

Le noyau

Étendre, contribuer, publier

Écrire un plugin, ajouter une fonctionnalité au cœur, et comprendre ce que le noyau accorde, refuse et confine.

Août 2026. Décrit le noyau de contributions version 1.0.0 et le manifeste v3.

Les quatre premières sections se lisent d'affilée. Le reste est un catalogue de points de contribution.

Sommaire

Le modèle

1. Trois façons d'étendre IsoFind	3
2. Le noyau de contributions	4
3. Le cycle de vie d'une page	6
4. Le manifeste v3	7

Les points de contribution

5. Commandes	10
6. Panneaux	11
7. Éditeurs et zone centrale	12
8. Menus	13
9. Recherche, état, tableau de bord	14
10. Tâches de fond, notifications, modales	15
11. Enregistrement, fichiers, projet	16
12. Connecteurs, LIMS, workflows	17
13. Réglages, contexte, évènements	19

Exécution et sécurité

14. Les trois runtimes	21
15. Zone, source, signature	22
16. Capacités et consentement	22
17. Ce que le noyau refuse	25

Écrire un plugin

18. De zéro à un plugin chargé	26
19. Signer et distribuer	28
20. Quatre plugins types	28
21. Dépannage	30
22. Référence de l'API	31

Le modèle

Ce qu'est une contribution, comment une page se construit, et comment un plugin déclare ce qu'il apporte avant même d'être exécuté.

1. Trois façons d'étendre IsoFind

Elles passent toutes par le même registre, et c'est ce qui rend la première utile pour apprendre les deux autres.

Voie	Ce que c'est	Ce qu'elle suppose
Script utilisateur	Du code exécuté dans la page, qui enregistre des contributions sous la source <code>user</code> . C'est aussi le seul endroit d'où l'on peut envelopper une commande existante.	La zone locale. Rien à emballer, rien à signer.
Plugin	Un manifeste plus du code, installé, consenti, éventuellement signé. La voie normale pour ce qui doit être distribué.	Un identifiant, des capacités déclarées, et un runtime confiné.
Contribution du cœur	Un fichier de plus dans <code>static/js/kernel/</code> , chargé avec les autres. Même API que les deux précédentes, source <code>core</code> .	Un accès au dépôt. Le cœur n'est pas soumis au consentement, puisqu'il est l'application.

Un point mérite d'être posé tout de suite : le cœur d'IsoFind s'écrit avec la même API qu'un plugin. Les panneaux, les commandes, les menus, les widgets du tableau de bord du produit passent tous par `K.contribute`. Ce n'est pas une politesse envers les auteurs de plugins, c'est ce qui garantit que l'API tient : une surface d'extension qui ne sert qu'aux extensions se dégrade silencieusement, parce que personne ne s'en sert assez pour s'apercevoir qu'elle manque quelque chose.

La règle qui tient tout le reste

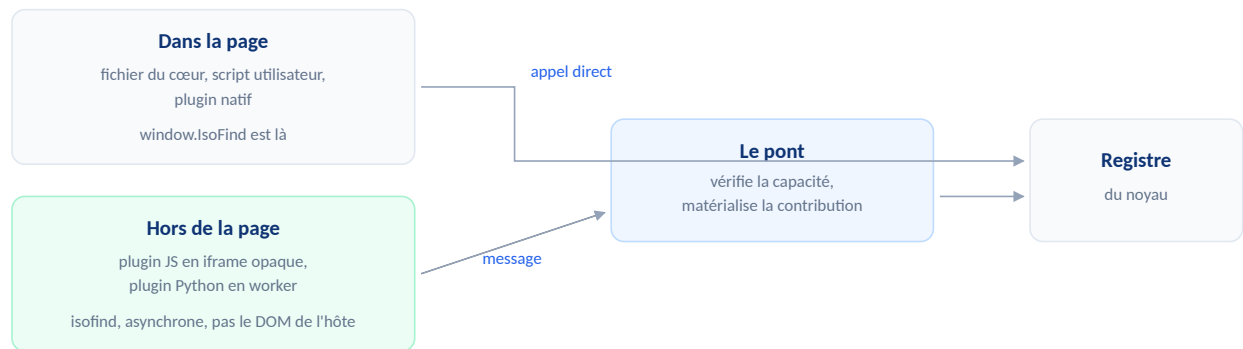
Dès qu'une fonction est câblée en dur dans une page, elle échappe au registre. Rien ne peut plus la lister, la surcharger, la lancer depuis une console, la lier à un raccourci, ni la faire apparaître dans la palette. La frontière entre le cœur et les plugins se reforme alors d'elle-même, et elle se reforme toujours du même côté.

C'est la seule règle qui, tenue partout, empêche cela. Elle explique la plupart des choix décrits dans ce guide.

Où tourne votre code, et ce que cela change

C'est la distinction la plus importante de ce guide, et celle qui rend les exemples lisibles. Tout ce qui s'exécute dans la page voit `window.IsoFind` et peut appeler les registres directement. Un plugin confiné ne le voit pas, et n'en a pas besoin : il parle à travers une façade que l'hôte arbitre.

Votre code tourne	K	Ce que vous appelez
Fichier du cœur , dans la page	oui	Tous les registres de ce guide, en direct et en synchrone.
Script utilisateur , dans la page	oui	Les mêmes, plus <code>K.commands.override</code> .
Plugin natif , dans la page	oui	Les mêmes, plus <code>window.IsoFindPluginAPI</code> . C'est ce qui le rend dangereux, et déprécié.
Plugin JS confiné , en iframe	non	L'objet <code>isofind</code> , posé par le pont du bac à sable. L'iframe a son PROPRE document : on peut y créer des nœuds.
Plugin Python , en worker	non	L'objet <code>isofind</code> également, mais un worker n'a aucun document. Les deux façades ne portent pas les mêmes verbes, voir la section 16.



Les deux chemins aboutissent au même registre. Le second passe par un arbitre, et c'est la seule différence qui compte.

Le manifeste traverse la frontière, votre code non

Les contributions déclarées dans le manifeste sont enregistrées par l'hôte, qui lit ce manifeste dans la page. Une commande de plugin confiné existe donc dans le registre du noyau, apparaît dans la palette et se lie à un raccourci, sans que le bac à sable ait jamais vu `K`.

Ce que votre code enregistre depuis le bac à sable passe, lui, par le pont : un message part, l'hôte vérifie la capacité, puis matérialise la contribution de son côté. Les sections 5 à 13 décrivent le registre tel que l'hôte le voit ; la section 16 donne les verbes de la façade et la capacité qu'exige chacun.

Les exemples de ce guide sont écrits côté page. Ils valent tels quels pour un fichier du cœur, un script utilisateur et un plugin natif. Depuis un bac à sable, `K` est indéfini : le code équivalent appelle la façade, dont la surface est plus étroite et dont les verbes diffèrent d'un runtime à l'autre. La section 16 donne les deux tables, la section 18 les deux formes côte à côte.

Ce que ce guide suppose

Du JavaScript, sans dépendance ni étape de compilation. Le noyau est écrit en ES5, les fichiers du cœur aussi, et les plugins JS tournent dans une iframe où vous êtes libre de votre style. Aucun paquet npm n'est requis nulle part.

Le vocabulaire du domaine, échantillon, analyse, campagne, workflow, n'est pas expliqué ici. Pour les workflows et leurs tâches, le guide dédié va plus loin que la section 12 de celui-ci.

2. Le noyau de contributions

Quatre notions suffisent : la contribution, la source, le jeton, le registre. Tout le reste en découle.

Notion	Ce que c'est
Contribution	Un objet déclaré au registre sous un type et un identifiant : une commande, un panneau, un éditeur, un indicateur de barre d'état. Le type n'est pas une liste fermée.
Source	Qui contribue : <code>core</code> , <code>user</code> , ou l'identifiant d'un plugin. C'est la clef de tout le reste, du nommage à la libération.
Jeton	Ce que rend toute inscription. L'appeler défait exactement ce que l'inscription avait fait.
Registre	Une table par type. Les surfaces d'affichage lisent le registre et ne connaissent jamais leurs contributeurs.



Une surface n'a jamais de liste écrite en dur. Elle lit le registre, et se trouve enrichie par un plugin sans qu'une ligne du noyau ne bouge.

Le jeton, et pourquoi il n'est pas optionnel

Toute inscription rend un jeton. Le motif existait déjà dans l'application, mais réimplémenté à la main pour chaque type de contribution : des listes de noms sur `window`, des marqueurs `__pluginId`, des attributs `data-plugin-id`, balayés un par un au déchargement. Chacune de ces variantes pouvait oublier un puits, et un rechargement de plugin laissait alors des contributions fantômes.

```

var K = window.IsoFind;

var jeton = K.commands.register({
  id: 'monplugin.saluer',
  source: 'monplugin',
  title: { fr: 'Dire bonjour', en: 'Say hello' },
  handler: function () { K.notifications.info('Bonjour'); }
});

jeton.dispose();           // retire cette commande
K.releaseSource('monplugin'); // retire tout ce que ce plugin a inscrit
  
```

Un magasin par source accumule les jetons, et la libération se fait en ordre inverse d'inscription : une contribution tardive peut dépendre d'une antérieure, jamais l'inverse.

Déclarez toujours votre source. Une contribution sans `source` est rangée sous `core`, et `releaseSource` ne la libèrera jamais. Elle survivrait au déchargement de son plugin, vidée de son code, et son entrée de menu mènerait à rien. Les registres récents avertissent dans la console quand la source manque ; les plus anciens se taisent.

Un identifiant appartient à une source

Une source peut remplacer sa propre contribution : c'est ce qui permet à un plugin de déclarer une commande dans son manifeste, puis d'enregistrer sa véritable implémentation sous le même identifiant une fois activé. Une source ne peut jamais écraser celle d'une autre. Le nommage n'est pas une convention de propreté, c'est la frontière qui empêche un plugin de surcharger la navigation du noyau en déclarant `core.go.dashboard`.

Un refus rend un jeton inerte, marqué `accepte: false`. Testez-le si le silence vous coûterait cher : sans cela, vous croiriez vous être enregistré, et l'absence se manifesterait plus tard, ailleurs, sans cause visible.

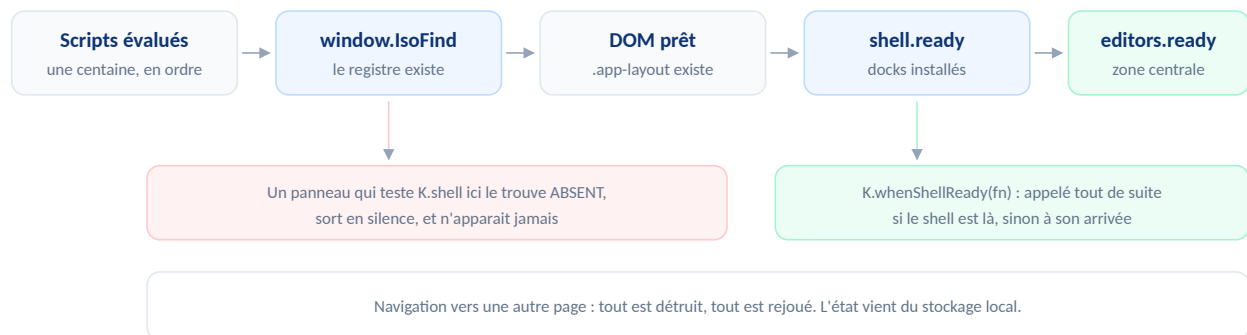
Les titres bilingues

Deux formes sont acceptées partout. Un objet `{ fr, en }` quand le contributeur porte lui-même ses chaînes, ce qui est le cas des plugins, qui ne peuvent pas écrire dans le dictionnaire de l'application. Une clef de traduction quand le contributeur est le cœur et que la chaîne vit dans `translations.js`. `K.resolveTitle` résout les deux, et `K.currentLang()` rend la langue courante.

3. Le cycle de vie d'une page

IsoFind est une application multi-pages. Chaque lien de la barre latérale détruit le document et le reconstruit. C'est la contrainte qui explique la forme de toutes les API qui suivent.

Rien ne persiste en mémoire d'une page à l'autre. Ce qui persiste, c'est l'enregistrement, rejoué au chargement, plus l'état de disposition relu depuis le stockage local. Toute contribution doit donc être bon marché à reconstruire, et tout panneau à état lourd doit savoir sérialiser cet état plutôt que de le garder en mémoire.



La panne la plus difficile à voir

Les scripts sont évalués pendant l'analyse du document ; le shell, lui, ne s'installe qu'au DOM prêt, parce qu'il a besoin de `.app-layout`. Un panneau qui teste `K.shell` à son chargement le trouve donc absent, et un retour anticipé le fait disparaître sans aucun message : le fichier est bien chargé, la fonction s'exécute, et rien n'apparaît.

`K.whenShellReady(fn)` existe pour cela. Le patron était recopié dans chaque panneau, donc oubliable ; le poser dans le noyau est la seule façon de ne plus l'oublier.

```
(function () {
  'use strict';
  var K = window.IsoFind;
  if (!K) return; // charge hors de l'application

  K.whenShellReady(function () {
    K.shell.registerPanel({ /* ... */ });
  });
})();
```

Le même patron vaut pour la zone centrale, qui pose `editors.ready`, et pour tout ce qui dépend d'une surface installée plus tard. La liste des clefs de contexte standard est en section 13.

Script classique, pas module

Le noyau et les fichiers du cœur sont des scripts classiques. Un module serait différé et s'exécuterait après la centaine de scripts globaux, donc trop tard pour que le cœur puisse s'enregistrer au chargement. Si vous ajoutez un fichier au cœur, gardez la même forme : une fonction immédiatement invoquée, `'use strict'`, et une sortie propre quand le noyau est absent.

4. Le manifeste v3

Un plugin v2 s'installait lui-même : pour savoir ce qu'il apportait, il fallait l'exécuter. Le v3 renverse cela, et c'est ce qui remet le consentement à l'endroit.

Avec l'ancien modèle, on demandait des permissions avant de savoir ce que le plugin ferait, sur la foi d'une liste de capacités abstraites. Et l'on exécutait tous les plugins actifs au démarrage, parce que c'était le seul moyen de découvrir ce qu'ils apportaient. Un plugin dont l'utilisateur ne se servirait pas de la journée tournait quand même, avec toute la surface d'attaque que cela suppose.

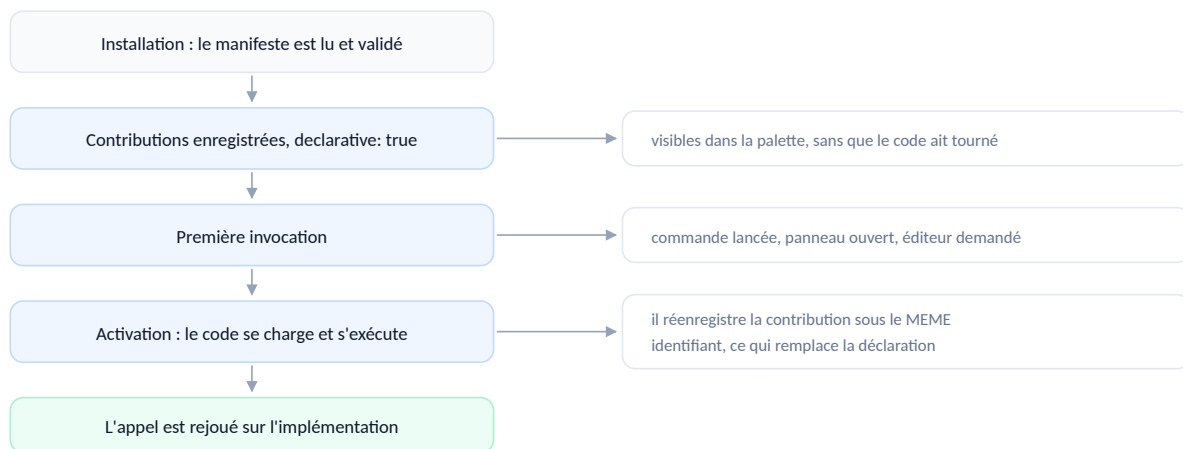
```
{
  "manifest": 3,
  "id": "monplugin",
  "name": "Mon plugin",
  "version": "1.0.0",

  "engine": { "runtime": "js" },          // js | python | native
  "capabilities": ["samples:read", "ui:panel"],

  "activation": [                        // quand executer le code
    "onCommand:monplugin.analyser",
    "onPanel:monplugin.resultats"
  ],

  "contributes": {
    "commands": [
      { "id": "monplugin.analyser",
        "title": { "fr": "Analyser la sélection", "en": "Analyse selection" },
        "category": "monplugin",
        "keybinding": "ctrl+alt+a",
        "when": "editor.active" }
    ],
    "panels": [
      { "id": "monplugin.resultats", "dock": "right",
        "title": { "fr": "Résultats", "en": "Results" }, "icon": "..." }
    ],
    "editors": [ { "kind": "monplugin.spectre", "title": { /* ... */ } } ],
    "menus": [ { "cible": "db.ligne", "command": "monplugin.analyser" } ],
    "settings": [ { "key": "monplugin.seuil", "type": "number", "default": 3 } ]
  }
}
```

L'activation paresseuse, et le gestionnaire de remplacement



Un panneau déclaré s'affiche immédiatement, en attente, pendant que le code charge. Charger avant d'ouvrir donnerait une fenêtre qui reste vide sans qu'on sache si elle travaille ou si elle a échoué.

Si votre plugin déclare `onStartup`, son code est exécuté au démarrage comme en v2. Réservez-le à ce qui doit réellement tourner sans être appelé, et sachez que c'est le seul cas où la validation ne signale pas une contribution qu'aucune activation ne vise.

Ce que la validation refuse, et pourquoi

Refus	Ce qu'il ferme
Un identifiant qui ne commence pas par <code><id du plugin></code> .	Le plugin déclarerait <code>core.go.dashboard</code> et détournerait la navigation du noyau.
Une expression <code>when</code> invalide	Une expression fausse s'évalue à faux, la commande disparaît sans message, et l'auteur cherche un bogue du noyau.
Un dock autre que <code>left</code> , <code>right</code> , <code>bottom</code>	Le panneau ne serait rattaché nulle part.
Une entrée de menu vers une commande non déclarée	Un menu qui mène à rien.
Une contribution qu'aucune activation ne vise	Du code mort, et le signe d'un manifeste bricolé à la main.
Le runtime <code>native</code> sans la capacité <code>runtime:native</code>	C'est exactement le manifeste qu'écrirait un attaquant pour obtenir la pleine confiance sans la demander.

Les expressions de contexte

Le champ `when` d'une commande déclarée est évalué par le noyau, sans exécuter le plugin : c'est ce qui permet de masquer une commande non pertinente sans jamais charger son code. La grammaire est volontairement pauvre, et il n'y a pas d'`eval` : clefs, négation, conjonction, disjonction, égalité. Une expression de manifeste est une donnée fournie par un tiers, et l'évaluer serait exactement le trou qu'on passe son temps à fermer ailleurs.

```
"when": "editor.active"
"when": "!plugins.unconfined"
"when": "core.page == echantillons && kernel.zone == local"
```

Un manifeste v2 reste accepté. C'est exactement un v3 avec `onStartup` et zéro contribution déclarée. Ce n'est pas une coïncidence : c'est la définition du modèle qu'on remplace, et qu'il s'exprime sans effort dans le nouveau vocabulaire est ce qui montre que celui-ci est plus général, et pas seulement différent.

Les points de contribution

Le catalogue : ce que chaque registre attend, ce qu'il rend, et les pièges de chacun.

5. Commandes

Le chemin de dispatch unique. Palette, raccourci, menu, barre d'outils, console, script, assistant : tous convergent ici. C'est ce qui rend l'application pilotable, et c'est le seul endroit où la capacité et la zone sont vérifiées, donc le seul endroit à auditer.

```
K.commands.register({
  id:      'monplugin.analyser',      // prefixe par la source, obligatoire
  source:  'monplugin',
  title:   { fr: 'Analyser', en: 'Analyse' },
  category: 'monplugin',              // regroupe dans la palette
  capability: 'samples:read',         // verifiée à chaque appel
  zone:    null,                     // 'local' pour réserver à la zone locale
  when:    function (ctx) { return !!ctx.get('editor.active'); },
  keybinding: 'ctrl+alt+a',
  quickAccess: true,                  // propose en accès rapide
  handler:  function (args, ctx) { return Promise.resolve(); }
});
```

Appel

Effet

<code>K.commands.execute(id, args)</code>	Rend une promesse. Vérifie la zone, la capacité et <code>when</code> , réveille la source si besoin, puis appelle le gestionnaire.
<code>K.commands.isAvailable(id)</code>	La même règle, sans exécuter. C'est ce que les vues emploient.
<code>K.commands.list(opts)</code>	Les commandes disponibles, avec leur libellé résolu. Options : <code>quickAccess</code> , <code>source</code> , <code>category</code> , <code>available: false</code> pour tout voir.
<code>K.commands.override(id, fn)</code>	Enveloppe une commande existante. Zone locale, source <code>user</code> .
<code>K.keybindings.bind(combo, id)</code>	Lie une touche. Table séparée des commandes, pour que le remappage agisse sur la liaison.

Trois évènements accompagnent chaque appel : `command.before`, `command.after` et `command.error`. Ils portent l'identifiant, les arguments, puis le résultat ou l'erreur. C'est par là qu'on instrumente sans toucher aux commandes.

La surcharge

Le motif est celui retenu pour les extensions CRM d'IsoSupply : on enveloppe sans toucher à l'original, et la libération du jeton restaure exactement l'état antérieur. C'est la primitive sur laquelle repose toute la personnalisation avancée.

```
var jeton = K.commands.override('core.enregistrer', function (suite, args, ctx) {
  if (!confirme()) return Promise.resolve(); // on refuse, sans toucher au cœur
  return suite(args).then(function (r) {
    K.log('user', 'Enregistrement passé par la surcharge');
    return r;
  });
});
```

La surcharge n'est pas offerte aux plugins. Elle vit sous la source `user` et suppose la zone locale. Un plugin distribué qui pourrait envelopper les commandes du cœur pourrait tout : ce serait la voie native, sous un autre nom.

Les raccourcis

Un raccourci sans modificateur ne se déclenche pas pendant qu'on écrit. Lier `n` à une commande rendrait sinon impossible de taper la lettre n dans un champ, et le symptôme serait attribué au clavier, jamais au raccourci. Avec modificateur, on laisse passer : `ctrl+f` dans un éditeur doit atteindre l'application, c'est même la raison d'être d'un modificateur.

Un conflit se résout par le contexte : la première liaison disponible l'emporte, et une liaison posée plus tard passe devant.

6. Panneaux

Un panneau vit dans un dock, à gauche, à droite ou en bas. Il peut ensuite être déplacé, épinglé, ouvert en onglet central, détaché en fenêtre ou rendu flottant, sans que son auteur ait rien à prévoir.

```
K.shell.registerPanel({
  id:      'monplugin.resultats',
  source:  'monplugin',
  title:   { fr: 'Résultats', en: 'Results' },
  icon:    '<svg ...>',           // SVG inline, pas de fichier
  dock:    'right',              // left | right | bottom
  factory: function (ctx) {
    ctx.root.innerHTML = '<div class="mp-corps"></div>';

    // L'etat de la page precedente, s'il y en avait un.
    var etat = ctx.state || { filtre: '' };

    function majEtat() { ctx.saveState(etat); }

    return {
      dispose: function () { /* auditeurs, minuteurs, requetes */ }
    };
  }
});
```

ctx	Contenu
ctx.panelId	L'identifiant du panneau
ctx.root	Le nœud où construire. Vide à l'appel.
ctx.state	Ce que <code>saveState</code> avait rangé, ou <code>null</code>
ctx.saveState(obj)	Range un état sérialisable. C'est par cet appel, et seulement par lui, qu'un panneau survit à une navigation.
ctx.close()	Referme le panneau depuis l'intérieur

La fabrique rend un contrôleur, dont seul `dispose` est lu. Un panneau qui pose des auditeurs sur `window`, un minuteur ou un abonnement au bus doit les défaire ici, sinon ils survivront à la fermeture et se cumuleront à chaque réouverture.

Tout panneau devient basculable sans que vous y pensiez. L'enregistrement inscrit automatiquement une commande `<id>.toggle`, donc le panneau apparaît dans la palette, dans la recherche universelle et dans le menu Panneaux. C'est le genre d'uniformité que le registre rend gratuite.

Les quatre hôtes possibles

Appel	Effet
<code>K.shell.open / close / toggle</code>	Dans son dock
<code>K.shell.move(id, dock)</code>	Change de dock
<code>K.shell.placer(id, hote)</code>	Libère les trois autres hôtes puis installe dans celui demandé : <code>dock</code> , <code>flottant</code> , <code>onglet</code> , <code>fenetre</code>
<code>K.zone.ouvrirPanneau(id)</code>	En onglet dans la zone centrale
<code>K.rail.epingler(...)</code>	Sur une bande latérale
<code>K.shell.signaler(id)</code>	Pastille d'attention, si le panneau n'est pas déjà sous les yeux

Tous les panneaux ne se détachent pas également. Un panneau dont la fabrique construit dans `ctx.root` traverse les quatre hôtes sans difficulté. Un panneau adaptateur, qui déplace un nœud construit ailleurs, ne le peut pas : le nœud ne peut être qu'à un endroit à la fois. Si vous voulez que votre panneau voyage, construisez dans la racine qu'on vous donne.

La pastille, et quand elle se justifie

Une question qui met trente secondes, on va lire son fichier, et on ne revient jamais : la réponse est là, personne ne le sait.

`K.shell.signaler(id)` le dit une fois, et la marque s'efface à l'ouverture. Elle ne s'affiche pas si le panneau est déjà visible : clignoter pour annoncer ce que l'utilisateur est en train de lire est le plus sûr moyen de lui apprendre à ignorer la pastille.

Les fenêtres détachées

Une fenêtre détachée est un contexte JavaScript entièrement séparé. Le bus du noyau ne franchit pas cette frontière : un émetteur et son abonné doivent vivre dans la même fenêtre pour se voir. Un pont fait traverser une poignée de sujets inscrits explicitement, avec un sens de circulation. Tout relayer serait une faute : la plupart des sujets du bus décrivent l'état interne d'une fenêtre, et les relayer ferait croire à chacune que le registre d'à côté est le sien.

Si votre panneau doit parler à un autre à travers une fenêtre, la ligne s'ajoute dans `pont-fenetres.js`, et c'est un acte délibéré : chaque ligne est un couplage entre deux fenêtres.

7. Éditeurs et zone centrale

La zone centrale est une zone à onglets. Le premier onglet est la page elle-même ; les suivants sont des éditeurs ouverts sur un contenu : un CSV, une table de base, un document ISO, ce que votre plugin sait afficher.

Traiter la page comme un onglet parmi d'autres est le geste qui rend le reste cohérent. Empiler les éditeurs par-dessus la page, en surimpression, aurait donné deux notions d'affichage central, et toutes les questions de focus, de raccourcis et de sauvegarde se poseraient deux fois.

```
K.editors.register({
  kind: 'monplugin.spectre', // prefixe par la source
  source: 'monplugin',
  title: { fr: 'Spectre', en: 'Spectrum' },
  icon: '<svg ...>',

  // Cette reference me concerne-t-elle ?
  match: function (ref) { return /\s.sp$/i.test(ref.chemin || ''); },
  // Le libelle de l'onglet
  label: function (ref) { return ref.nom || 'Spectre'; },
```

```

    factory: function (ctx) {
      // ctx.root, ctx.ref, ctx.state, ctx.saveState, ctx.close
      return { dispose: function () {} };
    }
  });

K.editors.open({ kind: 'monplugin.spectre', chemin: '/data/ech01.spc' });

```

Une référence est résoluble, ou elle ne l'est pas

C'est ce qui décide de sa persistance. Une table de base se rouvre à partir de son nom : elle survit à une navigation. Un fichier déposé n'a pas de chemin que l'application puisse relire ; il ne survit que si son contenu tient dans l'état sérialisé, et l'éditeur en décide lui-même.

Le registre ne fait pas semblant de restaurer ce qu'il ne peut pas retrouver. Au delà d'environ quatre cent mille caractères, un état n'est plus persisté : le quota du stockage local emporterait sinon la disposition entière, y compris celle des autres panneaux.

Appel	Effet
<code>K.editors.open(ref)</code>	Ouvre ou active l'onglet
<code>K.editors.close / activate</code>	Ferme, met au premier plan
<code>K.editors.detach / reattach</code>	Vers un dock, et retour
<code>K.editors.isOpen(ref)</code>	Un booléen
<code>K.zone.scinder / plein / épingle</code>	Volets, plein écran, onglet épinglé

`ctrl+w` ferme l'onglet actif, et la commande correspondante (`core.editor.close`) n'est disponible que si la clef de contexte `editor.active` est posée. C'est un bon exemple du patron : le raccourci n'est pas gardé par une condition écrite dans un écouteur, il est gardé par la disponibilité de la commande.

8. Menus

Deux registres distincts : la barre de menus en haut, et les menus contextuels attachés à des types d'objets.

Le menu contextuel

La cible est un type d'objet, pas un sélecteur CSS. `db.ligne` plutôt que `.db-row` : un sélecteur lierait les menus à la structure du DOM d'un panneau, renommer une classe casserait un plugin, et le plugin n'aurait aucun moyen de le savoir. Le type est un contrat, la classe est un détail.

```

K.menus.declarer({
  id:      'monplugin.db.simuler',
  source:  'monplugin',
  cible:   'db.ligne',           // le TYPE d'objet
  ordre:   200,                 // les petits en premier
  items:   function (ctx) {
    return [{
      id:      'simuler',
      titre:   { fr: 'Simuler dans PHREEQC', en: 'Simulate in PHREEQC' },
      groupe:  'analyse',        // les groupes sont separees par un trait
      ecrit:   false,
    }];
  }
});

```

```

    actif: function () { return !!ctx.ligne; },
    raison: { fr: 'Sélectionnez une ligne', en: 'Select a row' },
    executer: function (c) { /* ... */ }
  }];
}
});

```

Un item peut aussi porter `commande: 'monplugin.analyser'`, et son titre comme son raccourci viennent alors du registre des commandes : une seule définition, et le menu suit un renommage tout seul.

Ce qui est indisponible est grisé, jamais caché

Cacher un item selon l'état fait croire qu'il n'existe pas, et l'utilisateur cherche ailleurs une fonction qu'il avait sous la main. Un item inactif reste visible, avec sa raison.

Sur un logiciel dont la provenance est signée, « Tirer maintenant » et « Copier le chemin » n'engagent pas la même chose. Les items marqués `ecrit` sont séparés des autres et poussés en bas du menu, et `danger` les passe en rouge. Ce n'est pas de la décoration : c'est ce qui fait qu'un geste rapide, fait sans lire, reste un geste sûr. `confirmation` ajoute une question avant d'agir.

Les sous-menus tiennent sur un seul niveau, et c'est assez. `K.menus.ouvrir(cible, ctx, x, y)` permet l'ouverture impérative, pour un panneau dont les items ne sont connus qu'à l'instant du clic, comme l'explorateur de fichiers dont la liste d'ouvreurs dépend du fichier visé.

La barre de menus

```

K.menubar.register({
  id: 'monplugin.menu',
  title: { fr: 'Mon plugin', en: 'My plugin' },
  ordre: 60, // Fichier 5, Aller 8, Panneaux 10, Journal 20
  items: function () { return [ /* memes items */ ]; }
});

```

9. Recherche, état, tableau de bord

Trois surfaces qui n'ont rien d'écrit en dur, et qu'un plugin enrichit de la même façon qu'il enrichit un menu.

La recherche universelle

Elle s'ouvre par un double appui sur Maj. Un fournisseur reçoit la requête et rend des résultats ; le noyau les groupe, les ordonne et gère la navigation au clavier.

```

K.recherche.enregistrer({
  id: 'monplugin.spectres',
  source: 'monplugin',
  title: { fr: 'Spectres', en: 'Spectra' },
  order: 120, // les petits en premier
  icone: '<svg ...>',
  chercher: function (requete) {
    // rend un tableau, ou une promesse de tableau
    return [{ id: 's1', libelle: 'Spectre 1', sousTitre: 'ech-01',
      ouvrir: function () { /* ... */ } }];
  }
});

```

La barre d'état

```
K.statut.enregistrer({
  id:      'monplugin.etat',
  source:  'monplugin',
  title:   { fr: 'Moteur', en: 'Engine' },
  side:    'right',           // left | right
  order:   100,
  defaultOn: true,           // visible par défaut
  render:  function (hote) { hote.textContent = 'prêt'; },
  onClick: function () { K.shell.toggle('monplugin.resultats'); }
});
```

Un indicateur écrit ses libellés à la construction. Le registre le reconstruit lors d'un changement de langue, donc n'y mettez pas d'état que vous ne sauriez pas rebâtir. L'utilisateur peut masquer n'importe quel indicateur, le vôtre compris, et

`K.statut.disponibles()` rend la liste réelle, plugins compris, pour les préférences.

Le tableau de bord

Le tableau de bord a cessé d'être une page écrite d'avance pour devenir une surface. Un tableau de bord codé en dur ne peut montrer que ce que son auteur a prévu : il ne montrera jamais la dernière simulation d'un plugin, puisque la page ne connaît pas le plugin.

```
K.dashboard.registerWidget({
  id:      'monplugin.dernieres',
  source:  'monplugin',
  order:   50,
  factory: function (hote) {
    hote.innerHTML = '<h3>Dernières simulations</h3>';
    return { dispose: function () {} };
  }
});
```

Ce qu'un tableau de bord doit répondre. Une seule question : « où en étais-je ». Pas « combien d'échantillons ai-je », qui est une statistique et qu'on regarde une fois. La reprise du travail est la raison pour laquelle on ouvre un logiciel le matin, et c'est la seule chose qu'une page d'accueil puisse faire mieux qu'un menu.

`K.dashboard.remonter()` redessine le tableau de bord là où il est déjà monté. Sans cela, un utilisateur qui lance un export et revient au tableau de bord ne verrait rien, et conclurait que rien n'a été enregistré.

10. Tâches de fond, notifications, modales

Trois façons de parler à l'utilisateur, et elles ne sont pas interchangeables. Une opération longue déclare une tâche, un fait accompli produit une notification, une question bloque et attend.

Les tâches de fond

Un import met deux minutes, une simulation davantage, une génération de texte s'étire. Sans registre, chacune signale son avancement à sa façon, ou pas du tout, et l'utilisateur ne sait pas si l'application réfléchit ou si elle est bloquée.

```
var t = K.taches.demarrer({
  titre:  'Simulation PHREEQC',
  sujet:  'ech-01',
  source: 'monplugin',
```

```

    determinee: true, // false : barre indeterminée
    onAnnuler: function () { moteur.stop(); }
  });

t.progresser(0.34, '340 lignes sur 1000');
t.message('Équilibrage');
t.reussir('12 espèces calculées');
// ou t.echouer(erreur), ou t.annuler()

```

Deux sortes de progression, et il faut les deux

Certaines tâches connaissent leur avancement. D'autres non : une génération ne sait pas quand elle finira. La première a une barre qui se remplit, la seconde une barre indéterminée. Forcer un pourcentage sur la seconde afficherait un chiffre inventé, pire qu'une absence de chiffre, puisqu'on croirait pouvoir estimer la fin.

Une tâche finie ne disparaît pas aussitôt. Une tâche qui s'évanouit à la milliseconde où elle se termine ne laisse pas voir qu'elle a réussi, ni surtout qu'elle a échoué. Les échecs restent jusqu'à ce qu'on les acquitte : un import raté qui disparaît tout seul est un import dont on ne saura jamais qu'il a raté.

Les notifications

```

K.notifications.info('Export terminé');
K.notifications.attention('3 fichiers ignorés');
K.notifications.erreur('Source injoignable');
window.showNotification(message, 'error'); // forme historique, équivalente

```

Une notification ne garde rien. Pour ce qui doit survivre à la nuit, il y a les alertes de workflow, que le panneau Workflows présente au démarrage suivant. Le système de notifications les relaie en une seule ligne plutôt que d'en empiler quarante : quarante bandeaux seraient quarante fois moins lus qu'un seul.

Les surfaces flottantes partagées

`window.IsoFindModales` porte le fond, la modale, les boutons, les popups et les listes déroulantes que tout le logiciel emploie. Trois formes couvrent presque tous les cas, et elles rendent des promesses.

```

IsoFindModales.confirmer({ titre: 'Supprimer ?', message: '...' })
  .then(function (ok) { if (ok) supprimer(); });

IsoFindModales.informer({ message: 'Terminé' }); // ce que faisait alert()
IsoFindModales.demander({ titre: 'Nom du lot' }); // ce que faisait prompt()

```

`window.confirm` et `window.prompt` sont cassés sous Tauri. Ils ne s'affichent pas, ou bloquent la fenêtre. Toute l'application est passée à ces modales en promesse ; un plugin qui appellerait les fonctions natives verrait son code s'arrêter sans message.

11. Enregistrement, fichiers, projet

Déclarer ce qui est modifié

Le noyau tient une notion unique d'état non enregistré, à laquelle chaque éditeur, chaque panneau, chaque console s'inscrit. C'est ce qui permet à `core.enregistrer` de tout enregistrer, et à une fermeture de prévenir.

```
K.enregistrement.declarer({
  id:      'monplugin.spectre',
  source:  'monplugin',
  title:   { fr: 'Spectre', en: 'Spectrum' },
  modifie: function () { return sale; },
  enregistrer: function () { return ecrire(); } // promesse acceptee
});
```

Les ouvreurs de fichiers

Ce qui se passe quand on clique un fichier dans l'explorateur. La tentation serait d'écrire un `switch` sur l'extension dans le panneau : ce serait la neuvième liste en dur, et la neuvième fois qu'un plugin ne pourrait rien y ajouter.

```
K.openers.declare({
  id:      'monplugin.spectre',
  source:  'monplugin',
  title:   { fr: 'Ouvrir comme spectre', en: 'Open as spectrum' },
  extensions: ['spc', 'sp'], // ou accepte: function (chemin) -> bool
  default: true, // premier propose
  ouvrir:  function (chemin, nom) { return K.editors.open({ /* ... */ }); }
});
```

Plusieurs ouvreurs peuvent accepter la même extension, et c'est voulu. Un `.csv` peut être ouvert dans le tableur, joint à l'assistant, ou ingéré par le LIMS : ce ne sont pas des variantes d'une même action, ce sont trois intentions différentes, et seul l'utilisateur sait laquelle est la sienne. Tous les ouvreurs qui acceptent le fichier sont proposés, le premier étant le défaut. Les défauts du cœur passent devant : un plugin chargé tard ne doit pas voler le comportement d'un format du cœur, l'utilisateur verrait son double-clic changer de sens après une installation sans comprendre pourquoi.

Les fournisseurs de projet

L'arbre d'un projet n'est pas écrit dans le panneau. Chaque famille d'objets s'y déclare, et un plugin peut y ajouter la sienne.

```
K.projet.fournir({
  id:      'monplugin.spectres',
  titre:   { fr: 'Spectres', en: 'Spectra' },
  ordre:  40,
  compter: function (projet) { return 12; },
  lister:  function (projet) { return [{ id: 's1', libelle: 'Spectre 1' }]; },
  ouvrir:  function (element) { /* double-clic sur une feuille */ },
  rattachable: true
});
```

L'arbre suit l'ordre déclaré, pas l'alphabet : « Échantillons » avant « Analyses » parce que les seconds découlent des premiers, et l'ordre raconte cette dépendance. Trier alphabétiquement mélangerait la logique du travail.

12. Connecteurs, LIMS, workflows

Les connecteurs de raisonnement

Un connecteur relie IsoFind à un moteur local. L'abstraction existe pour qu'IsoFind ne dépende d'aucun, et un plugin peut en déclarer un.

Le connecteur ne fait pas l'appel réseau

S'il effectuait lui-même son `fetch`, il appellerait ce qu'il veut, et la souveraineté d'IsoFind ne serait plus qu'une promesse tenue par le code d'autrui. Le connecteur déclare une adresse et met en forme ses requêtes ; le noyau transporte, et refuse toute adresse qui n'est pas la machine locale.

La différence n'est pas rhétorique : une promesse se vérifie en relisant tout le code, une contrainte se vérifie en relisant un fichier.

```
K.connectors.declare({
  id:      'monplugin.moteur',
  title:   { fr: 'Mon moteur', en: 'My engine' },
  base:    'http://127.0.0.1:11434', // refuse si non locale
  dialoguer: function (messages, options) { /* met en forme, rend une requete */ }
});
```

Trois pièges sont fermés par le transport, et aucun n'est théorique. Les redirections : un serveur local peut répondre 302 vers l'extérieur, donc le transport refuse de suivre. Les noms qui résolvent ailleurs : seuls les littéraux d'adresse locale et `localhost` sont admis. Les adresses déguisées : `2130706433` ou `127.1` sont normalisées par `new URL()`, mais `127.0.0.1.evil.com` reste un nom de domaine, et c'est pourquoi la comparaison porte sur l'hôte entier, jamais sur son début.

Les connecteurs LIMS

Même principe pour l'acquisition : un connecteur déclare comment lire une source d'instruments, et le registre vérifie le verrou de transport au dernier moment. Un plugin qui écrirait un `fetch` direct vers la route d'acquisition contournerait la règle qui interdit aux connecteurs réseau de sortir de la machine. `K.lims` porte `tirer`, `tester`, `transportDe` et `verrou`.

Les tâches de workflow

Une tâche déclarée par un plugin apparaît dans l'éditeur de workflow, dans le calcul des capacités, dans le journal des accès, et, si elle le permet, dans les outils de l'assistant. Rien à câbler : le registre suffit.

```
K.workflows.declarerTache({
  id:      'monplugin.ranger',
  source:  'monplugin',
  ecrit:   true, // absent vaut « écrit » : le default prudent
  outil:   false, // invisible au modele
  capacite: 'samples:write',
  title:   { fr: 'Ranger un lot', en: 'File a batch' },
  entrees: [ { nom: 'lot', type: 'texte', requis: true,
              libelle: { fr: 'Identifiant du lot', en: 'Batch id' } } ],
  sorties: [ { nom: 'ranges', type: 'nombre' } ],
  executer: function (args, ctx) { return Promise.resolve({ ranges: 3 }); }
});
```

Le drapeau `ecrit` décide de la visibilité en mode conversation et de la simulation à vide ; `outil: false` retire la tâche au modèle dans tous les modes. Le guide des workflows détaille les deux, ainsi que les capacités, le journal des accès et le catalogue complet des tâches du cœur.

Les outils de l'assistant

`K.tools` transforme les tâches en schémas d'outils pour le modèle : `schemas('agent')` ou `schemas('conversation')`, `appeler(nom, args)` pour l'exécution, `lisible` pour l'affichage. Un plugin n'a normalement rien à faire ici : il déclare une tâche, et l'assistant la voit ou non selon les deux drapeaux.

13. Réglages, contexte, évènements

Les réglages en couches

Quatre couches, de la moins forte à la plus forte : `default`, `edition`, `user`, `project`. La valeur effective est calculée à la lecture, jamais figée à l'écriture ; sans cela, une surcharge posée tardivement n'aurait aucun effet sur ce qui a déjà lu.

```
K.settings.declare({ id: 'monplugin.seuil', source: 'monplugin',
                    type: 'number', default: 3,
                    title: { fr: 'Seuil', en: 'Threshold' } });

K.settings.get('monplugin.seuil');
K.settings.set('monplugin.seuil', 5); // couche user, persistee
K.settings.onChange('monplugin.seuil', function (v) { /* ... */ });
```

Les clefs de contexte

Le contexte est le reflet de l'état courant contre lequel s'évaluent les prédicats de disponibilité. C'est ce qui permet à une commande de n'exister que là où elle a un sens, sans que le déclencheur ait à le savoir.

Clef	Posée par
<code>shell.ready</code>	Les docks sont installés
<code>editors.ready</code> , <code>zone.ready</code>	La zone centrale est prête
<code>actions.ready</code>	Le panneau Actions est monté
<code>editor.active</code>	Un onglet d'éditeur a le focus
<code>core.page</code>	La page courante de l'application
<code>core.hasSidebar</code>	La page porte une barre latérale
<code>kernel.zone</code>	<code>distribution</code> ou <code>local</code>
<code>dev</code> , <code>dev.build</code>	Mode développeur, build de développement
<code>plugins.unconfined</code>	Un plugin non confiné est chargé
<code>plugin.<id>.active</code>	Ce plugin a été activé
<code>fichier.ouvrir</code> , <code>isof.importer</code> , <code>isof.verifier</code>	Points d'accroche du domaine

Posez les vôtres avec `K.context.set('monplugin.pret', true)`, et lisez-les dans un `when`. Un changement notifie tous les auditeurs, ce qui permet à une surface de se redessiner sans sondage.

Le bus, et les hooks bloquants

Deux régimes. La notification simple, après coup, non annulable. Et le hook bloquant, avant coup, qu'un abonné peut refuser : c'est le point d'injection propre pour les scripts de surcharge, qui peuvent ainsi refuser une sauvegarde ou un lancement de moteur sans toucher au cœur.

```
K.events.on('command.after', function (p) { console.log(p.id); });

K.events.emit('monplugin.calcul.fini', { n: 12 });
```

```
K.events.emitBlocking('monplugin.avant.ecriture', charge)
  .then(function (ok) { if (ok) ecrire(); }); // false = un abonne a refuse
```

Un abonné qui lève ne casse pas la chaîne : l'erreur est consignée et les autres abonnés continuent. Les sujets du noyau les plus utiles sont `registry.changed`, `setting.changed`, `command.before`, `command.after`, `command.error`, auxquels s'ajoutent ceux des registres : `dashboard.changed`, `openers.changed`, `connectors.changed`, `projet.changed`.

Le journal

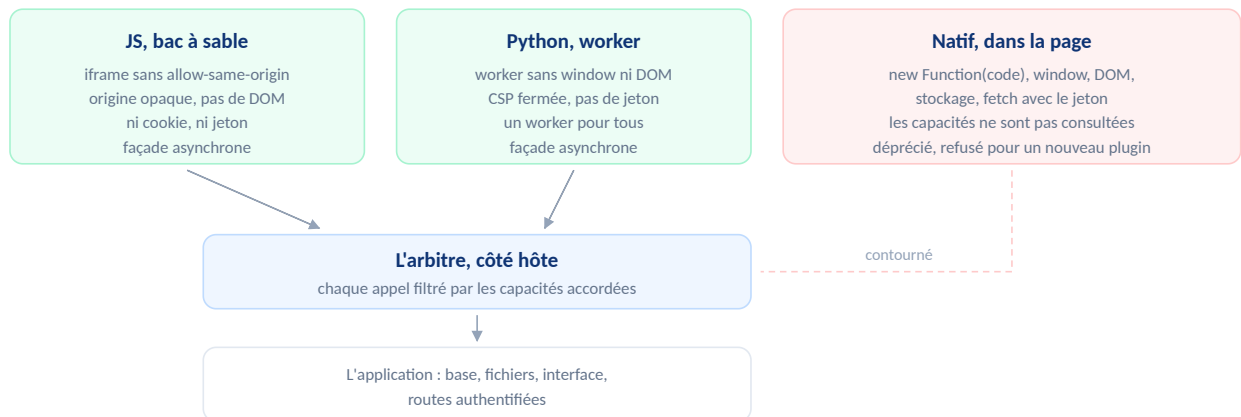
`K.log(source, texte, niveau)` écrit dans le panneau Journal, qui est la trace d'intention : ce que l'utilisateur a entrepris. À ne pas confondre avec le registre des modifications, qui dit ce qui a changé dans les données, durablement et sous signature. Les deux sont nécessaires, et un logiciel de traçabilité qui ne montrerait que le premier mentirait par omission.

Exécution et sécurité

Où tourne le code d'un plugin, ce qui décide de son chargement, et ce que le noyau refuse quoi qu'on lui demande.

14. Les trois runtimes

Trois voies coexistent, et elles n'offrent pas le même confinement. La différence n'est pas une préférence de langage, c'est ce qui décide de ce qu'un plugin peut atteindre.



La voie JS

Elle est déjà confinée, et bien. Iframe en bac à sable sans `allow-same-origin`, donc origine opaque : pas d'accès au DOM de l'hôte, pas de cookie, pas de jeton, et une surface d'appel filtrée par capacités. C'est la voie recommandée pour tout nouveau plugin.

La voie Python

Elle ne l'était pas, et c'était un trou laissé ouvert par inadvertance : le worker existait, fonctionnait, et personne ne l'appelait. Un plugin Python disposait donc de `window`, du DOM et du jeton d'API. Le confinement était écrit et débranché, ce qui est pire qu'absent, puisqu'on croyait l'avoir.

Le worker est désormais le seul chemin. Corollaire assumé : la façade Python est devenue asynchrone. Le synchrone n'était possible que parce que rien ne séparait Python de la page.

```
# Dans un plugin Python
async def setup(isofind):
    samples = await isofind.get_samples()
    await isofind.notify(f"{len(samples)} échantillons")

    isofind.register_formula("delta_norm", calculer,
                           group="Mon plugin", syntax="delta_norm(x)")
    isofind.register_panel("resultats", "Résultats", rendre)
    isofind.register_menu("analyser", "Analyser", action)
```

La façade offre `get_samples`, `get_samples_df`, `get_analyses`, `add_sample`, `run_analysis`, `notify`, `get_config`, `set_config`, `open_panel`, et les trois enregistrements ci-dessus. Chacun devient un message que l'hôte arbitre, et c'est là, et seulement là, que les capacités sont vérifiées.

La voie native n'est pas confinable. Elle exécute `new Function(code)` dans la page, avec `window`, le DOM, le stockage et le `fetch` authentifié. Aucun garde ne changera cela : gouverner un accès total reste un accès total. Elle fonctionne encore, elle avertit, et elle est refusée pour tout nouveau plugin. La suppression d'un coup casserait les consoles existantes, ce qui rendrait la sécurité responsable d'une régression, et c'est la meilleure façon de la faire abandonner.

15. Zone, source, signature

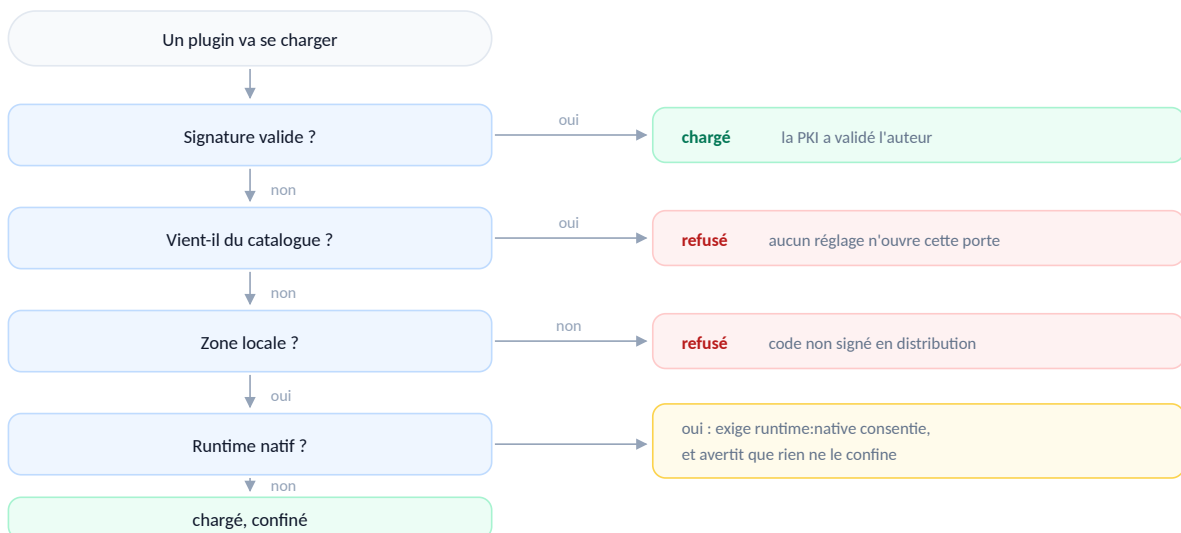
Deux axes indépendants, et les confondre est ce qui rend une case à cocher dangereuse.

La zone contrôle l'exécution de code non signé. La source dit d'où vient le plugin. Un plugin non signé importé depuis un fichier local, en zone locale, est autorisé. Un plugin non signé venant du catalogue est refusé, quelle que soit la zone.

Pourquoi le second invariant compte le plus

On peut convaincre quelqu'un de cocher une case. On ne peut pas faire qu'un plugin distant non signé devienne installable, parce qu'aucun réglage n'ouvre cette porte. Le scénario d'attaque, qui est toujours « activez le mode développeur et installez ceci depuis le catalogue », perd sa seconde moitié.

Le contrôle se fait au chargement, pas à l'installation. Une fiche de plugin vit dans le stockage local, qui est modifiable ; un contrôle au chargement s'exerce à chaque démarrage, sur ce qui va réellement s'exécuter, et c'est le seul moment qui compte.



Entrer en zone locale sur un build de distribution reste un acte grave. Ce n'est pas interdit, et le refuser serait une erreur : un laborantin qui écrit son propre plugin sur une installation de production est exactement le public visé, et lui refuser le sideload viderait le produit de son sens. Le risque n'est pas le développeur, c'est l'utilisateur manipulé. L'entrée exige donc une confirmation qui nomme les choses, et laisse une marque permanente et visible : un mode dangereux dont on oublie qu'il est actif est pire qu'un mode absent.

16. Capacités et consentement

Le noyau ne décide rien ici, il interroge. Les politiques restent chez l'hôte, qui seul connaît la PKI, le consentement accordé à l'installation et le mode développeur. Le noyau se contente de refuser un dispatch quand l'hôte répond non, ce qui garantit qu'il n'existe pas de chemin latéral.

```
K.security.setZone('local');
K.security.setCapabilityProvider(function (source) { return accordees(source); });
K.security.setActivator(function (source) { return charger(source); });
```

Une commande porteuse d'une `capability` est refusée si la source ne l'a pas obtenue. Le cœur n'est pas soumis au consentement : il est l'application. Toute autre source doit prouver sa capacité, et l'absence de fournisseur vaut refus, jamais autorisation.

Le vocabulaire, en entier

Treize capacités, et pas une de plus : une capacité hors de cette liste est ignorée à l'installation, avec un avertissement en console. Elle n'ouvre rien, et la ranger avec les autres laisserait croire le contraire.

Capacité	Ce que l'utilisateur lit
<code>samples:read</code>	Lire les échantillons : accéder aux données d'échantillons chargées
<code>samples:write</code>	Ajouter des échantillons : créer de nouveaux échantillons dans le projet
<code>analysis:run</code>	Lancer des analyses : déclencher le moteur d'analyse
<code>config:read</code>	Lire la configuration : consulter les réglages de l'application
<code>config:write</code>	Modifier la configuration : enregistrer des réglages
<code>ui:notify</code>	Afficher des notifications : émettre des messages à l'écran
<code>ui:menu</code>	Ajouter des entrées de menu : étendre les menus de l'application
<code>ui:panel</code>	Afficher des panneaux : ouvrir une interface dédiée
<code>ui:formula</code>	Ajouter des formules : étendre les fonctions de la feuille de calcul
<code>storage:read</code>	Lire son stockage : relire ses propres réglages
<code>storage:write</code>	Écrire son stockage : conserver ses propres réglages
<code>compute:r</code>	Exécuter du code R : calculer dans un moteur R isolé, sans accès aux données
<code>network</code>	Accéder au réseau : établir des connexions sortantes

À l'installation, l'écran de consentement affiche ces libellés, cases cochées, et n'accorde que ce qui reste coché. En mode air-gap, `network` est grisée et refusée. Deux capacités du noyau vivent à part parce qu'elles ne se demandent pas de la même façon : `runtime:native`, qui doit être déclarée et signée pour que la voie native soit admise, et les capacités de workflow, qui sont accordées à l'import d'un flux et non à l'installation d'un plugin.

Les façades, et pourquoi il y en a deux

Un plugin confiné n'appelle pas les registres, il appelle un objet nommé `isofind` que son runtime lui pose. Cet objet n'est pas le même selon la voie, et les confondre coute une séance de mise au point : la table du backend hôte sert la voie Python et la voie native, celle du pont sert la voie JS.

Verbe	Capacité	JS	Python
<code>getSamples()</code>	<code>samples:read</code>	oui	oui
<code>getAnalyses()</code>	<code>samples:read</code>	non	oui

Verbe	Capacité	JS	Python
<code>addSample(objet)</code>	<code>samples:write</code>	oui	oui
<code>runAnalysis(params)</code>	<code>analysis:run</code>	oui	oui
<code>getConfig / setConfig</code>	<code>config:read, config:write</code>	oui	oui
<code>notify(message, type)</code>	<code>ui:notify</code>	oui	oui
<code>registerMenu(id, libellé, rappel)</code>	<code>ui:menu</code>	oui	oui
<code>registerPanel(id, libellé, rendu)</code>	<code>ui:panel</code>	oui	oui
<code>registerFormula(nom, fn, ...)</code>	<code>ui:formula</code>	oui	oui
<code>openPanel(id)</code>	<code>ui:panel</code>	oui	oui
<code>log(...)</code>	aucune	oui	oui
<code>storage</code>	<code>storage:read, storage:write</code>	oui, sous-objet	non
<code>r</code>	<code>compute:r</code>	oui, sous-objet	non
<code>version</code>	aucune	chaîne, 2.0.0	-

L'absence de `getAnalyses` côté JS n'est pas un oubli à contourner

Un plugin JS qui veut des mesures à plat les tire de `getSamples()` et aplatit `isotope_data` lui-même : chaque mesure porte `element`, `ratio_value` et `uncertainty`, et le contexte de l'échantillon se recolle en même temps.

La règle générale : avant d'écrire un appel, énumérez ce que la façade expose réellement dans votre runtime. La liste ci-dessus décrit une version donnée, et `Object.keys(isofind)` décrit la vôtre.

Deux verbes, deux formes. Les enregistrements de contribution sont synchrones et ne rendent rien. Les appels de données rendent des promesses, et un `await` oublié vous donne une promesse là où vous attendiez un tableau. Une capacité refusée lève, avec le nom de la capacité, plutôt que de rendre du vide.

Le rendu d'un panneau

C'est le point sur lequel les deux voies divergent le plus, et pour une raison de fond : une `iframe` a son propre document, un `worker` n'en a aucun.

Voie	Le rendu reçoit	Il doit
JS confiné	<code>{ root, id }</code>	Peupler <code>ctx.root</code> . Sa valeur de retour est ignorée, et rendre une chaîne de HTML ne produit donc rien : le panneau s'ouvre et reste blanc.
Python	Rien	Rendre du HTML, que l'hôte affiche inerte dans une <code>iframe</code> cloisonnée sans <code>allow-scripts</code> .
Natif, dans la page	Le conteneur réel	Le peupler, avec tout le DOM de l'application à portée.

Le rendu est synchrone, même quand les données ne le sont pas

Rien ne garantit que l'hôte attende une promesse. Un rendu déclaré `async` rend une promesse que personne ne lit, et le panneau reste vide. Peignez un état d'attente tout de suite, puis remplissez : le conteneur reste valide entre les deux.

Un plugin JS confiné peut donc créer des nœuds, poser des écouteurs et dessiner, dans son propre document. Ce qu'il n'a pas, c'est le document de l'hôte.

Le thème

Un plugin ne code pas de couleurs. Il écrit `var(--surface-1)` comme un composant du cœur, et le pont garantit que ces variables existent dans le document du bac à sable, puis les met à jour à chaque changement de thème. La liste exposée est délibérément fermée : les surfaces, les textes, les bordures, les ombres et la marque, plus quelques noms hérités en transition. Tout exposer ferait du socle entier une surface publique qu'on ne pourrait plus faire évoluer sans casser des plugins tiers.

Ne demandez que ce dont vous vous servez. L'écran de consentement pré-coche les lectures et laisse les écritures à cocher. Un plugin qui demande `samples:write` pour afficher un graphique se fait refuser par des utilisateurs attentifs, et c'est le comportement attendu.

17. Ce que le noyau refuse

Six refus, et chacun ferme une porte précise. Les connaitre évite de chercher un bogue là où il y a une règle.

1. **Un identifiant appartenant à une autre source.** Le jeton rendu est inerte et porte `accepte: false`, et une erreur nomme le détenteur. Sans cette règle, un plugin déclarerait `core.go.dashboard` et surchargerait la navigation.
2. **Une contribution de plugin dont l'identifiant ne commence pas par l'identifiant du plugin.** Même frontière, vérifiée dès la validation du manifeste, donc avant que le code ne tourne.
3. **Une commande sans gestionnaire après activation.** Le plugin déclare ce qu'il n'implémente pas. Sans ce garde, on rappellerait le gestionnaire de remplacement, donc l'activation, indéfiniment.
4. **Un connecteur dont l'adresse n'est pas locale.** Refusé à l'enregistrement, pas à l'appel : le laisser apparaître dans l'interface pour le refuser au moment où l'utilisateur en a besoin serait une façon de mentir en différé.
5. **Le runtime natif sans capacité consentie,** et le runtime natif non signé en zone de distribution.
6. **Toute évaluation d'expression fournie par un manifeste.** La grammaire de `when` est pauvre par construction, et il n'y a pas d'`eval`.

Un refus doit se voir

Le noyau rend un jeton inerte plutôt que de lever : l'appelant n'a pas à se protéger d'une exception qu'il ne provoque pas. Mais il ne rend pas non plus un jeton qui ressemble à un succès, et il écrit dans la console qui détient l'identifiant.

Une contribution qui disparaît sans message est la panne la plus coûteuse à diagnostiquer, parce que rien n'invite à chercher au bon endroit.

Écrire un plugin

De l'atelier à la signature, quatre plugins complets, le dépannage et la référence de l'API.

18. De zéro à un plugin chargé

L'atelier de plugin est un panneau de développement : dossier de travail, validation du manifeste, rechargement à chaud, inspection des contributions, empaquetage. Il n'existe qu'en zone locale.

Ce n'est pas un confort d'affichage. Le backend de l'atelier lit et écrit des fichiers sur la machine ; l'exposer hors du mode développeur reviendrait à offrir une lecture de fichier arbitraire depuis l'interface.

Les cinq gestes

Geste	Ce qu'il fait
Générer un squelette	Écrit un manifeste v3 valide et un fichier de code dans un dossier vide. Le plus court chemin vers un plugin qui se charge.
Ouvrir un dossier	L'atelier relit le manifeste, valide, et affiche les erreurs avec leur raison.
Recharger à chaud	Libère la source, relit, réenregistre. Un relevé d'horodatage toutes les 1,2 seconde le fait tout seul si vous laissez l'automatique.
Inspecter	Confronte ce que le manifeste déclare à ce que le registre contient réellement.
Empaqueter	Produit l'archive distribuable.

L'inspecteur, et pourquoi il n'existait pas avant

Le manifeste v3 déclare ce qu'un plugin apporte, et le code l'implémente. Deux choses distinctes, donc deux choses qui peuvent diverger. Un plugin peut déclarer une commande qu'il n'implémente jamais, ou enregistrer une contribution qu'il n'a pas déclarée.

Un écart n'est pas toujours une faute. Une contribution déclarée mais non implémentée peut être un travail en cours. Une contribution enregistrée sans déclaration est plus grave : elle échappera au consentement de l'utilisateur, qui n'aura jamais vu qu'elle existait.

Cet outil n'était pas possible en v2 : il n'y avait rien à confronter, puisque la surface n'était connue qu'après exécution, donc trop tard.

Les deux formes, selon où tourne le code

Un plugin natif s'exécute dans la page et voit le noyau. Un plugin confiné ne le voit pas, et déclare ce qu'il apporte par son manifeste, puis appelle la façade. Les deux formes ci-dessous font la même chose.

```
// Cote page : fichier du coeur, script utilisateur, plugin natif
(function () {
  'use strict';
  var K = window.IsoFind;

  K.commands.register({
    id:      'monplugin.analyser',
    source:  'monplugin',
```

```

        title: { fr: 'Analyser', en: 'Analyse' },
        handler: function () { return analyser(); }
    });
})();

```

```

// Cote bac a sable : K est INDEFINI. On declare dans le manifeste, et
// l'on appelle la facade. Les enregistrements sont SYNCHRONES, les
// appels de donnees rendent des promesses.
(function () {
    'use strict';

    // On s'enregistre AVANT tout appel : un plugin qui leve avant de
    // s'enregistrer ne laisse aucune surface derriere lui.
    isofind.registerPanel('resultats', 'Résultats', function (ctx) {
        var hote = ctx.root; // { root, id }, pas le conteneur
        hote.innerHTML = '<p>Calcul...</p>';

        isofind.getSamples().then(function (echantillons) {
            hote.innerHTML = rendre(echantillons); // l'iframe a son DOM
        }).catch(function (e) {
            hote.textContent = e.message; // capacite refusee : ca leve
        });
    });

    isofind.registerMenu('ouvrir', 'Analyser', function () {
        isofind.openPanel('resultats');
    });
})();

```

Vérifiez la surface avant d'écrire un appel. La façade JS et la façade Python ne portent pas les mêmes verbes, et une version du SDK n'est pas l'autre : `getAnalyses` existe côté Python et pas côté JS, le stockage et R sont des sous-objets côté JS seulement. `Object.keys(isofind)` décrit la vôtre en une ligne, et `isofind.version` la date.

La forme d'un plugin natif

```

// monplugin.js, charge dans le bac a sable
(function () {
    'use strict';
    var K = window.IsoFind;

    // Le code ne tourne qu'a la premiere invocation d'une contribution declaree.
    // Il REENREGISTRE ce que le manifeste a declare, sous les memes identifiants.

    K.commands.register({
        id: 'monplugin.analyser',
        source: 'monplugin',
        title: { fr: 'Analyser', en: 'Analyse' },
        handler: function () { return analyser(); }
    });

    K.shell.registerPanel({
        id: 'monplugin.resultats',
        source: 'monplugin',
        title: { fr: 'Résultats', en: 'Results' },
        dock: 'right',
        factory: function (ctx) { return construire(ctx); }
    });
})();

```

Rien à défaire à la main au téléchargement. Toutes vos inscriptions portent votre source, donc

`K.releaseSource('monplugin')` les retire d'un coup, y compris celles des registres que vous n'avez pas écrits. Ce que vous devez défaire vous-même, c'est ce qui ne passe pas par le registre : un auditeur sur `window`, un minuteur, une requête en vol. C'est le rôle du `dispose` de vos fabriques.

19. Signer et distribuer

Une signature d'auteur permet à un plugin d'être chargé sans passer par la zone locale, et d'être proposé au catalogue. Elle repose sur une clef qui ne quitte jamais votre machine.

Étape	Route	Ce qui se passe
Créer son identité	<code>/plugins/author/identity</code>	Le backend local produit la paire de clefs et l'écrit chiffrée dans le répertoire de données.
Demander un certificat	<code>/plugins/author/request</code>	Une demande portant votre identité part vers l'autorité IsoFind.
Installer le certificat	<code>/plugins/author/import</code>	Le certificat revient et se range à côté de la clef.
Signer	<code>/plugins/author/sign</code>	L'archive est signée localement.
Empaqueter	<code>/plugins/dev/package</code>	L'atelier produit l'archive à distribuer.

La clef ne transite par aucun réseau

C'est la seule propriété qui compte. Une clef de signature qui existe en deux exemplaires ne prouve plus rien, et la révocation d'un certificat dont l'auteur n'a jamais détenu la clef ne révoque rien du tout.

L'interface expose cette contrainte plutôt que de la cacher. Un auteur qui ne comprend pas pourquoi sa clef doit rester chez lui finira par l'envoyer par courriel, en croyant rendre service.

Une signature vérifiée dispense de la zone locale et lève l'interdiction faite aux plugins non signés venant du catalogue. Elle ne dispense pas du consentement : les capacités restent à accorder, et un plugin signé demandant `runtime:native` affiche toujours son avertissement.

20. Quatre plugins types

Quatre formes complètes, du manifeste au code. Les identifiants sont à remplacer, la structure ne bouge pas.

Ces quatre exemples sont écrits côté page. Ils valent pour un fichier du cœur, un script utilisateur ou un plugin natif. Pour un plugin confiné, le manifeste reste le même, et le code appelle la façade de la section 16 au lieu de `K`.

Une commande et son panneau

La forme la plus courante : un calcul lancé depuis la palette, un résultat affiché à droite.

```
// manifeste
{ "manifest": 3, "id": "spectro", "name": "Spectro", "version": "1.0.0",
  "engine": { "runtime": "js" },
  "capabilities": ["samples:read", "ui:panel"],
  "activation": ["onCommand:spectro.analyser", "onPanel:spectro.resultats"],
```

```

"contributes": {
  "commands": [{ "id": "spectro.analyser",
    "title": { "fr": "Analyser le spectre", "en": "Analyse spectrum" },
    "category": "Spectro" }],
  "panels": [{ "id": "spectro.resultats", "dock": "right",
    "title": { "fr": "Spectro", "en": "Spectro" } }]
}
}

// code
K.commands.register({
  id: 'spectro.analyser', source: 'spectro',
  title: { fr: 'Analyser le spectre', en: 'Analyse spectrum' },
  handler: function () {
    var t = K.taches.demarrer({ titre: 'Analyse', source: 'spectro' });
    return analyser()
      .then(function (r) {
        dernier = r;
        t.reussir(r.pics.length + ' pics');
        K.shell.open('spectro.resultats');
        K.shell.signaler('spectro.resultats');
      })
      .catch(function (e) { t.echouer(e); throw e; });
  }
});

```

Un éditeur pour un format

Un ouvreur pour que le double-clic mène quelque part, et un éditeur pour l'afficher en onglet central.

```

K.editors.register({
  kind: 'spectro.fichier', source: 'spectro',
  title: { fr: 'Spectre', en: 'Spectrum' },
  label: function (ref) { return ref.nom; },
  factory: function (ctx) {
    var vue = dessiner(ctx.root, ctx.ref, ctx.state);
    // Ce qui tient dans l'état survit à une navigation ; le reste se relit.
    ctx.saveState({ zoom: vue.zoom });
    return { dispose: vue.detruire };
  }
});

K.openers.declare({
  id: 'spectro.ouvrir', source: 'spectro',
  title: { fr: 'Ouvrir comme spectre', en: 'Open as spectrum' },
  extensions: ['spc'], default: true,
  ouvrir: function (chemin, nom) {
    return K.editors.open({ kind: 'spectro.fichier', chemin: chemin, nom: nom });
  }
});

```

Un widget de tableau de bord alimenté par une tâche

```

K.dashboard.registerWidget({
  id: 'spectro.recents', source: 'spectro', order: 60,
  factory: function (hote) {
    hote.innerHTML = '<h3>Spectres récents</h3><ul></ul>';
    var liste = hote.querySelector('ul');

    function rendre() { /* ... */ }
  }
});

```

```

        rendre();

        // Un widget se redessine quand quelque chose bouge, sans sondage.
        var off = K.events.on('spectro.analyse.finie', rendre);
        return { dispose: function () { off.dispose(); } };
    }
});

```

Une tâche de workflow et son menu contextuel

La tâche rend le plugin utilisable depuis un workflow nocturne, le menu le rend utilisable à la souris. Les deux pointent le même code.

```

K.workflows.declarerTache({
  id: 'spectro.analyser_lot', source: 'spectro',
  ecrit: false, capacite: 'samples:read',
  title: { fr: 'Analyser un lot de spectres', en: 'Analyse a batch of spectra' },
  notice: { fr: '...', en: '...' },
  entrees: [{ nom: 'lot', type: 'texte', requis: true,
    libelle: { fr: 'Identifiant du lot', en: 'Batch id' } }],
  sorties: [{ nom: 'pics', type: 'liste' }, { nom: 'nombre', type: 'nombre' }],
  executer: function (args) { return analyserLot(args.lot); }
});

K.menus.declarer({
  id: 'spectro.menu.echantillon', source: 'spectro',
  cible: 'echantillon', ordre: 300,
  items: function (ctx) {
    return [{ id: 'analyser', commande: 'spectro.analyser', groupe: 'analyse',
      actif: !!ctx.echantillon,
      raison: { fr: 'Aucun échantillon', en: 'No sample' } }];
  }
});

```

21. Dépannage

Les pannes courantes, et par où commencer. La plupart sont des silences, pas des erreurs, et c'est ce qui les rend longues à trouver.

Symptôme	Cause probable	Où regarder
Le panneau n'apparaît pas, aucun message	<code>K.shell</code> était absent au chargement du fichier	Enrober l'enregistrement dans <code>K.whenShellReady</code> .
La contribution disparaît après un rechargement de plugin	Elle a été enregistrée sans <code>source</code> , donc rangée sous <code>core</code> et jamais libérée, puis remplacée	Les avertissements des registres dans la console.
L'enregistrement semble réussir mais rien n'apparaît	L'identifiant appartient à une autre source, le jeton est inerte	L'erreur console nomme le détenteur. Tester <code>jeton.accepte</code> .
La commande n'est jamais proposée	Son <code>when</code> est faux, sa capacité n'est pas accordée, ou elle est réservée à la zone locale	<code>K.commands.list({ available: false })</code> la montre quand même.
Le manifeste est refusé au chargement	Préfixe d'identifiant, dock inconnu, expression invalide, contribution jamais activable	La console liste chaque erreur avec sa raison ; l'atelier les affiche.
« déclare X mais ne l'implémente pas »	Le code s'est chargé sans réenregistrer la contribution déclarée	L'inspecteur de l'atelier confronte les deux listes.

Symptôme	Cause probable	Où regarder
Des auditeurs se cumulent à chaque ouverture	La fabrique ne rend pas de <code>dispose</code> , ou il ne défait pas tout	Ouvrir et fermer le panneau dix fois, regarder ce qui s'accumule.
L'état du panneau est perdu à chaque navigation	Rien n'a été rangé par <code>ctx.saveState</code>	Rappel : aucune mémoire ne survit à une page.
Le connecteur n'apparaît nulle part	Son adresse n'est pas locale, l'enregistrement a été refusé	La console, et le journal de sécurité.
Le plugin Python se bloque sur un appel	La façade est asynchrone depuis le passage au worker	<code>await</code> devant chaque appel à <code>isofind</code> .
Le plugin refuse de se charger en production	Non signé, et venant du catalogue ou hors zone locale	Section 15, et l'arbre de décision.
Une modale ne s'affiche pas et le code s'arrête	Appel à <code>window.confirm</code> ou <code>window.prompt</code>	Passer par <code>IsoFindModales</code> .

Le réflexe utile. Ouvrir la console et lancer `K.list('command', function (c) { return c.source === 'monplugin'; })`. Ce que le registre contient est la vérité ; ce que vous croyez avoir enregistré ne l'est pas toujours.

22. Référence de l'API

Les surfaces publiques, et le fichier qui les installe. Toutes vivent sous `window.IsoFind`, que ce guide appelle `K`.

Le noyau

Surface	Contenu
<code>K.contribute(type, entry)</code>	Inscription générique, rend un jeton
<code>K.get(type, id) / K.list(type, filtre)</code>	Lecture du registre
<code>K.releaseSource(source)</code>	Libère tout ce qu'une source a inscrit
<code>K.whenShellReady(fn)</code>	Attend l'installation du shell
<code>K.commands</code>	register, execute, override, get, isAvailable, list
<code>K.keybindings</code>	bind, unbindAll, list
<code>K.context</code>	set, get, all, onChange
<code>K.events</code>	on, emit, emitBlocking
<code>K.settings</code>	declare, get, set, onChange, declared
<code>K.security</code>	setZone, getZone, setCapabilityProvider, setActivator
<code>K.resolveTitle / K.currentLang</code>	Titres bilingues
<code>K.Disposable / K.DisposableStore</code>	Jetons, et agrégats de jetons

Les surfaces

Tout ce qui suit vit dans la page. Depuis un runtime confiné, rien de cela n'est atteignable directement : on passe par la façade de la section 16, et le pont matérialise la contribution ici.

Surface	Points d'entrée	Fichier
K.shell	registerPanel, open, close, toggle, move, placer, fold, snapshot, restore, signaler, zoneOf	shell-dock.js
K.editors	register, open, close, activate, detach, reattach, isOpen	editor-registry.js
K.zone	ouvrir, ouvrirPanneau, scinder, plein, epingler, deplacer, liste	zone-centrale.js
K.rail	epingler, retirer, deplacer, liste	shell-rail.js
K.menus	declarer, attacher, ouvrir, fermer, items, cibles	menu-contextuel.js
K.menubar	register, refresh	shell-menu.js
K.recherche	enregistrer, ouvrir, fermer, fournisseurs	shell-recherche.js
K.statut	enregistrer, disponibles, basculer, visible	shell-statut.js
K.dashboard	registerWidget, widgets, mount, remonter	dashboard-registry.js
K.taches	demarrer, liste, actives, compte, annuler, acquitter	taches-registre.js

Surface	Points d'entrée	Fichier
K.notifications	info, attention, erreur, taire, rouvrir	notifications.js
K.enregistrement	declarer, enAttente, estModifie, tout	enregistrement.js
K.openers	declare, list, pour, defaultPour, ouvrir	file-openers.js
K.projet	fournir, liste, charger, courant, rattacherLocal, rafraichir	projet-fournisseurs.js
K.connectors	declare, list, get, available, fetch, isLocal	connector-registry.js
K.lims	tirer, tester, transportDe, verrou, get	lims-connectors.js
K.workflows	declarerTache, taches, executer, simuler, deboguer	workflow-engine.js
K.tools	schemas, lisible, tache, appeler	tool-schema.js
K.acces	pour, courante	workflow-acces.js
K.manifest	normaliser, estV3, valider, enregistrer	manifest-v3.js
K.notebooks / K.agents / K.campagnes	Noyaux de carnet, agents de l'assistant, campagnes	notebook-kernels.js, agent-manager.js, campagnes.js
K.log(source, texte, niveau)	Écrit au journal	panel-journal.js

Les services d'interface

Global	Ce qu'il rend
IsoFindModales	confirmer, informer, demander, popup, deroulant, ouvrir
K.DataGrid	creer, typeDe, TYPES : la grille de données commune

Global	Ce qu'il rend
IsoFindGraphiques	Le thème des graphiques
IsoFindColoration	La coloration syntaxique
IsoFindListes	Les listes déroulantes de toute l'application
IsoFindPalette / IsoFindRaccourcis / IsoFindLayouts	Palette, raccourcis, dispositions nommées

Employez les services communs plutôt que les vôtres. Une grille écrite à la main dans un plugin ne suivra pas le thème, ne saura pas exporter, ne se copiera pas au format tableur, et divergera à la première évolution. C'est aussi vrai des listes déroulantes, des modales et de la coloration.

Ce qui reste hors du noyau

Le gestionnaire de plugins vit dans `window.pluginsHandlers` : installation, signature, capacités connues, chargement du code. Le noyau ne l'appelle jamais directement, il lui délègue par les trois points de la section 16. C'est ce qui permet à l'un d'évoluer sans l'autre, et à un audit de sécurité de se concentrer sur un fichier à la fois.