu'il a le droit de toucher, et sur quoi il est permis de s'appuyer quand on en écrit un.

1. Modèle mental

Six mots suffisent à décrire le système, à condition de ne pas confondre les deux premiers. Le manifeste est un texte que l'auteur écrit ; la capacité accordée est une décision que l'utilisateur prend. Le premier demande, le second dispose, et aucune ligne de code du plugin ne franchit cet écart.

Mot	Ce que c'est	Où il vit
Paquet	Une archive <code>.isplugin</code> , qui est un zip : un manifeste à la racine, un point d'entrée, éventuellement une icône et un dossier de ressources. C'est la seule forme sous laquelle un plugin circule.	Sur le disque, ou au catalogue.
Manifeste	Ce que le plugin déclare de lui-même : son identité, sa version, son point d'entrée, les capacités qu'il demande et les contributions qu'il apporte. Un texte, donc une demande, jamais une autorisation.	<code>manifest.json</code> , conservé tel quel à l'installation, champ <code>signature</code> compris.
Capacité	Un droit nommé, demandé par le manifeste et accordé case par case au moment de l'installation. Ce qui n'a pas été coché est refusé à l'exécution.	Sur la fiche du plugin installé, dans <code>isofind_plugins</code> .
Instance	Un plugin en cours d'exécution : son bac à sable, ses capacités accordées, son verdict de signature, ses contributions vivantes. Deux chargements du même plugin ne partagent rien.	En mémoire, une par plugin chargé.
Contribution	Ce que le plugin ajoute à l'application : une entrée de menu, un panneau, une formule de tableur. Elle est enregistrée par l'hôte, sous le nom du plugin, et retirée avec lui.	Dans les registres de l'hôte, jamais dans le bac à sable.
Bac à sable	L'iframe dans laquelle le code du plugin s'exécute, sans <code>allow-same-origin</code> , donc sur une origine opaque. Le plugin y dispose d'un document entier, et de rien d'autre.	Sous <code>document.body</code> , en permanence, révélée quand un panneau s'ouvre.

Le cycle de vie



Trois idées qui expliquent la plupart des choix

- 1. Le manifeste demande, l'hôte décide.** Rien de ce qu'un plugin écrit ne lui donne un droit. Un manifeste qui déclare une capacité inconnue la voit ignorée, un plugin qui appelle une méthode dont la capacité n'a pas été cochée reçoit un refus, et un panneau muni d'une zone de dock ne se dispense pas pour autant de `ui:panel`. Ce qui protège n'est jamais une consigne, mais une méthode qu'on n'a pas répondue.
- 2. Le plugin ne partage rien avec la page.** Il vit sur une origine opaque : pas de `document` de l'hôte, pas de `localStorage`, pas de cookies, et pas de requête vers le serveur local, que la politique de sécurité du contenu ferme indépendamment des capacités. Tout ce qu'il obtient passe par un protocole de messages dont l'hôte est l'unique arbitre.
- 3. Le confinement et la vérification répondent à deux questions différentes.** Le bac à sable dit ce qu'un plugin peut faire ; la signature dit d'où il vient et si son contenu est celui qui a été signé. L'un ne remplace pas l'autre. Un plugin peut être parfaitement confiné et malveillant dans son domaine, et un plugin signé par un auteur connu reste confiné exactement comme les autres.

2. Le modèle d'exécution

Un plugin ne s'exécute pas dans la page. C'est la décision dont tout le reste découle, et elle a un prix qu'il vaut mieux connaître avant d'écrire.

Le bac à sable est une iframe portant `sandbox="allow-scripts"`, sans `allow-same-origin`. Son document ne vient pas d'une URL mais d'un `srcdoc` construit par l'hôte, ce qui permet d'y injecter une politique de sécurité du contenu calculée pour cette instance, et les jetons de thème en vigueur au moment du chargement. L'origine étant opaque, l'hôte lui-même ne peut pas lire le `contentDocument` de l'iframe qu'il vient de créer : la seule voie de communication, dans les deux sens, est `postMessage`.

```
default-src 'none';
script-src 'unsafe-inline' 'unsafe-eval';
style-src 'unsafe-inline';
connect-src 'none' | https:      // selon la capacité network et le mode air-gap
```

`connect-src` est le seul terme variable, et il ne s'ouvre que sur `https:`, jamais sur l'origine locale. Un plugin ne peut donc pas appeler l'API d'IsoFind directement, même avec toutes les capacités du monde : ce qui lui est accordé passe par les méthodes de l'hôte, qui sont nommées et comptées. En mode air-gap, le terme

retombe à `'none'` quelle que soit la capacité demandée, et la case correspondante apparaît grisée à l'installation plutôt que simplement absente.

L'instantané des échantillons, et pourquoi il existe. Les plugins lisent les données de façon synchrone, au milieu d'un rendu. Rien de ce qui traverse une frontière d'isolation ne peut l'être. Plutôt que d'imposer à chaque auteur une réécriture complète en asynchrone, l'hôte livre une copie des échantillons dans le message d'initialisation, et le plugin la lit comme un tableau ordinaire.

Cette copie ne part par aucune requête : elle est recopiée en mémoire dans le bac à sable. Le journal du serveur ne la verrait donc pas, et laisserait croire que le plugin n'a rien lu alors qu'il détient toute la base. Elle est déclarée explicitement au journal des accès, et l'hôte attend le verdict avant de livrer. Une révocation posée après coup ne reprend pas ce qui est déjà parti : elle ne mord qu'au prochain chargement.

Ce qui manque volontairement

L'accès au DOM de l'hôte. Un plugin ne dessine que dans son propre document. Ce que l'application affiche autour, en-tête de modale, onglet de dock, bouton de fermeture, appartient à l'hôte. C'est ce qui permet de déplacer un panneau d'une surface à l'autre sans que le plugin ait à le savoir.

Le réseau vers le serveur local. Il a été essayé, et il ne marche pas : la politique de sécurité du contenu bloque la sortie depuis le bac à sable vers le serveur local, quelle que soit la capacité accordée. Un plugin qui a besoin de données passe par `samples:read` et l'instantané, ou il embarque ce dont il a besoin.

Deux panneaux du même plugin affichés ensemble. Un plugin n'a qu'un bac à sable, donc qu'une iframe, donc qu'un rectangle à l'écran. Ouvrir le second ferme le premier, explicitement, par la surface où il se trouvait.

L'exécution hôte sans le dire. La voie native existe, mais elle se déclare dans le manifeste et se lit sur la fiche du plugin. Un plugin ne peut pas y glisser en silence, et l'utilisateur qui accorde des capacités voit à quoi il consent.

3. Anatomie d'un plugin

Un plugin est une archive zip portant l'extension `.isplugin`. Elle contient un manifeste à la racine, et tout ce que ce manifeste désigne.

```
mon-plugin.isplugin
├─ manifest.json      // obligatoire, à la racine, nommé ainsi
├─ plugin.js          // le point d'entrée que le manifeste désigne
├─ icon.png           // facultatif
├─ assets/            // facultatif, extrait à l'installation
│   └─ moteur.wasm
│   └─ base.json
```

Le manifeste est le seul fichier dont l'emplacement et le nom sont imposés. Le point d'entrée, l'icône et le dossier de ressources sont désignés par des champs, et c'est le champ qui fait foi : le dépôt des ressources suit le préfixe déclaré et non un `assets/` codé en dur, parce que le signataire s'appuie lui aussi sur le manifeste.

Deux conventions différentes pour désigner les mêmes fichiers produiraient deux cartes d'empreintes différentes, donc un refus permanent et inexplicable.

```
{
  "id": "mon-plugin",           // stable, sert de clef partout
  "name": "My Plugin",         // en anglais : catalogue et consentement
  "version": "1.4.0",
  "author": "Nom de l'auteur",
  "description": "Short description",
  "category": "analysis",      // analysis | visualization | data | tools | science

  "entry": "plugin.js",        // l'extension décide du langage
  "capabilities": ["samples:read", "ui:menu", "ui:panel"],

  "contributes": {
    "menus": [ { "id": "ouvrir", "label": "Open the panel" } ],
    "panels": [ { "id": "principal", "label": "My Plugin", "dock": "right" } ]
  },

  "assets": "assets/",         // déposé dans le Cache Storage à l'installation
  "dependencies": ["autre-plugin"],
  "icon": "icon.png",

  "signature": { ... }         // posé par la signature, jamais écrit à la main
}
```

Le manifeste installé n'est pas reconstruit. À l'installation, certains champs sont recopiés à plat sur la fiche du plugin pour la liste et les badges, mais le manifeste d'origine est conservé entier à côté, champ `signature` compris. C'est lui, et lui seul, que le vérificateur recalcule. Une fiche reconstruite à partir des champs aplatis ne se vérifierait plus, et le plugin serait accusé d'altération alors que rien n'aurait bougé.

Métadonnées : anglais au catalogue, bilingue à l'exécution

Les champs `name`, `description` et les libellés de `contributes` s'écrivent en anglais. Ce sont les textes que voient le catalogue, les modérateurs et l'écran de consentement, au même titre que le reste d'un code destiné à circuler. L'interface que le plugin dessine lui-même reste libre, et les plugins livrés choisissent leur langue à l'exécution, en lisant celle de l'application.

Champs hérités

Trois noms de la première génération sont encore lus, et ne devraient plus être écrits. `main` tient lieu de `entry`. `permissions` tient lieu de `capabilities`. `language` n'a plus de raison d'être, le langage se déduisant de l'extension du point d'entrée, et il n'est consulté que lorsque cette extension ne tranche pas.

4. Les quatre voies d'exécution

Le même gestionnaire installe, charge et décharge quatre sortes de plugins, qui ne s'exécutent pas au même endroit. Le confinement diffère, le consentement non : les quatre passent par le même écran de capacités et la même vérification de signature.

Voie	Où le code s'exécute	Confinement	Ce qui la déclenche
Bac à sable	Dans une iframe sans <code>allow-same-origin</code> , sur une origine opaque. Toute interaction passe par le protocole de messages.	Complet. Ni DOM, ni stockage, ni réseau local.	Le cas par défaut : un point d'entrée en <code>.js</code> .
Python	Dans la page, sous Pyodide, dans un espace de noms dédié où <code>isofind</code> est lié à un porteur de capacités.	Aucun. La barrière de capacités vaut consentement et cohérence, pas isolation.	Un point d'entrée en <code>.py</code> .
Native	Dans la page, par <code>new Function(code)(plugin)</code> . Le code reçoit son propre enregistrement et s'installe lui-même.	Aucun.	<code>"runtime": "isofind-native"</code> dans le manifeste.
Atelier	Comme la voie du bac à sable, mais le code est lu depuis un dossier de travail et rechargé à chaque modification.	Complet pour le plugin ; le service qui lit le dossier, lui, ne l'est pas.	Un dossier ouvert dans l'atelier, en zone locale seulement.

Deux voies s'exécutent dans la page, et cela se voit. Python et natif ne sont pas confinés. Ils gardent la vérification de signature, le consentement de capacités et le journal des accès, ce qui répond à « d'où cela vient » et à « qu'est-ce que l'utilisateur a accordé », mais pas à « que peut-il atteindre s'il en décide autrement ». Le chargement le dit en clair dans la console, et l'interface le marque. Un plugin qui n'a pas besoin de sortir du bac à sable ne devrait pas en sortir.

Le langage se déduit de l'extension du point d'entrée, et non d'un champ. Le manifeste est scellé, il n'a pas de champ langage, et une déclaration qui contredirait l'extension ferait chercher longtemps pourquoi un fichier `.py` passe par le pont JavaScript.

La voie Python, en une phrase

Une façade `isofind` est définie une fois dans le runtime partagé, alignée sur la surface du pont : mêmes méthodes, mêmes capacités, deux langages. Chaque plugin reçoit une instance liée à son propre porteur de capacités, et son code s'exécute dans un espace de noms conservé après la fin du chargement. Sans cette conservation, les rappels enregistrés pour un menu ou une formule ne seraient plus appelables une fois la première exécution terminée. Le déchargement libère l'espace de noms et détruit les `PyProxy` retenus, faute de quoi le runtime Python garderait indéfiniment les fonctions enregistrées.

5. Le protocole du pont

Douze messages, et rien d'autre. Chacun porte un type dans le champ `t` et l'identifiant du plugin. L'hôte reconnaît l'émetteur par la fenêtre d'origine du message, et non par ce que le message déclare : un plugin qui se prétendrait un autre parlerait quand même depuis sa propre iframe.

Vers le bac à sable	Ce qu'il transporte
<code>init</code>	Le code du plugin, ses capacités accordées, son nom, l'URL de base de ses ressources, l'instantané de son stockage, celui des échantillons, et le réglage du moteur R.
<code>theme</code>	La feuille de jetons recalculée, à chaque bascule de thème.
<code>rpc-result</code>	La réponse à un appel, avec son identifiant, un drapeau de succès, et la valeur ou le message d'erreur.
<code>invoke-menu</code>	L'utilisateur a cliqué une entrée de menu du plugin.
<code>open-panel</code>	Un panneau doit se peindre. Émis par l'ouverture demandée comme par la fabrique du dock.
<code>close-panel</code>	Le panneau n'est plus visible. L'état du plugin n'est pas détruit pour autant.
<code>invoke-formula</code>	Une cellule a demandé une formule contributive, avec ses arguments.
Vers l'hôte	
<code>ready</code>	Le document du bac à sable est prêt à recevoir son initialisation.
<code>register</code>	Une contribution : <code>formula</code> , <code>menu</code> ou <code>panel</code> , avec sa fiche.
<code>rpc</code>	Un appel de méthode hôte, avec son nom et ses arguments.
<code>request-open-panel</code>	Le plugin demande l'ouverture d'un de ses panneaux.
<code>formula-result</code>	La valeur d'une formule, qui ira dans le cache et déclenchera un recalcul de la feuille.
<code>log</code>	Une trace, préfixée au nom du plugin dans la console de l'hôte.
<code>error</code>	Une erreur non rattrapée dans le bac à sable.

Un chargement, du premier message au premier clic



Ce que le pont fait pour chaque appel

Un appel entrant est traité en quatre temps, dans cet ordre. On cherche la méthode dans la table des méthodes hôtes : absente de la table, elle est refusée, ce qui rend le refus implicite plutôt que déclaré ailleurs. On vérifie que la capacité exigée par cette méthode figure dans l'ensemble accordé. On ouvre une portée de déclaration d'accès autour de l'exécution, qui nomme le plugin, la capacité et la méthode. On répond, en attendant la résolution si la méthode rend une promesse.

Ce dernier point vaut d'être dit. Répondre sans attendre enverrait au bac à sable une promesse sérialisée en objet vide, et le plugin croirait son calcul terminé avec un résultat nul. C'est la panne la plus difficile à diagnostiquer, celle où rien ne signale l'erreur.

Ce que le bac à sable fait de son côté

Le code du plugin n'est pas évalué au niveau du document. Il est enveloppé dans une fonction qui reçoit deux arguments, et l'appel ressemble à ceci :

```
new Function('isofind', 'plugin', '"use strict";\n' + code)(isofind, contexte);
```

`isofind` est la surface décrite en section 17. `plugin` porte ce que l'initialisation a livré sur l'identité de l'instance : son identifiant, son nom, l'URL de base de ses ressources, la liste des capacités accordées, et un raccourci de notification. Un plugin n'a donc rien à importer ni à chercher dans `window`.

Cette portée est une commodité, et le commentaire du fichier le dit sans détour : l'isolement réel vient de l'origine opaque de l'iframe, pas d'une fonction anonyme. Un plugin qui écrirait dans `window` ne gênerait que lui-même.

Deux gardes encadrent la réception. Un message dont la source n'est pas la fenêtre de l'hôte est ignoré : sans ce test, une iframe soeur atteint celle-ci par `parent.frames[i]`, qui est accessible entre origines, et lui injecte du code qui s'exécuterait ensuite sous les capacités accordées à ce plugin, puisque l'hôte attribue un appel à l'instance dont la fenêtre a émis et non à celle qui a décidé d'émettre. Et un second `init` est refusé : il réinstallerait la surface et exécuterait un autre code dans le même contexte, donc avec le stockage et les échantillons déjà livrés au premier.

Les contributions restent locales. Une formule enregistrée garde sa fonction dans le bac à sable ; seule sa fiche part vers l'hôte, qui se charge de l'enveloppe synchrone, du sélecteur et de l'aide. Il en va de même d'un panneau : l'hôte ne reçoit que son identifiant et son libellé, la fonction de rendu ne franchit jamais la frontière.

6. Contrat et implémentation

Un plugin installé est du code figé qui cite des noms, et ces noms sont résolus au chargement, parfois des mois plus tard, sur une version de l'application qui n'existait pas quand il a été écrit. La frontière entre ce qui est promis et ce qui est libre de bouger mérite donc d'être explicite.

Objet	Contrat	Implémentation
-------	---------	----------------

Manifeste	Le nom des champs lus, le vocabulaire des capacités, la forme de contributes , le fait que le manifeste installé soit conservé tel quel.	Les champs recopiés à plat sur la fiche, l'ordre des étapes d'installation, le texte des messages.
Pont	Le nom et le sens des douze messages, le nom des méthodes hôtes et leur capacité, le fait qu'une capacité non accordée fasse échouer l'appel.	Le transport, la numérotation des appels, la façon dont l'instantané est constitué, le contenu exact des messages d'erreur.
Contributions	Les trois genres, leur capacité, le fait qu'une contribution disparaisse avec son plugin sans laisser de trace.	La surface où elle atterrit, l'ordre dans les listes, l'icône, la zone de dock par défaut.
Ressources	Le préfixe d'URL, le fait qu'elles soient servies hors ligne et jamais téléchargées, la clef d'empreinte, qui est le chemin dans l'archive.	Le nom du cache, les en-têtes posés, le moment de l'extraction.
Thème	La liste des jetons exposés, et le fait qu'ils existent dans le document du bac à sable dès le premier rendu.	Leur valeur, qui change avec le thème, et le mécanisme d'injection.

Ce qui casse des plugins déjà installés

Renommer un identifiant de menu ou de panneau. L'ordre et le masquage de l'accès rapide sont mémorisés sous une clef qui joint l'identifiant du plugin et celui de l'entrée. Un renommage perd la préférence de l'utilisateur, sans erreur, et son entrée réapparaît à la fin de la liste après avoir été masquée.

Retirer une capacité du vocabulaire connu. Elle n'est pas refusée, elle cesse d'exister pour tout le monde : l'installation l'écarte de l'écran de consentement avec un avertissement en console, et les appels correspondants échouent ensuite sur une capacité que l'utilisateur n'a jamais pu accorder.

Changer le préfixe des ressources ou le nom du cache. Les plugins déjà installés gardent une URL de base calculée à l'ancien préfixe, et leur moteur embarqué ne trouve plus rien. Le service worker répond alors une explication plutôt qu'un 404 muet, mais il faut réinstaller.

Toucher au contenu du dossier de ressources après signature. Les empreintes sont recalculées depuis le Cache Storage, c'est-à-dire depuis les octets qui seront réellement servis. Un fichier ajouté, retiré ou modifié fait diverger la carte, et le plugin est refusé au chargement.

Retirer un jeton de thème de la liste exposée. Un plugin qui l'écrivait retombe sur sa valeur de repli, quand il en a prévu une, ou perd ses couleurs. La liste est fermée pour cette raison : tout exposer ferait du socle entier une surface publique qu'on ne pourrait plus faire évoluer.

Ce que l'atelier permet de vérifier, et que rien d'autre ne permettait. Le manifeste déclare ce qu'un plugin apporte, et le code l'implémente. Deux choses distinctes, donc deux choses qui peuvent diverger. Une contribution déclarée mais jamais enregistrée peut n'être qu'un travail en cours. Une contribution enregistrée sans avoir été déclarée est plus grave : elle échappe à l'écran de consentement, où l'utilisateur ne l'aura jamais vue. L'inspecteur de la section 20 confronte les deux listes.

Étendre

Ce qu'un plugin ajoute à l'application, où cela atterrit, et ce dont il dispose pour tenir dans la durée.

7. Les contributions

Trois genres, et trois seulement. Chacun relève d'une capacité, se déclare dans le manifeste, s'enregistre à l'exécution, et se retire sans résidu au déchargement.

Contribution	Capacité	Ce qu'elle pose dans l'application
Menu	<code>ui:menu</code>	Une entrée d'accès rapide dans la barre latérale, réordonnable et masquable, et une ligne dans le lanceur. Le clic revient au plugin sous forme de message.
Panneau	<code>ui:panel</code>	Une entrée au registre du shell, la même porte que les panneaux du noyau : menu Panneaux, palette, recherche universelle, docks et zone centrale.
Formule	<code>ui:formula</code>	Un nom de fonction dans le tableur, une fiche dans l'aide des formules, et une entrée dans le sélecteur de fonctions, groupée sous un intitulé choisi.

```
// Les trois, du côté du plugin. La signature est la même en Python,
// en nommage serpent : register_menu, register_panel, register_formula.

isofind.registerMenu('ouvrir', 'Open the panel', function () {
    isofind.openPanel('principal');
});

isofind.registerPanel('principal', 'My Plugin', function (ctx) {
    // Le rendu reçoit { root, id } : c'est ctx.root qu'il faut peupler.
    var racine = ctx.root;

    racine.textContent = 'Bonjour';
});

isofind.registerFormula('MON_CALCUL', function (a, b) { return a * b; },
    'Mon plugin', 'Produit', 'MON_CALCUL(a, b)', 'MON_CALCUL(2, 3)');
```

La formule est un cas particulier

Le tableur évalue ses formules de manière synchrone, dans la page, alors que le bac à sable ne répond qu'en asynchrone. L'hôte pose donc dans la page une enveloppe synchrone qui sert la valeur en cache si elle existe, et sinon en demande le calcul et rend `NaN`. La réponse remplit le cache et déclenche un rafraîchissement de la feuille, groupé pour ne pas multiplier les recalculs quand plusieurs formules se résolvent ensemble. La cohérence est donc différée, et exacte après un cycle de recalcul.

Deux garde-fous encadrent le nom. Il doit répondre à `^[A-Z][A-Z0-9_]{1,30}$` : une formule est un identifiant de feuille de calcul, pas une adresse dans `window`, et sans contrainte de forme un plugin désignerait n'importe quel objet global de l'application. Et le nom doit être libre, ou déjà tenu par une

enveloppe de plugin. Le contrôle initial ne couvrait que les fonctions : tout global qui n'en était pas une, à commencer par les objets de service, se laissait donc écraser en silence par un plugin pourtant confiné.

Contribuer une formule est une capacité. Elle a longtemps été la seule des trois contributions à ne rien exiger, ce qui n'avait pas de sens : une formule pose un nom dans la page et une entrée dans le sélecteur du tableur, ce qui est au moins aussi visible qu'un menu.

8. Où s'ouvre un panneau

Un panneau de plugin peut vivre à trois endroits : la zone centrale à onglets, un dock latéral ou inférieur, une fenêtre flottante. Le plugin n'en sait rien et n'a pas à le savoir. Il déclare un panneau, il le peint quand on le lui demande, et c'est l'hôte qui décide de la surface.

La zone de dock se déclare dans le manifeste, sous `contributes.panels[].dock`, et non dans l'appel JavaScript : celui-ci ne prend que trois arguments simples et ne transporte pas cette information. La valeur par défaut est `right`, qui est l'endroit où les panneaux de plugins se rangent déjà.

Valeur	Effet
<code>left</code>	Dock de gauche, à côté des panneaux du noyau.
<code>right</code>	Dock de droite. Défaut si le champ est absent.
<code>bottom</code>	Dock inférieur, pour une console ou un journal.

Le champ ne décide pas de l'existence du panneau, seulement de sa zone par défaut. Tout panneau déclaré entre au registre, avec ou sans `dock`. Sans cette règle, un panneau restait cantonné à la modale : absent du menu Panneaux, de la palette, du lanceur, et surtout de la zone centrale, qui est pourtant la seule surface assez grande pour une console ou un éditeur. Trois cent quarante pixels de large ne suffisent pas à un éditeur, et c'est ce qui les avait laissés en modale.

Une seule iframe, donc un seul panneau visible

Un plugin n'a qu'un bac à sable. Deux de ses panneaux ne peuvent donc pas être affichés ensemble : le second recible l'iframe, et le premier deviendrait un rectangle vide sans que rien ne le dise. L'hôte ferme donc explicitement l'autre, et par la surface où il se trouve, la zone centrale d'abord, le dock ensuite. Une surface qui disparaît sur un geste se comprend ; une surface vide qui reste ouverte se met sur le compte du plugin.

L'iframe n'est jamais déplacée. Elle vit en permanence sous `document.body`, en `position: fixed`, masquée tant qu'aucun conteneur ne la vise. Ouvrir un panneau ne la reparente pas : une boucle de synchronisation recopie, à chaque image, le rectangle du conteneur cible sur les coordonnées de l'iframe. Reparenter un iframe la recharge, et le plugin perdrait son état à chaque changement d'onglet.

Le plan d'affichage suit la surface. Un panneau docké est posé très bas, pour qu'une vraie modale de l'application le recouvre et l'assombrit comme le reste du shell. La modale du plugin lui-même est posée juste au-dessus de son propre fond assombri. Un plan unique et élevé pour les deux cas donnait l'un et l'autre défaut à la fois : le panneau docké flottait au-dessus des modales natives, et l'iframe passait sous son propre voile en mode modal.

Deux courses, et comment elles se perdent

Le shell n'est pas encore là. Il s'installe quand le DOM est prêt ; un plugin se déclare dès que son bac à sable répond, ce qui dépend de sa taille et de ce qu'il charge. Les plugins légers arrivaient après le shell et s'inscrivaient ; ceux qui répondaient tôt trouvaient porte close et retombaient dans la modale, définitivement, sans autre signal qu'un avertissement de console. L'enregistrement attend donc le shell plutôt que d'abandonner, et rejouer l'enregistrement est sans effet de bord.

Le panneau n'est pas déclaré au chargement. Certains plugins déclarent tout d'emblée, d'autres construisent leur surface au premier usage, et ils ont raison de le faire. Un panneau demandé mais pas encore inscrit entre donc au registre à cet instant, plutôt que de retomber sur la modale parce que l'ordre ne lui convenait pas.

La modale reste, en dernier recours. Sur une page qui n'a ni dock ni zone centrale, c'est la seule surface disponible.

9. Thème, langue et instantané

Un plugin ne code pas de couleurs. Il écrit `var(--surface-1)` comme un composant natif, et c'est l'hôte qui garantit que ces variables existent dans le document du bac à sable. Comme l'origine est opaque, la seule voie est le `srcdoc` à la création, puis un message à chaque bascule de thème. Sans les valeurs initiales, le plugin rendrait une première fois avant l'arrivée du message, donc sans couleurs.

Les jetons exposés

Famille	Jetons
Surfaces	<code>--surface-1</code> , <code>--surface-2</code> , <code>--surface-3</code> , <code>--surface-inverse</code> , <code>--surface-inverse-soft</code>
Texte	<code>--text-primary</code> , <code>--text-secondary</code> , <code>--text-muted</code> , <code>--text-inverse</code> , <code>--text-on-solid</code>
Bordures	<code>--border-subtle</code> , <code>--border-primary</code> , <code>--border-strong</code>
Ombres	<code>--shadow-1</code> à <code>--shadow-4</code>

Marque	<code>--brand-primary, --brand-primary-dark, --brand-primary-soft, --brand-accent, --brand-on-primary, --brand-halo, --brand-tint-50, --brand-tint-100</code>
Tons	<code>--tone-{vert, ambre, orange, rouge, bleu, cyan, violet}-{bg, bg-strong, solid, solid-strong, solid-deep, text, border, on-solid}</code>
Hérités	<code>--border-color, --bg-primary, --bg-secondary, --surface, --text, --hover</code>

La liste est délibérément fermée. Les six derniers noms sont des alias de la première génération, exposés en transition parce que des plugins déjà publiés les emploient : les couper pour une question de propreté casserait l'existant. Ils disparaîtront quand ces plugins auront migré.

Une valeur de jeton vient d'une feuille de l'hôte, mais elle traverse une frontière de sécurité. Seul ce qui ressemble à une couleur ou à une ombre passe, jamais une accolade ni un point-virgule, qui permettraient de sortir de la déclaration et d'écrire du style arbitraire dans le document du bac à sable.

La langue

La langue de l'application n'appartient à aucun plugin, mais tous la lisent pour s'afficher dans la bonne. Elle voyage donc avec l'instantané du stockage, sous la clef `isofind_language`, dès que `storage:read` est accordée. La refuser forcerait les plugins à retomber sur le français, ou à lever. Comme elle arrive dans l'instantané, elle se lit sans aller-retour, au premier rendu, par la façade de compatibilité.

L'instantané des échantillons

Il est livré avec le message d'initialisation, quand `samples:read` est accordée, et il est vide sinon. Vide, et non absent : le plugin lit la clef sans se demander si elle existe, et la supprimer le ferait lever au lieu de le laisser afficher une liste vide.

Un échantillon y arrive avec ses mesures isotopiques attachées. La méthode `getAnalyses` n'interroge pas une autre source : elle aplatit les mesures des échantillons, chacune devenant une ligne portant le contexte de son échantillon. Il n'existe pas de stock d'analyses distinct, une analyse étant une mesure isotopique.

10. Stockage et ressources embarquées

Un plugin qui doit retenir quelque chose ne pouvait pas le faire : le `localStorage` est inaccessible depuis une origine opaque, et c'était l'une des deux raisons pour lesquelles la voie hôte survivait. Un stockage lui est donc offert par le pont, sous capacité.

Appel	Capacité	Effet
<code>isofind.storage.get(clé)</code>	<code>storage:read</code>	Rend la valeur, ou rien.
<code>isofind.storage.keys()</code>	<code>storage:read</code>	Les clefs du plugin, sans le préfixe.
<code>isofind.storage.set(clé, valeur)</code>	<code>storage:write</code>	Écrit, ou rejette si le quota est atteint.

`isofind.storage.remove(clé)` `storage:write` Retire une clef.

Les quatre rendent une promesse, puisqu'elles franchissent la frontière. Deux raccourcis les doublent, `getJSON` et `setJSON`, parce que la quasi-totalité des usages du stockage sont du JSON sérialisé à la main. Et la façade de compatibilité expose par-dessus un `localStorage` synchrone, adossé à l'instantané livré à l'initialisation, décrit en section 17.

L'espace de noms est dérivé de l'instance côté hôte, jamais d'un argument fourni par le plugin. C'est la seule chose qui compte ici : si la clef était choisie librement, un plugin lirait l'espace d'un autre, et le cloisonnement serait décoratif. Les valeurs vivent sous `isofind_plugin_store:<id>:<clef>`, avec un plafond de deux mégaoctets par plugin. Le plafond n'est pas une politesse : sans lui, un plugin remplit le stockage de l'origine, et c'est l'application entière qui cesse de pouvoir écrire, y compris ses propres préférences.

Désactiver n'est pas désinstaller. Le stockage d'un plugin est purgé à la désinstallation, jamais au déchargement. Désactiver un plugin le rend muet ; cela ne doit pas lui faire perdre la mémoire. Confondre les deux ferait disparaître le travail d'un utilisateur qui voulait simplement mettre un plugin en veille.

Les ressources embarquées

Un moteur en WebAssembly charge ses binaires par URL. Il ne sait pas lire un Cache Storage, et on ne peut pas lui passer un tableau d'octets : il fait une requête, et il faut que quelqu'un réponde. C'est le rôle du provisionnement de ressources, qui est une capacité de plateforme et non un arrangement propre à un plugin.

À l'installation, les fichiers du dossier déclaré par `assets` sont extraits du paquet et déposés dans le Cache Storage sous des URL virtuelles préfixées par `/__plugin-assets/<id>/`. Le plugin reçoit cette base dans son message d'initialisation, sous `assetsBaseUrl`, et y charge ce dont il a besoin. Il n'y a ni dossier `vendor`, ni script de placement, ni intervention de l'utilisateur.

Le chemin d'une ressource, de l'archive au moteur



`/__plugin-assets/<id>/assets/moteur.wasm`

Une seule URL, la même à chaque lancement : le cache persiste, l'extraction n'a lieu qu'une fois.

Le service worker ne va jamais sur le réseau

Il n'y a aucun repli. Un service worker qui, faute de trouver une ressource en cache, irait la chercher dehors serait une sortie réseau que rien n'annonce, dans un logiciel qui promet de n'en avoir aucune. Absente du cache, la ressource est absente : on répond 404, et on dit pourquoi, en toutes lettres, parce qu'un moteur WebAssembly qui reçoit un 404 muet échoue sur une erreur de compilation incompréhensible et l'utilisateur croit le plugin corrompu.

C'est aussi ce qui protège la signature. Les empreintes ont été vérifiées sur le contenu déposé dans le cache. Servir autre chose que ce contenu, fût-ce le fichier authentique récupéré ailleurs, rendrait la vérification

décorative. Et seules les URL sous le préfixe sont interceptées : l'application, l'API et les fichiers statiques suivent leur chemin habituel, un service worker qui met en cache ce qu'on ne lui a pas demandé étant un service worker qu'on finit par désinstaller à la main un jour de mise à jour.

La purge précède le dépôt. Sans elle, les ressources d'une version antérieure survivent à côté des nouvelles : écrire dans le cache écrase les URL identiques mais ne supprime pas celles qui ont disparu de l'archive. La carte d'empreintes recalculée comporte alors des entrées en trop, elle ne correspond plus à celle qui a été signée, et le plugin est accusé d'altération alors qu'il trainait simplement les restes de son prédécesseur.

Les empreintes

Chaque ressource en cache est condensée en SHA-256, et la carte obtenue est indexée par le chemin dans l'archive, tel que le signataire l'a vu, donc `assets/moteur.wasm` et non l'URL virtuelle sous laquelle le cache la range. Le chemin d'une URL est encodé : un nom de fichier accentué ou espacé en ressort en pourcentage, et sans décodage la clef ne correspondrait pas, le fichier passant pour manquant.

Un cache illisible n'est pas une absence de ressources. Rendre un dictionnaire vide ferait échouer la signature avec une raison trompeuse ; on rend donc l'absence de carte, que le vérificateur traduit en refus explicite plutôt qu'en altération supposée.

11. Le moteur R

`compute:r` accorde l'exécution de code R arbitraire, et c'est assumé. R tourne en WebAssembly, dans un travailleur sans DOM ni jeton : il calcule, il ne peut pas atteindre l'application. La capacité reste séparée de l'accès aux données, et l'avoir ne donne ni `samples:read` ni le droit d'écrire quoi que ce soit.

Le moteur ne vit pas dans le bac à sable. WebR crée lui-même un travailleur, et une origine opaque ne peut pas en créer de façon fiable ; il pèse par ailleurs plusieurs dizaines de mégaoctets, qu'on ne rechargera pas une fois par plugin. Il vit donc dans un travailleur de l'hôte, et le bac à sable en présente la surface habituelle pour que le code du plugin l'appelle sans changer.

Appel	Effet
<code>WebRManager.charger(demande)</code>	Amorce le moteur : version épinglée, et éventuellement l'URL du WebR embarqué dans les ressources du plugin. Rend un objet de moteur. C'est le seul chemin d'amorçage.
<code>moteur.evalR(code)</code>	Évalue et rend un objet dont <code>toJs</code> donne la valeur convertie.
<code>moteur.evalRVoid(code)</code>	Évalue sans rien rendre, pour les effets de bord.
<code>moteur.evalRBoolean(code)</code>	Évalue et rend un booléen.
<code>moteur.captureR(code, options)</code>	Évalue en capturant la sortie console, et rend la sortie avec la valeur.
<code>moteur.installPackages(paquets, options)</code>	Installe des paquets R dans l'environnement du plugin.

<code>isofind.r.eval(code, options)</code>	Voie directe, une fois le moteur amorcé. Rend la sortie texte ; <code>{ value: true }</code> ajoute la valeur convertie, ce qui évite de sérialiser des objets R volumineux à chaque appel.
<code>isofind.r.text(code)</code>	La sortie seule, concaténée. C'est ce que veut une console.

L'identifiant de plugin employé pour résoudre l'environnement vient de l'instance, jamais d'un argument. C'est lui qui détermine l'environnement R dans lequel le code s'exécute, et le laisser choisir à l'appelant reviendrait à lui laisser lire l'environnement d'un autre plugin partageant le même runtime.

L'URL du moteur embarqué n'est honorée que si la signature du plugin couvre ses ressources. Un moteur qu'on charge sur la foi d'un chemin déclaré dans un manifeste non vérifié n'est pas un moteur souverain, c'est un exécutable choisi par l'auteur du paquet.

Un exemple qui tourne en production. La console PHREEQC fait tourner PHREEQC dans WebR, sans aucun exécutable externe. Elle déclare `samples:read`, `ui:menu`, `ui:notify`, `compute:r`, `storage:read` et `storage:write`, et embarque son moteur dans ses ressources.

Gouverner

Ce qu'un plugin a le droit de toucher, d'où il vient, et ce que le journal en retient.

12. Capacités et consentement

Un paquet signé dit qui a écrit le plugin. Il ne dit pas ce que ce plugin touche, et c'est l'autre question. Elle est ramenée à une poignée d'accès nommés, demandés par le manifeste, présentés case par case à l'installation, et vérifiés à chaque appel.

Le vocabulaire complet

Capacité	Ce qu'elle accorde	
<code>samples:read</code>	Lire les échantillons chargés : l'instantané livré à l'initialisation, et les mesures qui en sont dérivées.	lecture
<code>samples:write</code>	Créer de nouveaux échantillons dans le projet.	écrit
<code>analysis:run</code>	Déclencher le moteur d'analyse avec des paramètres.	écrit
<code>config:read</code>	Consulter les réglages de l'application.	lecture
<code>config:write</code>	Enregistrer des réglages de l'application. À distinguer du stockage propre au plugin, qui relève de <code>storage:write</code> .	écrit
<code>ui:notify</code>	Émettre des messages à l'écran.	lecture
<code>ui:menu</code>	Poser des entrées de menu, donc une entrée d'accès rapide et une ligne au lanceur.	lecture
<code>ui:panel</code>	Déclarer et ouvrir des panneaux, dans les docks comme dans la zone centrale.	lecture
<code>ui:formula</code>	Ajouter des formules au tableur, avec leur fiche d'aide et leur groupe dans le sélecteur.	lecture
<code>storage:read</code>	Relire son propre stockage, et connaître la langue de l'application.	lecture
<code>storage:write</code>	Conserver ses propres réglages, dans la limite de son quota.	écrit
<code>compute:r</code>	Calculer dans un moteur R isolé, sans accès aux données.	lecture
<code>network</code>	Établir des connexions sortantes, en https : seulement, et jamais vers l'origine locale. Refusée d'office en mode air-gap.	écrit

Une capacité hors vocabulaire n'existe pour personne

Elle n'est refusée par aucun code : elle est simplement écartée de l'écran de consentement, avec un avertissement en console. L'afficher reviendrait à faire consentir l'utilisateur à un intitulé inventé par l'auteur

du plugin, ce qui donne l'apparence d'un contrôle sans en avoir la substance.

Le vocabulaire n'est pas tout à fait celui des workflows. Les deux systèmes partagent `samples:read`, `samples:write` et `ui:notify`, qui désignent la même chose et n'ont aucune raison de porter deux noms. Le reste diverge parce que les surfaces diffèrent : un workflow écrit des fichiers et lance des solveurs, un plugin pose des formules et calcule en R. Un plugin qui déclare une tâche de workflow, lui, range sa capacité dans le vocabulaire des workflows, qui est celui de son registre d'accueil.

Comment le consentement est présenté

Toutes les cases sont cochées par défaut, sauf celles que le mode air-gap rend impossibles à honorer, qui apparaissent grisées avec la raison plutôt que d'être retirées. Ce que l'utilisateur décoche sera refusé à l'exécution, avec un message qui nomme la capacité manquante. Annuler l'écran annule l'installation entière.

Le nom du plugin, ses capacités et leurs libellés arrivent dans une fenêtre construite par concaténation de texte. C'est le moment le plus sensible du cycle de vie, celui où l'utilisateur accorde des droits, et tout ce qui vient du manifeste y est échappé avant d'être affiché.

Où la capacité est vérifiée

Deux endroits, et ils ne se recouvrent pas. Le pont vérifie à chaque appel entrant, avant de chercher la méthode. Et l'enregistrement d'un panneau vérifie `ui:panel` au moment de la déclaration, non seulement à l'ouverture : un panneau muni d'une zone de dock n'est jamais ouvert par le plugin, c'est la coque qui appelle sa fabrique. Le manifeste déclare la zone, il ne peut pas déclarer le droit, puisqu'il est écrit par l'auteur du plugin.

13. Signature, identité d'auteur et provenance

La signature répond à deux questions à la fois : qui a produit ce paquet, et est-ce que ce qui va s'exécuter est bien ce qui a été produit. La seconde est la moins évidente, et c'est celle qui commande l'endroit où les empreintes sont calculées.

Ce qui est signé

Le manifeste, le code du point d'entrée, et la carte d'empreintes des ressources. Le manifeste porte cette carte à titre informatif, sous `signature.assets_sha256`, mais l'hôte ne la croit pas : il recalcule tout depuis le Cache Storage, c'est-à-dire depuis les octets qui seront réellement servis au plugin et exécutés par lui. C'est le seul endroit où le calcul a un sens. Vérifier l'archive téléchargée prouverait qu'elle était saine à l'instant du téléchargement, pas que ce qui s'exécutera l'est.

La carte déclarée sert quand même, une fois. Comparée à celle qu'on trouve, elle permet de nommer le fichier fautif : absent du cache, en trop, ou de contenu différent. Un refus de signature est sinon opaque, il dit qu'il y a un écart, jamais lequel, et nommer le fichier est la seule chose utile à qui doit corriger.

Qui exige une signature

L'exigence est décidée par l'appelant, seul à connaître la provenance de l'installation et l'état du mode développeur. Elle n'est plus une propriété globale du pont : une politique en écriture libre sur un objet de `window` se désarme d'une ligne depuis n'importe quel script de la page. Un enregistrement muet vaut exigence, faute de quoi l'oubli du champ rouvrirait sans bruit ce qu'on ferme là.

Situation	Signature exigée
Cas courant	Oui.
Réglage <code>verifySignatures</code> mis à faux	Non. Seul un refus explicite lève l'exigence ; un réglage absent, corrompu ou d'une version antérieure vaut exigence.
Mode développeur, plugin local	Non, et la dérogation est tracée en console. Une dérogation qui ne laisse rien derrière elle est une dérogation qu'on oublie avoir accordée.
Mode développeur, plugin venu du catalogue	Oui. Un plugin du catalogue est signé par construction : si sa vérification échoue, ce qui s'exécuterait n'est plus ce qui a été signé, et aucun réglage local n'a de raison de couvrir ce cas.

Est local tout ce qui n'a pas été installé depuis le catalogue, import de fichier compris. Seul le chemin d'installation depuis le catalogue pose cette provenance sur la fiche.

Le vérificateur ne s'installe qu'une fois. Autoriser son remplacement laisserait n'importe quel script de la page rendre un verdict favorable à volonté, ce qui viderait la vérification de son sens sans rien changer de visible.

Une indisponibilité ne vaut pas un succès. Un service de vérification injoignable, une réponse en erreur, un manifeste sans champ `signature` : tous rendent un verdict non valide, avec une raison distincte. C'est ensuite l'exigence qui décide si le plugin se charge quand même, et l'interface affiche « non vérifié » plutôt que rien.

L'identité d'auteur

Pour signer, il faut une clef et un certificat. La clef est produite par le service local, en Python, et écrite chiffrée dans le répertoire de données. Elle ne transite par aucun réseau, ce qui est la seule propriété qui compte : une clef de signature qui existe en deux exemplaires ne prouve plus rien, et révoquer le certificat d'un auteur qui n'a jamais détenu sa clef ne révoque rien du tout. L'interface expose cette contrainte plutôt que de la cacher, faute de quoi un auteur qui ne la comprend pas finit par envoyer sa clef par courriel, en croyant rendre service.

Étape	Ce qui se passe
1	Nom, organisation, courriel, phrase de passe d'au moins douze caractères. La clef et la demande de certificat sont produites localement.
2	La demande est transmise à IsoFind. Elle ne porte que la clef publique et le nom : elle peut circuler sans risque, elle ne permet pas de signer.

-
- 3 Le certificat renvoyé est installé. La chaîne d'autorité est déjà embarquée dans chaque installation d'IsoFind : la faire transiter par courriel n'apportait qu'une occasion d'oublier un fichier, et c'était l'erreur la plus fréquente.
-

Un auteur qui possède déjà une clef l'importe plutôt que d'en créer une autre. C'est la seule façon de continuer à signer les mises à jour de plugins déjà publiés : ceux-ci resteraient valides, mais leurs correctifs ne parviendraient jamais à leurs utilisateurs. La clef importée est réécrite chiffrée avec la phrase de passe fournie, on ne recopie pas le fichier tel quel : rien ne garantirait qu'il l'était, et une clef de signature en clair sur un disque est une clef déjà perdue.

Le certificat expire. Un avertissement apparaît dans les soixante derniers jours, et une signature produite avec un certificat expiré est refusée. Régénérer une clef ne se fait qu'en cas de perte ou de compromission, et il faut alors demander aussi la révocation de l'ancien certificat : sans elle, celui qui détient la clef dérobée continue de signer au nom de son propriétaire.

Signer

L'archive est signée localement, puis vérifiée avant d'être rendue. Une archive qui ne se vérifie pas n'est pas restituée : la laisser sortir reviendrait à la faire publier et à découvrir le problème chez ses utilisateurs.

L'archive signée reste en mémoire plutôt qu'être téléchargée d'office, parce qu'un auteur enchaîne généralement sur la soumission et qu'un aller-retour par le disque n'apporterait qu'une occasion de soumettre le mauvais fichier, typiquement la version non signée restée à côté.

14. Journal des accès

Le journal répond à « qu'est-ce qui a touché cet échantillon ». Un plugin est le cas où la réponse est la plus sûre : il vit dans un bac à sable sans même origine, il ne parle pas HTTP, et il ne peut donc pas poser lui-même un en-tête d'identité. Ce que l'hôte déclare là, c'est ce que l'hôte a fait.

La portée est ouverte autour de l'appel, et non dans chaque méthode. C'est le seul endroit qui connaisse à la fois l'instance et la capacité exigée, et une méthode qui devrait le redire s'en dispenserait un jour sans que rien ne le signale, alors qu'un accès non déclaré ressemble à un accès qui n'a pas eu lieu.

Champ déclaré	Contenu
genre	plugin
id	L'identifiant du plugin, qui sert de clef de jointure.
capacite	La capacité exigée par la méthode appelée.
tache	Le nom de la méthode hôte.

Ces valeurs alimentent la même enveloppe d'en-têtes que les workflows, et le serveur les traite de la même façon : une déclaration, pas une preuve, avec un marqueur d'ambiguïté quand deux consommateurs entrelacent leurs requêtes. Le pont des plugins est d'ailleurs l'un des deux consommateurs hors workflow à ouvrir une portée explicitement, parce qu'il est le seul à savoir quel plugin il sert.

La déclaration de l'instantané

L'instantané des échantillons ne part par aucune requête. Le middleware du serveur ne le verrait donc pas, et le journal laisserait croire que le plugin n'a rien lu alors qu'il détient toute la base. Une déclaration distincte est donc posée, avec la nature `instantane`, la liste des identifiants concernés, le volume, et une note qui dit ce que la ligne signifie : une livraison unique à l'initialisation, dont la lecture ultérieure n'est observable nulle part.

L'hôte attend le verdict avant de livrer. C'est ce qui rend une révocation effective sur un chemin qui ne passe pas par le middleware. Si le journal est injoignable, on livre quand même : priver un plugin de ses données parce qu'un registre a bronché serait une panne plus grave que l'absence de la ligne qu'on voulait écrire.

15. Conflits, dépendances et compatibilité

Trois contrôles ont lieu avant le consentement, dans cet ordre : la compatibilité d'édition, les dépendances, puis les conflits. Les deux premiers peuvent installer autre chose ou refuser ; le troisième avertit et laisse décider.

Les conflits

Ce qui est cherché	Pourquoi	
Variables globales	Deux plugins qui posent le même nom se recouvrent, et le second gagne sans que le premier le sache.	
Noms de menus	Deux entrées identiques dans l'accès rapide sont indiscernables une fois posées.	
Capacités partagées	Deux plugins qui peuvent écrire les mêmes données peuvent se défaire mutuellement.	
Catégorie	Signale les doublons fonctionnels probables, sans les interdire.	
Motifs de code	Cinq motifs relevés dans le code, avec leur gravité.	
Motif	Gravité	Ce qu'il ferait
<code>localStorage.clear()</code>	Critique	Efface le stockage de toute l'application, pas seulement le sien.
<code>document.write()</code>	Élevée	Peut effacer la page.
<code>eval()</code>	Élevée	Exécute du code construit à l'exécution.
<code>innerHTML = ... <script</code>	Élevée	Injecte un script par une chaîne.
<code>\${ ... eval(</code>	Élevée	Évaluation dynamique dissimulée dans un gabarit.

Ce relevé est un avertissement, pas un verdict. Il porte sur le texte du code, donc il signale aussi bien un plugin dangereux qu'un plugin qui manipule ces motifs pour une raison légitime. Il sert à ce que l'utilisateur pose la question avant d'installer, pas à trancher à sa place.

Les dépendances

Un plugin peut déclarer d'autres plugins dont il a besoin. Ils sont résolus avant lui, en profondeur, avec une garde contre les cycles : le plugin courant est marqué dès l'entrée, pour qu'une dépendance qui pointe en retour vers lui ne déclenche pas sa réinstallation. Ce qui est présent au catalogue est proposé à l'installation, en une seule fenêtre ; ce qui n'y est pas est nommé, et reste à installer à la main. Annuler cette fenêtre annule l'installation entière.

La compatibilité d'édition

Un plugin peut déclarer les éditions d'IsoFind avec lesquelles il fonctionne. Le blocage n'a lieu que si l'édition courante est connue et absente de la liste déclarée : un plugin qui ne déclare rien passe partout, et une édition indéterminée ne bloque pas. Un contrôle qui refuserait dans le doute empêcherait d'installer sur les configurations qu'il ne sait pas nommer.

Le mode air-gap

Il ne se contente pas de couper le réseau, il ferme les portes une par une, chacune avec son message.

Ce qui est fermé	Ce qui reste
La bibliothèque en ligne et le registre distant	Le catalogue de première partie, embarqué, code inclus.
Le partage et la soumission au catalogue	L'empaquetage et la signature, qui sont locaux.
Le téléchargement de Pyodide depuis un CDN	Le paquet local servi par l'application, qui est la voie souveraine par défaut. Le CDN n'était qu'un repli de développement.
La capacité network	Rien. La politique de sécurité du contenu retombe à 'none' quelle que soit la case cochée.

16. Le gestionnaire et les surfaces d'accès

Quatre surfaces donnent accès aux plugins, et elles ne servent pas au même moment. Le gestionnaire administre, le lanceur ouvre, l'accès rapide raccourcit, la palette retrouve.

Le gestionnaire de plugins

Une fenêtre, six vues : les plugins installés avec leurs badges et leurs réglages, la bibliothèque, l'installation par fichier, l'assistant de création, la documentation intégrée et les réglages du socle. C'est là que se font l'activation, la désactivation, la mise à jour, la révision de l'accès rapide et la désinstallation.

Le rechargement se décide à la fermeture, pas au clic. Une installation retire sa fenêtre de progression par code, et une fenêtre retirée par le code ne déclenche aucun gestionnaire de fermeture : la décision de rechargement dépendait donc du geste de l'utilisateur, alors qu'elle dépend de l'état de l'application. Elle s'accroche maintenant à un fait observable, la disparition de toute fenêtre de plugin du document, ce qui couvre les voies de fermeture qu'on oublierait d'instrumenter.

Deux rechargements existent. Le doux décharge toutes les contributions puis recharge les plugins actifs, ce qui garantit un état cohérent, sans contribution orpheline ni doublon. Le matériel recharge le document, et il est exigé par les opérations qui ont pu déposer de nouvelles ressources en cache. Une seule décision est prise pour tout un lot d'opérations, ce qui permet d'enchaîner plusieurs installations.

Le panneau Plugins, ou lanceur

Il ne reconstruit pas une liste, il rend visible ce que les registres contiennent. Les actions viennent de la catégorie des commandes de plugins, les panneaux du registre de panneaux filtré sur leur source, et le tout est groupé par plugin. Un plugin apporté par un complément y figure au même titre qu'un autre, sans que rien n'ait à le savoir.

Chaque ligne dit ce qu'elle est en toutes lettres, action ou panneau, à droite du nom. Une extension nomme souvent son panneau et son action du même nom : la liste montrait alors deux fois le même libellé, que seule une icône de seize pixels distinguait. Le mot dit où l'on va, l'icône ne fait que le rappeler.

Ouvrir un panneau depuis le lanceur passe par l'arbitre de placement, jamais par l'ouverture directe. Lui seul libère les autres surfaces : sans lui, ouvrir au centre un panneau déjà flottant en ferait deux instances vivantes, chacune avec son propre état et son propre abonnement au registre.

L'accès rapide

La section Plugins de la barre latérale a un propriétaire unique, auquel les deux sources d'entrées délèguent, le pont pour les plugins en bac à sable et le registre de contributions pour les plugins hôte. L'ordre, le masquage et la persistance sont donc cohérents quelle que soit l'origine. Masquer une entrée la retire de l'accès rapide sans désactiver le plugin : son menu reste lançable depuis le lanceur et depuis le gestionnaire.

La vérité est dans la liste, pas dans le DOM. La barre latérale est partagée et peut être reconstruite par l'application ; les entrées sont donc réinjectées à partir de la liste plutôt que sur la supposition que le DOM persiste, avec quelques réessais différés puis un observateur qui ne réagit que lorsqu'une entrée manque réellement.

Le bouton de la barre de titre

La palette et le menu supposent tous deux qu'on sache où chercher. Un bouton toujours visible ne le suppose pas, et c'est ce qui manquait : rien, nativement, ne menait aux extensions. Il est posé dans la barre de titre plutôt que dans un dock, parce qu'elle est le seul chrome présent sur toutes les pages, l'atelier compris, et parce que son registre libère ses outils au déchargement d'une source.

Les commandes

Commande	Effet
----------	-------

<code>core.plugins.toggle</code>	Afficher ou masquer le lanceur.
<code>core.plugins.authorIdentity</code>	Identité d'auteur et signature.
<code>core.plugins.checkBeforeSubmit</code>	Vérifier la signature avant soumission.
<code>core.plugins.atelier</code>	Ouvrir l'atelier de développement, en zone locale seulement.

Comme toutes les commandes, elles entrent dans la palette, dans le panneau Actions et dans la recherche universelle, et chacune peut recevoir le raccourci que l'utilisateur veut.

Référence

Le catalogue des méthodes hôtes, le manifeste champ par champ, la publication, l'atelier, les plugins types et les bornes.

17. Catalogue des méthodes hôtes

Tout ce qu'un plugin peut demander à l'application, groupé par ce que cela exige. Cette liste tient lieu d'architecture : une méthode absente est une chose que personne ne pourra faire depuis un plugin, et une méthode de trop est une porte qu'il faudra défendre.

La première colonne donne l'appel tel qu'un auteur l'écrit dans le bac à sable, sur l'objet `isofind`. La deuxième donne son équivalent sur la façade du runtime Python partagé. Sous les deux se trouve la même méthode hôte, dont le nom apparaît dans le journal des accès et dans les messages de refus : `getSamples`, `storageSet`, `rEval`. Les deux surfaces sont alignées volontairement, mêmes méthodes et mêmes capacités pour deux langages.

Tout ce qui franchit la frontière rend une promesse. C'est la seule concession du confinement, et elle est inévitable : rien de ce qui traverse une frontière d'isolation ne peut être synchrone. Les contributions font exception, puisqu'elles restent locales et ne font qu'annoncer.

Sans capacité

Deux seulement, et elles ne sortent pas du bac à sable. `isofind.version` porte la version du protocole, et `isofind.log(...)` écrit dans la console de l'hôte, préfixée au nom du plugin.

Lire les échantillons `samples:read`

Bac à sable	Python	Arguments et retour
<code>isofind.getSamples()</code>	<code>get_samples()</code>	Les échantillons chargés. Sans argument.
par la compatibilité	<code>get_samples_df()</code>	Les mêmes, en <code>DataFrame</code> pandas côté Python. Rend un tableau vide s'il n'y a rien.
par la compatibilité	<code>get_analyses()</code>	Les mesures isotopiques aplaties, une ligne par mesure, avec le contexte de son échantillon. Dérivé des échantillons, pas d'une autre source.

Écrire dans le projet `samples:write`

Bac à sable	Python	Arguments et retour
<code>isofind.addSample(objet)</code>	<code>add_sample(dict)</code>	Crée un échantillon. Un objet sans <code>name</code> est refusé. Rend un booléen. Depuis Python, l'objet est sérialisé pour traverser la frontière, faute de quoi il arriverait en <code>Map</code> et serait ignoré.

Analyser `analysis:run`

Bac à sable	Python	Arguments et retour
<code>isofind.runAnalysis(params)</code>	<code>run_analysis(dict)</code>	Déclenche le moteur d'analyse. Rend son résultat, ou rien si aucun moteur n'est monté.

Réglages de l'application `config:read`, `config:write`

Bac à sable	Python	Arguments et retour
<code>isofind.getConfig(clé)</code>	<code>get_config(clé=None)</code>	Une valeur, ou la table entière si la clef est omise.
<code>isofind.setConfig(clé, valeur)</code>	<code>set_config(clé, valeur)</code>	Enregistre. Rend un booléen. Les structures sont sérialisées côté Python.

Interface `ui:notify`

Bac à sable	Python	Arguments et retour
<code>isofind.notify(message, type)</code>	<code>notify(message, type='info')</code>	Affiche un message. <code>info</code> , <code>success</code> , <code>warning</code> ou <code>error</code> . <code>show_notification</code> reste accepté en Python, sous son ancien nom. Le raccourci <code>plugin.notify</code> fait la même chose.

Contributions `ui:menu`, `ui:panel`, `ui:formula`

Bac à sable	Python	Arguments et retour
<code>isofind.registerMenu(id, libellé, rappel)</code>	<code>register_menu(...)</code>	Pose une entrée d'accès rapide. Le rappel reste dans le bac à sable et sera invoqué au clic. Exige <code>ui:menu</code> .
<code>isofind.registerPanel(id, libellé, peindre)</code>	<code>register_panel(...)</code>	Déclare un panneau. <code>peindre</code> reçoit un <code>CONTEXTE</code> , <code>{ root, id }</code> , et non l'élément lui-même : c'est <code>ctx.root</code> , un élément du document du bac à sable, qu'il faut peupler. Sa valeur de retour est ignorée, donc rendre une chaîne de HTML ne produit rien et le panneau reste blanc. Exige <code>ui:panel</code> , vérifiée dès la déclaration.
<code>isofind.openPanel(id)</code>	<code>open_panel(id)</code>	Demande l'ouverture d'un panneau déjà déclaré. L'hôte révèle la coque puis renvoie l'ordre de rendu.
<code>isofind.registerFormula(nom, fn, fiche)</code>	<code>register_formula(nom, fn, groupe, libellé, syntaxe, exemple)</code>	Ajoute une formule au tableur. Les deux surfaces divergent ici : le bac à sable prend un objet de fiche, avec les clefs <code>group</code> , <code>label</code> , <code>syntax</code> et <code>example</code> ; Python les prend en arguments successifs. Le nom doit répondre à <code>^[A-Z][A-Z0-9_]{1,30}\$</code> et être libre. Une fonction manquante lève sur place, avant tout message.

Stockage du plugin `storage:read`, `storage:write`

Bac à sable	Méthode hôte	Arguments et retour
<code>isofind.storage.get(clé)</code>	<code>storageGet</code>	La valeur, sous forme de texte, ou rien.
<code>isofind.storage.set(clé, valeur)</code>	<code>storageSet</code>	Écrit. La valeur est convertie en texte. Rejette quand le quota est atteint.
<code>isofind.storage.remove(clé)</code>	<code>storageRemove</code>	Retire une clé.
<code>isofind.storage.keys()</code>	<code>storageKeys</code>	Les clés du plugin, préfixe retiré.
<code>isofind.storage.getJSON(clé, défaut)</code>	<code>storageGet</code>	Confort. Rend le défaut si la clé est absente ou si le texte n'est pas du JSON lisible, plutôt que de lever.
<code>isofind.storage.setJSON(clé, valeur)</code>	<code>storageSet</code>	Confort, parce que la quasi-totalité des usages du stockage sont du JSON sérialisé à la main.

Calcul en R `compute:r`

Bac à sable	Méthode hôte	Arguments et retour
<code>WebRManager.charger(demande)</code>	<code>rBoot</code>	Amorce le moteur et rend un objet de moteur. C'est le seul chemin d'amorçage : les appels de <code>isofind.r</code> supposent un moteur déjà amorcé et échouent sinon.
<code>isofind.r.eval(code, options)</code>	<code>rEval</code>	Évalue. Rend la sortie texte ; passer <code>{ value: true }</code> pour recevoir aussi la valeur convertie, ce qui évite de sérialiser des objets R volumineux à chaque appel.
<code>isofind.r.text(code)</code>	<code>rEval</code>	Confort. La sortie seule, concaténée. C'est ce que veut une console.
<code>isofind.r.install(paquets, dépôt)</code>	<code>rInstall</code>	Installe des paquets dans l'environnement du plugin.
<code>isofind.r.reset()</code>	<code>rReset</code>	Remet l'environnement à zéro. La méthode hôte correspondante n'est pas servie dans l'état livré : l'appel échoue sur « méthode inconnue ».
par la compatibilité	<code>rEvalVoid</code> , <code>rEvalBoolean</code> , <code>rCapture</code>	Évaluation sans retour, évaluation booléenne, et capture de la sortie avec la valeur. Atteintes par l'objet de moteur que <code>charger</code> rend.

Réseau `network`

Pas de méthode. La capacité n'ouvre rien du côté de l'hôte : elle desserre la politique de sécurité du contenu du bac à sable, qui laisse alors passer les connexions en `https`. Le plugin appelle lui-même, avec les outils du navigateur.

Une méthode absente de la table est refusée. Il n'existe pas de liste de méthodes interdites, seulement cette liste de méthodes servies. C'est ce qui rend le refus la règle et le service l'exception, plutôt que l'inverse.

L'API HTTP de l'application est documentée à part, et essayable. La plupart des méthodes ci-dessus finissent par appeler une route du serveur local. Ces routes sont décrites et exécutables depuis l'interface Swagger que le backend FastAPI expose sous `/docs`, avec sa variante ReDoc sous `/redoc`, à l'adresse où tourne l'application.

Un plugin ne peut pas les appeler : la politique de sécurité du contenu ferme la sortie vers l'origine locale, capacité ou non. Mais les lire reste le moyen le plus rapide de connaître la forme exacte d'un échantillon, d'une mesure isotopique ou d'un résultat d'analyse avant d'écrire le code qui les consomme, plutôt que de la deviner à partir d'un jeu de données particulier.

La couche de compatibilité

Les plugins écrits pour la voie hôte s'adressent à `window` : `localStorage`, `IsoFindPluginAPI`, `PluginContributions`, `WebRManager`. Rien de tout cela n'existe dans une iframe à origine opaque, et `localStorage` y lève même une exception à la simple lecture.

Le bac à sable recrée donc ces objets. Ce n'est pas une commodité : sans eux, faire passer un plugin de la voie hôte au confinement exigerait de le réécrire, et un confinement qui impose de tout réécrire est un confinement qu'on n'adopte pas.

Objet recréé	Ce qu'il fait, et ce qu'il ne peut pas faire
<code>localStorage</code>	Une façade synchrone adossée à l'instantané livré à l'initialisation. Les lectures viennent du cache, les écritures sont immédiates en mémoire et poussées vers l'hôte sans attendre la confirmation, un échec étant signalé en console plutôt que remonté : aucun appelant de <code>setItem</code> n'est écrit pour le recevoir. La seule limite est qu'un plugin ne verrait pas une écriture faite par une autre instance de lui-même, et il n'y en a qu'une. L'accesseur natif est ombré par une propriété propre, le redéfinir en place étant refusé.
<code>IsoFindPluginAPI</code>	<code>getSamples()</code> rend une copie de l'instantané, de façon synchrone, parce que c'est ainsi que les plugins l'appellent. La liste peut donc être périmée si un échantillon est ajouté pendant que le plugin tourne ; <code>getSamplesAsync()</code> interroge l'hôte pour qui tient à la fraîcheur. Le reste délègue au pont.
<code>PluginContributions</code>	<code>addMenu</code> et <code>addPanel</code> annoncent la contribution comme le feraient les méthodes de <code>isofind</code> . <code>teardown</code> ne fait rien : l'hôte défait par source.
<code>WebRManager</code>	<code>charger()</code> amorce le moteur côté hôte et rend un objet qui présente la même surface que WebR en local : <code>evalR</code> , <code>evalRVoid</code> , <code>evalRBoolean</code> , <code>captureR</code> , <code>installPackages</code> . Les objets rendus gardent <code>toJs</code> , <code>free</code> et <code>destroy</code> pour que le code appelant continue de fonctionner sans savoir que la conversion a déjà eu lieu dans le travailleur et qu'il n'y a plus rien à libérer.

Un point que la couche ne peut pas couvrir : `new Worker`, que WebR appelle et qu'une origine opaque interdit. C'est une limite du navigateur, pas une méthode manquante, et c'est la raison pour laquelle le moteur

R vit dans un travailleur de l'hôte plutôt que dans le bac à sable. Il pèse aussi plusieurs dizaines de mégaoctets, qu'on ne rechargera pas une fois par plugin.

18. Le manifeste, champ par champ

Champ	Rôle	Défaut
<code>id</code>	Identifiant stable. Sert de clef d'installation, d'espace de noms de stockage, de préfixe d'URL de ressources, de source des contributions et de clef de jointure dans le journal. Le changer revient à publier un autre plugin.	requis
<code>name</code>	Nom affiché, en anglais : catalogue, modération et écran de consentement.	requis
<code>version</code>	Comparée composante par composante pour les mises à jour.	requis
<code>entry</code>	Point d'entrée. Son extension décide du langage : <code>.py</code> pour la voie Python, tout le reste pour la voie JavaScript.	<code>plugin.js</code>
<code>capabilities</code>	Les capacités demandées. Ce qui n'appartient pas au vocabulaire connu est écarté à l'installation.	<code>[]</code>
<code>contributes.menus</code>	Les entrées de menu déclarées, chacune avec son identifiant et son libellé.	vide
<code>contributes.panels</code>	Les panneaux déclarés. Chacun porte un identifiant, un libellé, éventuellement <code>dock</code> et <code>icon</code> .	vide
<code>assets</code>	Le dossier des ressources à extraire dans le Cache Storage. Le préfixe déclaré fait foi, y compris pour la signature.	aucun
<code>dependencies</code>	Les identifiants d'autres plugins requis, résolus avant l'installation.	vide
<code>runtime</code>	<code>isofind-native</code> demande l'exécution côté hôte, hors bac à sable. Toute autre valeur, ou son absence, laisse la voie normale.	absent
<code>category</code>	Sert au filtrage du catalogue : <code>analysis</code> , <code>visualization</code> , <code>data</code> , <code>tools</code> , <code>science</code> .	<code>tools</code>
<code>icon</code>	Fichier d'icône dans le paquet.	<code>icon.png</code>
<code>author</code>	Nom affiché sur la fiche. À ne pas confondre avec l'identité du certificat, qui est celle qui fait foi et que les modérateurs voient.	inconnu
<code>description</code>	Texte du catalogue, en anglais.	vide
<code>email</code>	Contact affiché dans les réglages du plugin.	aucun
<code>pythonPackages</code>	Paquets à charger avant l'exécution, voie Python seulement.	<code>numpy</code> , <code>pandas</code>
<code>keywords</code> , <code>repository</code> , <code>license</code> , <code>website</code>	Métadonnées affichées, sans effet sur le chargement.	aucun

signature	Posé par la signature. Porte la carte d'empreintes des ressources, à titre informatif : l'hôte la recalcule.	absent
---------------------------	--	--------

Champs hérités, encore lus

Champ	Équivalent actuel
main	entry
permissions	capabilities
language	Déduit de l'extension d' entry . Consulté seulement quand l'extension ne tranche pas.

19. Publier un plugin

Quatre gestes, dans cet ordre, et chacun refuse ce que le précédent n'a pas fait.

De l'atelier au catalogue



Une archive non signée est refusée par le relais, et un plugin non signé venu du catalogue serait refusé au chargement chez chaque utilisateur. Le circuit le dit avant, pas après.

Empaqueter

L'atelier produit l'archive depuis le dossier de travail, nommée d'après l'identifiant et la version. Rien n'oblige à passer par lui : un zip contenant un manifeste valide à la racine fait l'affaire. L'atelier a l'avantage de refuser un manifeste invalide avant d'écrire le fichier.

Signer

La signature exige une identité d'auteur sur ce poste, décrite en section 13. L'archive part au service local avec la phrase de passe, revient signée, et n'est rendue que si elle se vérifie. Une archive qui échoue à la vérification n'est pas restituée du tout.

Soumettre

La soumission attache l'archive, une description et une note aux modérateurs. L'identité affichée aux modérateurs vient du certificat, pas de ce que l'auteur déclare dans le formulaire.

La preuve de travail exigée par le relais est calculée par le relais, et aucun secret ne transite par la fenêtre de l'application. Le contrôle de signature, lui, a lieu avant : le serveur refuse une archive non signée, mais il le fait après avoir fait calculer la preuve de travail. L'utilisateur aurait donc attendu, chauffé son processeur, et récolté un refus. L'application le lui dit avant, et lui montre la sortie plutôt que de le laisser devant un mur.

Le registre

Le catalogue distant est un document JSON portant le schéma `isofind-plugin-registry/v1` et une liste d'entrées. Il est chargé à l'ouverture de la bibliothèque, hors mode air-gap, et fusionné avec le catalogue de première partie embarqué. En cas d'identifiant commun, le local l'emporte, et toute indisponibilité laisse le local seul plutôt que de vider la bibliothèque.

Une entrée distante ne porte pas son code, mais une adresse de téléchargement. Son installation suit le même chemin que l'import d'un fichier : dépôt des ressources, contrôle de conflits, consentement de capacités, vérification de signature.

20. L'atelier de développement

Un panneau pour écrire un plugin depuis un dossier de travail : validation du manifeste, rechargement à chaud, inspection des contributions, empaquetage.

L'atelier n'existe qu'en zone locale. Ce n'est pas un confort d'affichage. Le service qui le sert lit et écrit des fichiers sur la machine ; l'exposer hors du mode développeur reviendrait à offrir une lecture de fichier arbitraire depuis la fenêtre de l'application.

Le rechargement à chaud

L'horodatage du dossier est relevé toutes les 1200 millisecondes. À la moindre modification, le plugin est rechargé, et il est déchargé d'abord. Sans ce déchargement préalable, les contributions de la version précédente survivraient à côté des nouvelles : un identifiant retiré du manifeste resterait dans la palette, et l'auteur croirait son manifeste ignoré.

Le plugin d'atelier porte une marque de provenance. Il ne vient ni du catalogue ni d'une archive signée, et cette origine doit rester lisible par les gardes : un plugin d'atelier ne doit jamais pouvoir se faire passer pour un plugin installé.

Fermer le panneau n'interrompt pas le sondage. Le plugin en cours de développement reste chargé, et masquer une surface ne doit pas arrêter le rechargement de ce qu'on est en train d'écrire.

L'inspecteur

Il confronte deux listes : ce que le manifeste déclare, et ce que les registres du noyau portent réellement pour cette source. C'est l'outil de mise au point le plus utile de l'atelier.

État	Ce qu'il signifie
implémentée	Déclarée au manifeste et enregistrée par le code. Le cas normal.
déclarée, en attente d'activation	Le noyau porte la déclaration, le code ne l'a pas encore remplacée. Le plugin ne l'a pas implémentée, ou n'est pas encore activé.
déclarée mais absente	Le manifeste l'annonce, le noyau n'en a pas trace. Souvent un travail en cours, parfois une faute de frappe dans un identifiant.

enregistrée sans déclaration

Le cas grave. Elle fonctionne ici, mais elle échappera au consentement de l'utilisateur, qui n'aura jamais vu qu'elle existait.

Le dernier cas est expliqué, pas seulement coloré. Une contribution qui apparaît sans avoir été annoncée est précisément ce que la déclaration au manifeste est censée rendre impossible.

Ce que l'atelier signale aussi

Un plugin qui déclare un démarrage automatique voit apparaître un avertissement : son code s'exécute au lancement, avant toute invocation. C'est permis, et un modérateur demandera pourquoi.

21. Quatre plugins types

Quatre formes qui reviennent, écrites ici dans la forme livrée plutôt que décrites. Les libellés de manifeste sont en anglais, l'interface du plugin choisit sa langue à l'exécution.

Le code d'un plugin reçoit `isofind` et `plugin` en portée : rien à importer, rien à chercher dans `window`. Tout ce qui franchit la frontière rend une promesse.

Un panneau, et rien d'autre

Le plus court des plugins utiles : une entrée de menu qui ouvre un panneau, et un panneau qui affiche ce qu'il a. Deux capacités suffisent, plus celle qui donne accès aux données.

```
// manifest.json
{
  "id": "compte-echantillons",
  "name": "Sample Counter",
  "version": "1.0.0",
  "entry": "plugin.js",
  "capabilities": ["samples:read", "ui:menu", "ui:panel"],
  "contributes": {
    "menus": [ { "id": "ouvrir", "label": "Open Sample Counter" } ],
    "panels": [ { "id": "principal", "label": "Sample Counter", "dock": "right" } ]
  }
}
```

```

// plugin.js
// La langue voyage avec l'instantané du stockage : la façade de
// compatibilité la rend sans aller-retour, dès le premier rendu.
var fr = (localStorage.getItem('isofind_language') || 'fr') !== 'en';

isofind.registerPanel('principal', 'Sample Counter', function (ctx) {
  // Le rendu reçoit { root, id } : c'est ctx.root qu'il faut peupler.
  var racine = ctx.root;

  var p = document.createElement('p');
  p.style.color = 'var(--text-primary)';
  p.textContent = fr ? 'Comptage...' : 'Counting...';
  racine.appendChild(p);

  // L'appel traverse la frontière, donc il rend une promesse. La
  // version synchrone de l'instantané reste disponible par
  // IsoFindPluginAPI.getSamples() pour un code déjà écrit ainsi.
  isofind.getSamples().then(function (echantillons) {
    var n = echantillons.length;
    p.textContent = fr ? n + ' échantillon(s) chargé(s)'
      : n + ' sample(s) loaded';
  }).catch(function (e) {
    p.textContent = e.message; // « Capacité refusée:samples:read »
  });
});

isofind.registerMenu('ouvrir', 'Open Sample Counter', function () {
  isofind.openPanel('principal');
});

```

La racine est vidée avant chaque rendu. Le panneau se reconstruit donc entier à chaque ouverture, et l'état qu'il veut conserver doit vivre ailleurs, dans une variable du plugin ou dans son stockage. Fermer un panneau ne détruit rien : le contenu reste en place, il sera simplement reconstruit au prochain affichage.

Un plugin qui retient ses réglages

Le stockage est propre au plugin et survit à sa désactivation. Il ne survit pas à sa désinstallation.

```

{
  "capabilities": ["ui:menu", "ui:notify", "storage:read", "storage:write"]
}

```

```

var DEFALTS = { seuil: 3, unite: 'permil' };

function lireReglages() {
    // getJSON rend le défaut si la clef est absente ou si le texte
    // n'est pas du JSON lisible, plutôt que de lever au milieu d'un rendu.
    return isofind.storage.getJSON('reglages', DEFALTS);
}

function poserReglages(r) {
    return isofind.storage.setJSON('reglages', r)
        .then(function () {
            isofind.notify('Réglages enregistrés', 'success');
        })
        .catch(function (e) {
            // Le quota du plugin est atteint, ou storage:write n'a pas été
            // accordée. Le dire, plutôt que de laisser croire que le réglage
            // a été pris.
            isofind.notify(e.message, 'error');
        });
}

```

Une formule contribué

La fonction reste dans le bac à sable ; seule sa fiche part vers l'hôte, qui se charge de l'enveloppe synchrone, du sélecteur de fonctions et de l'aide.

```

isofind.registerFormula('EXCES_D', function (d180, d2H) {
    return d2H - 8 * d180;
}, {
    group: 'Isotopes stables', // groupe dans le sélecteur
    label: 'Excès en deutérium', // libellé de la fiche d'aide
    syntax: 'EXCES_D(d180, d2H)',
    example: 'EXCES_D(-8, -54)'
});

```

Le nom est un identifiant de feuille de calcul : majuscule initiale, puis majuscules, chiffres et soulignés, trente et un caractères au plus. Un nom mal formé ou déjà pris est refusé avec un avertissement en console, et la formule n'est pas posée. Le contrôle vaut pour tous les globaux de la page, et non seulement pour les fonctions.

Le premier calcul d'une cellule rend une valeur vide, la réponse arrive du bac à sable, et la feuille se recalcule. Une exception levée dans la fonction rend une valeur vide elle aussi, sans faire tomber le plugin.

Un plugin à moteur embarqué

Le moteur voyage dans le paquet, il est déposé dans le cache à l'installation, et il est chargé depuis l'URL que l'hôte fournit. Rien n'est téléchargé, ni au premier lancement, ni aux suivants.

```
{
  "id": "console-r",
  "name": "R Console",
  "version": "0.11.0",
  "entry": "plugin.js",
  "assets": "assets/",
  "capabilities": ["samples:read", "ui:menu", "ui:notify",
                  "compute:r", "storage:read", "storage:write"],
  "contributes": {
    "panels": [ { "id": "console", "label": "R Console", "dock": "bottom" } ]
  }
}
```

```
// L'URL de base a été posée sur le contexte au chargement. Elle est
// déterministe, donc valable aux lancements suivants sans réextraction.
//
// charger() est le seul chemin d'amorçage : le moteur vit dans un
// travailleur de l'hôte, parce qu'une origine opaque ne peut pas créer
// de Web Worker et qu'on ne rechargera pas WebR une fois par plugin.
WebRManager.charger({ baseUrl: plugin.assetsBaseUrl + 'webr/' })
  .then(function (moteur) {
    return moteur.evalR('R.version.string');
  })
  .then(function (objet) {
    return objet.toJs();
  })
  .then(function (version) {
    isofind.notify(version, 'info');
  })
  .catch(function (e) {
    // Ressource absente du cache : le service worker répond en toutes
    // lettres plutôt qu'un 404 muet, et le message est lisible ici.
    isofind.notify(e.message, 'error');
  });

// Une fois le moteur amorcé, la console passe par le raccourci de sortie.
function executer(code) {
  return isofind.r.text(code);
}
```

L'URL de base n'est honorée que si la signature couvre les ressources. Un moteur chargé sur la foi d'un chemin déclaré dans un manifeste non vérifié n'est pas un moteur souverain, c'est un exécutable choisi par l'auteur du paquet, et il ne s'exécutera pas dans un travailleur de l'application.

22. Dépannage

Les pannes courantes, et par où commencer. La plupart se voient en une minute dans la console, qui est volontairement bavarde sur les refus : un refus silencieux se cherche dans le mauvais fichier pendant une demi-heure.

Symptôme	Cause probable	Où regarder
----------	----------------	-------------

Le plugin ne se charge pas et la console parle de signature	Signature exigée et non valide : archive non signée, certificat expiré, ou contenu modifié depuis la signature	Le badge de la fiche, qui porte la raison. Pour un plugin local en cours d'écriture, le mode développeur lève l'exigence.
Le refus de signature ne dit pas ce qui cloche	Un écart entre la carte signée et les ressources trouvées en cache	La console nomme les fichiers absents, en trop ou différents. Réinstaller redépose les ressources.
Le panneau s'ouvre en modale au lieu de rejoindre un dock	Le registre du shell n'était pas disponible, ou la page n'a ni dock ni zone centrale	La console au chargement du plugin. Une page hors du manifeste de démarrage ne porte pas le shell.
Un panneau devient vide, et un autre s'est fermé tout seul	Deux panneaux du même plugin : il n'a qu'une iframe, et le second recible la première	Attendu. Un plugin n'affiche qu'une surface à la fois.
Le moteur embarqué échoue sur une erreur de compilation	Ses ressources ne sont pas dans le cache : jamais déposées, ou effacées par un nettoyage du navigateur	Le corps de la réponse 404, qui le dit en toutes lettres. Réinstaller le plugin les redépose.
Une formule rend une valeur vide et ne change jamais	Nom refusé par le contrôle de forme, ou nom déjà pris par un autre global	La console, qui nomme la formule écartée. Le nom doit répondre à <code>^[A-Z][A-Z0-9_]{1,30}\$</code> .
Une formule rend une valeur vide au premier calcul, puis la bonne	Attendu. L'enveloppe synchrone rend une valeur vide en attendant la réponse du bac à sable, puis la feuille se recalcule	Rien à corriger.
Un appel échoue sur « capacité refusée »	La case a été décochée à l'installation, ou la capacité n'appartient pas au vocabulaire connu et a été écartée	Les réglages du plugin dans le gestionnaire, et l'avertissement de console émis à l'installation.
Le plugin ne voit aucun échantillon	<code>samples : read</code> non accordée, ou accès révoqué au moment du chargement	La console signale un instantané retenu. Une révocation ne mord qu'au chargement suivant.
Les couleurs du plugin ne suivent pas le thème	Un jeton employé qui n'appartient pas à la liste exposée	La liste de la section 9. Un jeton hors liste n'arrive jamais dans le document du bac à sable.
Une entrée a disparu de la barre latérale	Elle a été masquée par sa croix, ce qui ne désactive pas le plugin	Les réglages du plugin. L'action reste lançable depuis le lanceur.
Rien ne change après une installation	Le rechargement se décide à la fermeture de toutes les fenêtres de plugins, pour permettre d'enchaîner les installations	Fermer le gestionnaire. Une installation qui a déposé des ressources demande un rechargement du document.
L'atelier est introuvable	Il n'existe qu'en zone locale	Attendu. Le service qui le sert lit des fichiers sur la machine.

Le partage refuse l'archive avant même de l'envoyer	Elle n'est pas signée	Menu Plugins, identité d'auteur et signature. Le contrôle a lieu avant la preuve de travail, pour ne pas la faire calculer pour rien.
Un appel R échoue sur « moteur R non amorcé »	Le moteur n'a pas été amorcé : isofind.r suppose un moteur déjà en place	Passer par WebRManager.charger() avant tout appel. C'est le seul chemin d'amorçage.
Le panneau perd son état quand on change d'onglet	La racine est vidée avant chaque rendu, et le panneau se reconstruit entier	Attendu. L'état doit vivre dans une variable du plugin ou dans son stockage, pas dans le DOM du panneau.
Un plugin Python reste sans effet après son chargement	Ses rappels ont été libérés, ou son espace de noms n'a pas survécu	La console au déchargement. Un plugin rechargé perd les PyProxy de sa version précédente, ce qui est voulu.

Le réflexe utile. Regarder d'abord ce que la console dit au moment du chargement, et non au moment de la panne. Presque tous les refus du socle sont prononcés là : signature, capacité écartée, formule refusée, zone de dock indisponible, ressources absentes du cache. Quand la panne se manifeste, la cause a souvent déjà été nommée plusieurs secondes plus tôt.

23. Bornes et persistance

Les constantes du socle

Borne	Valeur	Raison
Quota de stockage par plugin	2 Mo	Sans lui, un plugin remplit le stockage de l'origine et l'application entière cesse de pouvoir écrire.
Longueur d'un nom de formule	2 à 31 caractères	Majuscule initiale, puis majuscules, chiffres et soulignés.
Sondage du dossier d'atelier	1200 ms	Assez court pour ne pas attendre, assez long pour ne pas relire un dossier en permanence.
Regroupement du recalcul de la feuille	50 ms	Plusieurs formules se résolvent souvent ensemble ; un recalcul par réponse serait gaspillé.
Réessais d'injection dans la barre latérale	0, 150, 500, 1500 ms	La barre peut apparaître après le plugin, ou être reconstruite ensuite.
Diagnostic de barre latérale introuvable	1600 ms	Dit une fois, et clairement. Un bouton absent sans explication se cherche longtemps.
Version du protocole	2.0.0	Annoncée par le pont au chargement.

Les plans d'affichage

Surface	Plan	Pourquoi
Panneau docké ou au centre	1	Sous n'importe quelle modale native, pour qu'un fond assombri le recouvre normalement, comme le reste du shell.
Voile de la modale du plugin	11000	Le fond assombri de sa propre fenêtre.
Iframe en mode modal	11001	Juste au-dessus de ce voile. L'iframe n'étant plus un enfant de l'overlay mais un élément séparé sous <code>body</code> , un plan trop bas la ferait passer sous son propre fond.

Ce qui survit à la page

Clef	Contenu
<code>isofind_plugins</code>	Les fiches des plugins installés : identité, capacités accordées, code, manifeste d'origine, verdict de signature, état d'activation.
<code>isofind_plugin_settings</code>	Les réglages du socle.
<code>isofind_quickaccess</code>	L'ordre et le masquage des entrées d'accès rapide.
<code>isofind_plugin_store:<id>:<clef></code>	Le stockage d'un plugin. Purgé à la désinstallation, jamais au téléchargement.
<code>isofind_config</code>	La configuration de l'application, lue et écrite par <code>config:read</code> et <code>config:write</code> .
<code>isofind_language</code>	La langue, transmise aux plugins avec l'instantané du stockage.
<code>isofind-plugin-assets</code>	Cache Storage, et non stockage local : les ressources des plugins, qui pèsent parfois plusieurs dizaines de mégaoctets et doivent persister entre les lancements.

Les réglages du socle

Réglage	Défaut	Effet
<code>verifySignatures</code>	vrai	Seul un refus explicite lève l'exigence de signature. Un réglage absent ou corrompu vaut exigence.
<code>sandboxMode</code>	vrai	Le confinement par défaut.
<code>devMode</code>	faux	Lève l'exigence de signature pour les seuls plugins locaux, et ouvre l'atelier.
<code>airgapMode</code>	faux	Ferme le catalogue, le partage, le CDN et la capacité réseau.
<code>autoUpdate</code>	faux	Mise à jour automatique depuis le catalogue.
<code>allowNetwork</code>	faux	Réglage global, distinct de la capacité accordée plugin par plugin.

<code>allowFilesystem</code>	faux	Réglage global.
<code>cacheLimit</code>	100	Plafond du cache, en mégaoctets.
<code>pyodideLocalPath</code>	vide	Chemin explicite du paquet Pyodide. Prime sur la sonde du paquet embarqué, qui prime elle-même sur le CDN.

Un stockage illisible ne lève plus : on retombe sur ces valeurs par défaut, qui sont les plus strictes. Rendre une exception à tous les appelants, dont la politique de signature et la détection d'air-gap, faisait dépendre la sécurité de la bonne santé d'une chaîne de caractères.

Les évènements

Évènement	Quand
<code>plugins.changed</code>	La liste des plugins installés change, avec le total et le nombre d'actifs. Émis aussi au démarrage, quand la liste passe de vide à pleine.
<code>registry.changed</code>	Une commande ou un panneau entre au registre ou en sort. Le lanceur et le bouton de la barre de titre s'y abonnent : sans ce relais, il faudrait rouvrir le panneau pour voir apparaître ce qu'on vient d'installer, et l'installation paraîtrait avoir échoué.
<code>themechange</code>	Émis par le gestionnaire de thème, écouté par le pont, qui rediffuse les jetons à tous les bacs à sable ouverts. Les plugins suivent donc sans être rechargés.

Les surfaces exposées à la page

Objet	Rôle
<code>PluginSandboxHost</code>	Le pont : charger, décharger, purger le stockage, installer le vérificateur, lire un verdict.
<code>PluginState</code>	La source unique de vérité sur les plugins installés. Exposée parce que l'inspecteur, le menu du shell et le panneau Journal la consultent.
<code>PluginQuickAccess</code>	Le propriétaire unique de la section Plugins de la barre latérale, quelle que soit l'origine des entrées.
<code>PluginContributions</code>	Les contributions matérialisées côté hôte, pour les plugins qui s'exécutent dans la page.
<code>PluginAssets</code>	Déposer, purger, vérifier la présence et calculer les empreintes des ressources.
<code>IsoFindPluginAPI</code>	L'accès aux données de l'application, derrière lequel les deux tables de capacités délèguent.

Les surfaces recrées dans le bac à sable

Objet	Rôle
<code>isofind</code>	La surface du SDK, version 2.0.0 : données, réglages, notifications, stockage, R, contributions, journal.

plugin	Le contexte de l'instance : identifiant, nom, URL de base des ressources, capacités accordées, raccourci de notification.
localStorage	Façade synchrone adossée à l'instantané, qui ombre l'accesseur natif, lequel lève sur une origine opaque.
IsoFindPluginAPI	Même nom que côté hôte, même surface, pour qu'un plugin venu de la voie hôte n'ait rien à réécrire.
PluginContributions	Annonce les contributions comme les méthodes de isofind . Son teardown ne fait rien, l'hôte défaisant par source.
WebRManager	Amorce le moteur R dans un travailleur de l'hôte et en présente la surface locale.

Écrire

De zéro à un plugin signé, dans l'ordre où les gestes se font, avec ce qu'il faut savoir à chaque étape.

24. De zéro à un plugin qui tourne

Six gestes, et le premier plugin utile tient en deux fichiers. Cette section les prend dans l'ordre ; les suivantes reviennent sur ce qui mérite d'être expliqué.

Geste	Ce qu'on fait	Ce qui l'exige
1	Créer un dossier avec un <code>manifest.json</code> et un <code>plugin.js</code> .	Rien.
2	Ouvrir ce dossier dans l'atelier, qui valide le manifeste et charge le code.	Le mode développeur, et la zone locale.
3	Écrire, enregistrer, regarder. Le plugin se recharge seul.	Rien.
4	Lire l'inspecteur, jusqu'à ce que toutes les contributions soient marquées implémentées.	Rien.
5	Empaqueter, ce qui produit une archive <code>.isplugin</code> .	Un manifeste valide.
6	Signer, puis installer ou soumettre.	Une identité d'auteur sur ce poste.

Le plus petit plugin qui charge

Un manifeste a besoin de trois champs pour être accepté, et d'un quatrième pour être utile. Sans capacité déclarée, l'installation ne demande rien et le plugin ne peut rien faire d'autre que calculer dans son coin.

```
// manifest.json
{
  "id": "mon-premier-plugin",
  "name": "My First Plugin",
  "version": "0.1.0",
  "entry": "plugin.js",
  "capabilities": ["ui:notify"]
}
```

```
// plugin.js
// Rien à importer. isofind et plugin sont déjà en portée, et le code
// est exécuté en mode strict par l'hôte.
isofind.log('Chargé :', plugin.id, plugin.capabilities);
isofind.notify('Bonjour depuis ' + plugin.name, 'info');
```

Si rien ne se passe, la console de l'application le dira. Elle est volontairement bavarde sur les refus, et presque toutes les pannes du socle y sont nommées au moment du chargement, pas au moment où l'on s'en aperçoit.

Travailler dans l'atelier

L'atelier lit le dossier, relève son horodatage toutes les 1200 millisecondes, et recharge le plugin dès qu'un fichier bouge. Il décharge d'abord, ce qui évite qu'une contribution retirée du manifeste survive à côté des nouvelles et laisse croire que le manifeste est ignoré.

Trois choses s'y lisent en permanence. La validité du manifeste, avec la liste des erreurs quand il y en a. L'inspecteur, qui confronte ce que le manifeste déclare à ce que les registres portent réellement. Et un avertissement si le plugin demande à s'exécuter au démarrage, ce qui est permis mais qu'un modérateur voudra justifier.

Sans atelier, le cycle est beaucoup plus long. On peut très bien zipper le dossier et l'installer par fichier, mais chaque correction demande alors de rezipper, de désinstaller, de réinstaller et de recharger l'application. L'atelier n'est pas un confort d'auteur, c'est ce qui rend le développement praticable.

Ce qu'on obtient à la fin

L'empaquetage produit une archive nommée d'après l'identifiant et la version. Elle n'est pas signée : un plugin non signé est refusé au chargement dès que la vérification est exigée, et refusé par le relais de soumission dans tous les cas. La signature est donc le geste suivant, et non une formalité que l'on remet à plus tard.

25. Choisir ses capacités

La règle tient en une phrase : demander ce qu'on emploie, rien de plus. Chaque case supplémentaire est une raison de plus, pour l'utilisateur qui la lit, de ne pas installer. Un plugin qui demande la lecture de la base pour afficher un convertisseur d'unités sera décoché, et il échouera ensuite sans que personne comprenne pourquoi.

Ce que le plugin veut faire	Ce qu'il doit demander
Afficher une fenêtre à lui	<code>ui:panel</code>
Être atteignable depuis la barre latérale	<code>ui:menu</code>
Dire quelque chose à l'utilisateur	<code>ui:notify</code>
Lire les échantillons chargés, ou leurs mesures	<code>samples:read</code>
Créer un échantillon	<code>samples:write</code>
Retenir un réglage d'une session à l'autre	<code>storage:read</code> , <code>storage:write</code>
Connaitre la langue de l'application	<code>storage:read</code>
Ajouter des fonctions au tableur	<code>ui:formula</code>
Calculer en R, avec ou sans moteur embarqué	<code>compute:r</code>
Lire ou écrire les réglages de l'application	<code>config:read</code> , <code>config:write</code>

Déclencher le moteur d'analyse	<code>analysis:run</code>
--------------------------------	---------------------------

Appeler un service extérieur en HTTPS	<code>network</code>
---------------------------------------	----------------------

Ce qui n'exige rien

Tout ce qui reste dans le bac à sable. Dessiner une interface, lire les jetons de thème, calculer, écrire dans la console par `isofind.log`, connaître son propre identifiant et l'URL de base de ses ressources. Un plugin de calcul pur qui reçoit ses données de l'utilisateur, par un champ de saisie ou un collage, n'a besoin d'aucune capacité de données.

Cinq pièges

La zone de dock ne remplace pas `ui:panel`. Un panneau déclaré avec une zone est ouvert par la coque de l'application, qui ne passe jamais par l'appel du plugin. La capacité est donc vérifiée dès la déclaration. Le manifeste déclare la zone, il ne peut pas déclarer le droit, puisque c'est l'auteur du plugin qui l'écrit.

La langue coute `storage:read`. Elle voyage avec l'instantané du stockage, et sans cette capacité le plugin ne la reçoit pas. Il retombera sur une langue par défaut, ce qui est acceptable, mais c'est un choix à faire consciemment.

`compute:r` ne donne aucune donnée. Le moteur R tourne dans un travailleur sans DOM ni jeton. Un plugin qui veut analyser les échantillons en R a besoin des deux capacités, et il devra passer les données au moteur lui-même.

`network` ne sert à rien en mode air-gap. La politique de sécurité du contenu retombe à zéro sortie quelle que soit la case cochée, et la case apparaît grisée à l'installation. Un plugin destiné à des postes isolés ne devrait pas en dépendre pour fonctionner.

Une capacité inventée est ignorée. Ce qui n'appartient pas au vocabulaire de la section 12 est écarté à l'installation, avec un avertissement en console. Le plugin s'installera, et échouera ensuite sur une capacité que l'utilisateur n'a jamais pu accorder.

26. Écrire l'interface d'un panneau

Le document du bac à sable est vierge. Aucune feuille de style de l'application n'y est chargée, et rien n'y ressemble à IsoFind tant que le plugin ne l'a pas écrit. Ce qui est fourni, c'est la palette : les jetons de thème listés en section 9, injectés dès la création et rafraichis à chaque bascule.

Poser sa feuille une fois

```
function styles() {
  if (document.getElementById('mp-style')) return; // posée une seule fois
  var st = document.createElement('style');
  st.id = 'mp-style';
  st.textContent = [
    // Le préfixe évite toute collision si le plugin gagne un second
    // panneau. Les couleurs viennent des jetons, jamais d'une valeur
    // écrite en dur : un thème sombre rendrait le panneau illisible.
    '.mp{font:13px/1.55 system-ui,-apple-system,"Segoe UI",sans-serif;',
    '  color:var(--text-primary);padding:14px 16px;}',
    '.mp h3{margin:0 0 8px;font-size:14px;color:var(--brand-primary);}',
    '.mp-ligne{display:flex;gap:10px;padding:6px 0;',
    '  border-bottom:1px solid var(--border-subtle);}',
    '.mp-ligne:hover{background:var(--surface-3);}',
    '.mp-muet{color:var(--text-secondary);font-size:12px;}',
    '.mp button{border:1px solid var(--border-primary);border-radius:var(--rayon-2);',
    '  background:transparent;color:var(--text-primary);padding:6px 12px;',
    '  font:inherit;cursor:pointer;}',
    '.mp button:hover{background:var(--brand-tint-50);color:var(--brand-primary);}'
  ].join('');
  document.head.appendChild(st);
}
```

Les jetons de géométrie de l'application, comme `--rayon-2`, ne font pas partie de la liste exposée. Écrire `var(--rayon-2)` reste sans danger, la valeur de repli du navigateur s'appliquant, mais mieux vaut prévoir un repli explicite quand la forme compte.

Six règles qui évitent les pannes les plus fréquentes

La racine est vidée avant chaque rendu. Le panneau se reconstruit entier à chaque ouverture, et tout ce qu'il veut conserver doit vivre ailleurs : dans une variable du plugin, qui survit tant que l'instance vit, ou dans son stockage, qui survit au redémarrage.

Un seul panneau visible à la fois. Le plugin n'a qu'une iframe. Ouvrir un second panneau ferme le premier, explicitement. Deux surfaces simultanées demandent deux plugins.

Pas de requête vers le serveur local. La politique de sécurité du contenu la bloque, capacité ou non. Les données passent par `samples:read`, ou elles sont saisies par l'utilisateur, ou elles voyagent dans le paquet.

Pas de `localStorage` pour du volumineux. La façade de compatibilité le rend utilisable, mais elle écrit dans le stockage cloisonné du plugin, plafonné à deux mégaoctets. Une base de référence se met dans le dossier de ressources, pas dans le stockage.

Les évènements ne traversent pas la frontière. Un menu contextuel dessiné dans le bac à sable ne rencontrera jamais celui de l'application : ce sont deux documents séparés. C'est une bonne nouvelle, et cela dispense de toute précaution de ce côté.

Écrire les deux langues. L'interface d'un plugin se lit en français comme en anglais. La langue arrive dans l'instantané du stockage, et la lire au chargement plutôt qu'à chaque rendu évite d'y penser ensuite.

27. Un plugin complet

Un plugin réel, court, et qui emploie l'essentiel de ce qui précède : un panneau, une entrée de menu, la lecture des échantillons, un réglage retenu d'une session à l'autre, et une notification. Rien qui ne figure pas dans les sections précédentes.

```
// manifest.json
{
  "id": "dispersion-isotopique",
  "name": "Isotopic Spread",
  "version": "1.0.0",
  "author": "Nom de l'auteur",
  "description": "Standard deviation of isotopic ratios, by element.",
  "category": "analysis",
  "entry": "plugin.js",
  "capabilities": ["samples:read", "ui:menu", "ui:panel",
                  "ui:notify", "storage:read", "storage:write"],
  "contributes": {
    "menus": [ { "id": "ouvrir", "label": "Open Isotopic Spread" } ],
    "panels": [ { "id": "principal", "label": "Isotopic Spread", "dock": "right" } ]
  }
}
```

```

// plugin.js

// La langue est lue une fois, au chargement : elle arrive avec
// l'instantané du stockage, donc sans aller-retour.
var fr = (localStorage.getItem('isofind_language') || 'fr') !== 'en';
function t(f, e) { return fr ? f : e; }

// Le réglage vit dans le stockage du plugin. Il survit à une
// désactivation, pas à une désinstallation.
var MINIMUM_DEFAUT = 3;
var minimum = MINIMUM_DEFAUT;

function styles() {
  if (document.getElementById('di-style')) return;
  var st = document.createElement('style');
  st.id = 'di-style';
  st.textContent = [
    '.di{font:13px/1.55 system-ui,-apple-system,"Segoe UI",sans-serif;',
    '  color:var(--text-primary);padding:14px 16px;}',
    '.di h3{margin:0 0 4px;font-size:14px;color:var(--brand-primary);}',
    '.di p{margin:0 0 12px;color:var(--text-secondary);font-size:12px;}',
    '.di-l{display:flex;gap:10px;padding:6px 0;',
    '  border-bottom:1px solid var(--border-subtle);}',
    '.di-el{flex:1;font-weight:600;}',
    '.di-v{font-variant-numeric:tabular-nums;}',
    '.di-n{color:var(--text-muted);font-size:12px;width:52px;text-align:right;}'
  ].join('');
  document.head.appendChild(st);
}

// Le calcul ne dépend d'aucune capacité : il travaille sur ce que
// l'hôte a déjà livré.
function dispersion(echantillons) {
  var par = {};
  echantillons.forEach(function (e) {
    (e.isotope_data || e.isotopic_data || []).forEach(function (m) {
      var v = parseFloat(m.ratio_value);
      if (!isFinite(v) || !m.element) return;
      (par[m.element] = par[m.element] || []).push(v);
    });
  });

  return Object.keys(par).map(function (el) {
    var xs = par[el];
    var moyenne = xs.reduce(function (a, b) { return a + b; }, 0) / xs.length;
    var variance = xs.reduce(function (a, b) {
      return a + (b - moyenne) * (b - moyenne);
    }, 0) / xs.length;
    return { element: el, n: xs.length, ecart: Math.sqrt(variance) };
  }).filter(function (r) {
    // Un écart-type sur deux mesures ne veut rien dire. Le seuil est
    // un réglage, parce qu'il dépend de ce que l'utilisateur analyse.
    return r.n >= minimum;
  }).sort(function (a, b) { return b.ecart - a.ecart; });
}

function peindre(racine, lignes) {
  // La racine arrive vide : l'hôte la nettoie avant chaque rendu.
  var d = document.createElement('div');

```

```

d.className = 'di';

var h = document.createElement('h3');
h.textContent = t('Dispersion par élément', 'Spread by element');
d.appendChild(h);

var p = document.createElement('p');
p.textContent = t('Écart-type des rapports mesurés, à partir de '
    + minimum + ' mesures.',
    'Standard deviation of measured ratios, from '
    + minimum + ' measurements. ');
d.appendChild(p);

if (!lignes.length) {
    var vide = document.createElement('p');
    vide.textContent = t('Rien à montrer avec ce seuil.',
        'Nothing to show at this threshold. ');
    d.appendChild(vide);
    racine.appendChild(d);
    return;
}

lignes.forEach(function (r) {
    var l = document.createElement('div');
    l.className = 'di-l';

    var el = document.createElement('span');
    el.className = 'di-el';
    el.textContent = r.element;

    var v = document.createElement('span');
    v.className = 'di-v';
    v.textContent = r.ecart.toFixed(4);

    var n = document.createElement('span');
    n.className = 'di-n';
    n.textContent = r.n + ' ' + t('mes.', 'meas. ');

    l.appendChild(el);
    l.appendChild(v);
    l.appendChild(n);
    d.appendChild(l);
});

racine.appendChild(d);
}

isofind.registerPanel('principal', 'Isotopic Spread', function (ctx) {
    // Le rendu recoit { root, id } : c'est ctx.root qu'il faut peupler.
    var racine = ctx.root;

    styles();

    var attente = document.createElement('div');
    attente.className = 'di';
    attente.textContent = t('Lecture...', 'Reading...');
    racine.appendChild(attente);

    // Deux appels qui franchissent la frontière, donc deux promesses.

```

```

// Le réglage d'abord, puisque le calcul en dépend.
isofind.storage.getJSON('reglages', { minimum: MINIMUM_DEFAULT })
    .then(function (r) {
        minimum = r.minimum || MINIMUM_DEFAULT;
        return isofind.getSamples();
    })
    .then(function (echantillons) {
        racine.textContent = '';
        peindre(racine, dispersion(echantillons));
    })
    .catch(function (e) {
        // Le message porte la capacité manquante quand c'en est une :
        // « Capacité refusée: samples:read ». Le montrer plutôt que de
        // laisser un panneau vide sans explication.
        racine.textContent = '';
        var err = document.createElement('div');
        err.className = 'di';
        err.style.color = 'var(--tone-rouge-text)';
        err.textContent = e.message;
        racine.appendChild(err);
    });
});

isofind.registerMenu('ouvrir', 'Open Isotopic Spread', function () {
    isofind.openPanel('principal');
});

// Le réglage se pose ailleurs dans une vraie version, par un champ du
// panneau. On garde ici la fonction qui écrit, pour montrer le traitement
// d'un quota atteint : une écriture refusée doit se dire, sinon
// l'utilisateur croit son réglage enregistré.
function poserMinimum(v) {
    return isofind.storage.setJSON('reglages', { minimum: v })
        .then(function () {
            minimum = v;
            isofind.notify(t('Seuil enregistré', 'Threshold saved'), 'success');
        })
        .catch(function (e) {
            isofind.notify(e.message, 'error');
        });
}

```

28. Liste de contrôle avant publication

Ce qu'un modérateur regardera, et ce qui coûte le plus cher à corriger après coup.

Point	Ce qu'on vérifie
Identifiant	Stable et définitif. Il sert de clef d'installation, d'espace de stockage, de préfixe de ressources et de source des contributions. En changer revient à publier un autre plugin, et à perdre le stockage des utilisateurs.
Capacités	Aucune qui ne soit employée. Chaque case en trop est une raison de ne pas installer, et une capacité décochée par prudence fera échouer le plugin plus tard.

Contributions	Toutes déclarées au manifeste, et toutes marquées implémentées par l'inspecteur. Une contribution enregistrée sans déclaration échappe au consentement de l'utilisateur.
Métadonnées	Nom, description et libellés en anglais. Ce sont les textes du catalogue, du consentement et de la modération.
Interface	Bilingue, et bâtie sur les jetons de thème. Un plugin testé en thème clair seulement se découvre illisible chez l'utilisateur suivant.
Réseau	Aucune sortie qui ne soit annoncée. Un plugin qui charge une ressource depuis un CDN cesse de fonctionner sur un poste isolé, et c'est le contraire de ce que le logiciel promet.
Ressources	Dans le dossier déclaré par assets , et pas ailleurs. Le préfixe déclaré fait foi pour la signature comme pour le dépôt.
Démarrage	Ne pas demander l'exécution au lancement sans raison. C'est permis, l'atelier le signale, et un modérateur demandera pourquoi.
Erreurs	Toutes les promesses ont un traitement d'échec qui affiche le message réel. Un panneau vide sans explication se met sur le compte du logiciel.
Signature	L'archive soumise est celle qui a été signée et rendue par le service, pas la version non signée restée à côté.
Identité	Le certificat n'expire pas dans les prochaines semaines. Une signature produite avec un certificat expiré est refusée.

Le dernier réflexe. Installer sa propre archive signée sur une installation neuve, comme un utilisateur, plutôt que de s'en tenir à l'atelier. L'atelier lève l'exigence de signature, charge depuis un dossier et ne dépose aucune ressource dans le cache : trois différences qui suffisent à faire passer un plugin cassé pour un plugin qui marche.