

Premier plugin

Un tutoriel, du dossier vide au paquet signé

Une heure, dix étapes, un plugin qui lit vos échantillons, en tire un contrôle rapide et l'affiche.

Août 2026. À lire avec le guide du noyau, qui sert de référence.

Ce tutoriel prend la voie confinée, celle recommandée pour tout nouveau plugin.

Sommaire

Avant de commencer

1. Ce qu'on va construire	3
2. Préparer le poste	3

Le plugin

3. Le dossier et le manifeste	5
4. Le premier chargement, et la première erreur	6
5. Lire les échantillons	7
6. Le calcul, et où le mettre	8
7. Un panneau qui montre le résultat	9
8. Une formule pour la feuille	11
9. Se souvenir d'un réglage	12

Publier

10. Empaqueter et signer	14
11. Ce qui casse, et pourquoi	15
12. Et ensuite	16

Avant de commencer

Ce que le plugin fera, et les deux réglages sans lesquels rien ne se chargera.

1. Ce qu'on va construire

Un plugin appelé **Contrôle rapide**. Il lit les échantillons chargés, compte les mesures par élément, repère celles dont l'incertitude sort de l'ordinaire, et affiche le tout dans un panneau. Une formule de feuille de calcul vient en prime.

Le sujet est choisi pour une raison : il touche à tout ce qu'un plugin fait habituellement. Il lit des données, il calcule, il affiche, il pose une entrée de menu, il garde un réglage. Si vous savez faire ces cinq choses, vous savez écrire un plugin, et le reste n'est que du vocabulaire.

Étape	Ce que vous ajoutez	Capacité
5	Lire les échantillons, dire combien il y en a	<code>samples:read, ui:notify</code>
6	Le calcul, dans une fonction à part	<code>aucune</code>
7	Un panneau, ouvert depuis le menu	<code>ui:menu, ui:panel</code>
8	Une formule dans la feuille de calcul	<code>ui:formula</code>
9	Un seuil que le plugin retient	<code>config:read, config:write</code>
-	Vérifier ce que la façade expose vraiment	<code>aucune</code>

On prend la voie confinée. Le plugin tournera dans une iframe à origine opaque, sans accès au DOM de l'application ni à `window.IsoFind`. Il aura en revanche son propre document, celui de l'iframe : on peut y créer des nœuds et dessiner. C'est la voie recommandée, et c'est aussi celle qui force des habitudes qui tiendront : tout passe par une façade, les données arrivent en asynchrone, et rien ne marche par accident.

Le plugin final tient en une centaine de lignes. Vous pouvez le lire en entier à l'étape 10, mais écrivez-le plutôt en avançant : les erreurs de l'étape 4 et de l'étape 11 sont la moitié de ce que ce tutoriel a à vous apprendre.

2. Préparer le poste

Deux réglages, et rien à installer. Pas de npm, pas d'étape de compilation, pas de serveur de développement.

Passer en zone locale

La zone locale est ce qu'on appelle ailleurs le mode développeur. C'est elle qui autorise l'exécution de code non signé, et donc du vôtre tant qu'il n'est pas signé. Elle ouvre aussi l'atelier de plugin, qui lit et écrit des fichiers sur votre machine.

Sur une installation de production, l'entrée en zone locale demande une confirmation qui nomme les choses, et laisse une marque visible. C'est voulu : un mode dangereux dont on oublie qu'il est actif est pire qu'un mode absent. Vous pouvez travailler ainsi sur votre poste de tous les jours, à condition de savoir que la marque est là et pourquoi.

La zone locale n'ouvre pas le catalogue. Un plugin non signé venant du catalogue reste refusé, quelle que soit la zone. C'est la protection qui rend inopérante la phrase « activez le mode développeur et installez ceci » : la seconde moitié ne marchera jamais.

Ouvrir l'atelier

L'atelier est un panneau. Il sait générer un squelette, relire votre dossier, valider le manifeste, recharger à chaud, comparer ce que vous déclarez à ce que le registre contient, et emballer. Il sonde l'horodatage de vos fichiers un peu moins de deux fois par seconde, donc en pratique vous enregistrez dans votre éditeur et le plugin se recharge seul.

Gardez la console du navigateur ouverte à côté. Presque tout ce qui rate dans un plugin y laisse une ligne, et presque rien ne se voit ailleurs.

Le plugin

Sept étapes, chacune laissant quelque chose qui fonctionne.

3. Le dossier et le manifeste

Deux fichiers suffisent : un manifeste et un fichier de code. L'atelier peut les générer, mais écrivez-les à la main cette fois, c'est la seule façon de savoir ce que chaque ligne fait.

```
controle-rapide/  
  manifest.json  
  plugin.js
```

```
// manifest.json  
{  
  "manifest": 3,  
  "id": "controle-rapide",  
  "name": "Contrôle rapide",  
  "version": "0.1.0",  
  "description": "Compte les mesures par élément et signale les incertitudes inhabituelles",  
  
  "engine": { "runtime": "js" },  
  
  "capabilities": ["samples:read", "ui:notify"],  
  
  "activation": ["onStartup"]  
}
```

Champ	Ce qu'il décide
<code>manifest: 3</code>	Sans lui, votre manifeste est lu comme un v2, et rien de ce qui suit n'a de sens.
<code>id</code>	Il préfixe tout ce que vous déclarerez. Choisissez-le une fois : le changer plus tard revient à publier un autre plugin.
<code>engine.runtime</code>	<code>js</code> pour la voie confinée. <code>python</code> pour un worker Pyodide, <code>native</code> pour l'ancienne voie, dépréciée.
<code>capabilities</code>	Ce que l'utilisateur verra à l'installation, et rien de plus. Un appel sans la capacité correspondante lève.
<code>activation</code>	Quand votre code s'exécute. <code>onStartup</code> pour commencer ; l'activation paresseuse viendra à l'étape 12.

Ne demandez pas ce dont vous ne vous servez pas encore

On ajoutera `ui:menu`, `ui:panel`, `ui:formula`, `config:read` et `config:write` quand les étapes correspondantes arriveront. Demander tout dès le départ vous éviterait deux modifications et vous apprendrait la mauvaise habitude : un utilisateur attentif refuse un plugin qui demande l'écriture pour afficher un tableau, et il a raison.

```
// plugin.js, première version
(async function () {
  'use strict';

  console.log('[Contrôle rapide] chargé');
})();
```

C'est tout pour l'instant. Un plugin qui écrit une ligne dans la console est un plugin qui se charge, et c'est exactement ce qu'on veut vérifier avant d'écrire quoi que ce soit d'autre.

4. Le premier chargement, et la première erreur

Ouvrez le dossier depuis l'atelier. Deux choses peuvent arriver, et toutes deux sont instructives.

Si tout va bien

L'atelier affiche le manifeste comme valide, le plugin apparaît dans la liste, et votre ligne s'écrit dans la console. Passez à l'étape 5.

Si le manifeste est refusé

L'atelier liste les erreurs, une par ligne, avec leur raison. Provoquez-en une volontairement, maintenant, pendant que le plugin est vide : ajoutez une commande mal nommée.

```
"contributes": {
  "commands": [{ "id": "analyser",           // SANS le prefixe controle-rapide.
                 "title": { "fr": "Analyser", "en": "Analyse" } }]
}
```

```
[Manifeste] controle-rapide : manifeste invalide
Commande « analyser » doit commencer par « controle-rapide. ».
Contribution jamais activable : analyser (aucune activation ne la vise).
```

Ce refus n'est pas une chicanerie de format

Sans la règle du préfixe, un plugin déclarerait `core.go.dashboard` et détournerait la navigation de l'application. Le nommage est une frontière, pas une convention de propreté.

La seconde ligne dit autre chose : une contribution qu'aucune activation ne vise est du code mort. Ici c'est un effet de bord de la première erreur, mais dans un vrai manifeste c'est presque toujours le signe d'un fichier bricolé à la main, où l'on a ajouté une commande sans ajouter son `onCommand`.

Retirez ce bloc `contributes`, on y reviendra à l'étape 12. Vérifiez que le plugin recharge, et regardez la console : elle doit dire que le manifeste est de nouveau valide.

5. Lire les échantillons

Premier vrai appel. Deux verbes, deux capacités, et une propriété qui surprend tout le monde une fois : la façade est asynchrone.

```
(async function () {
  'use strict';

  const echantillons = await isofind.getSamples();
  await isofind.notify(echantillons.length + ' échantillons chargés', 'info');
})();
```

Tout appel à la façade est asynchrone, sans exception

Votre code tourne dans une iframe à origine opaque. Il n'a ni le DOM de l'application, ni ses variables, ni son jeton. Chaque appel est un message que l'hôte reçoit, arbitre, exécute, puis renvoie. Il n'y a pas de version synchrone, et il ne peut pas y en avoir.

Un `await` oublié ne lève pas : vous recevez une promesse, votre `length` vaut `undefined`, et la notification affiche « undefined échantillons chargés ». C'est la première erreur que tout le monde commet.

Ce que la façade expose, et comment le savoir

C'est le réflexe à prendre avant d'écrire le moindre appel. La façade JS et la façade Python ne portent pas les mêmes verbes, et une version du SDK n'est pas l'autre. Une ligne vous donne la vôtre :

```
console.log(isofind.version, Object.keys(isofind));
// 2.0.0 ["getSamples", "addSample", "runAnalysis", "getConfig", "setConfig",
//       "notify", "log", "registerMenu", "registerPanel", "registerFormula",
//       "openPanel", "storage", "r", "version"]
```

Il n'y a pas de `getAnalyses` côté JS

La voie Python offre un verbe qui aplatit les mesures côté hôte. La façade JS s'arrête à `getSamples()`, et l'appeler donne une erreur sèche, `isofind.getAnalyses is not a function`.

On aplatit donc soi-même, ce qui prend six lignes et met le calcul à l'abri d'un changement de la façade.

```
function mesuresDe(echantillons) {
  var lignes = [];
  echantillons.forEach(function (s) {
    (s.isotope_data || []).forEach(function (iso) {
      lignes.push({
        sample_name: s.name,
        element:      iso.element,
        ratio_value:  iso.ratio_value,
        uncertainty:  iso.uncertainty
      });
    });
  });
  return lignes;
}
```

Deux membres de la liste ci-dessus sont des objets et non des fonctions : `isofind.storage`, le stockage propre au plugin, sous `storage:read` et `storage:write`, et `isofind.r`, le moteur R confiné, sous `compute:r`. Leurs méthodes s'énumèrent de la même façon, `Object.keys(isofind.storage)`.

Retirez `samples:read` du manifeste et rechargez. Faites-le une fois, vraiment : l'appel lève avec `Capacite refusee: samples:read`, et non avec un tableau vide. C'est important à voir de vos yeux, parce que la différence entre « je n'ai pas le droit » et « il n'y a rien » est exactement celle que vos utilisateurs vous rapporteront mal.

6. Le calcul, et où le mettre

Une fonction pure, qui ne connaît ni la façade ni l'affichage. C'est la partie que vous pourrez tester, réécrire et déplacer sans rien casser.

```
// Ne prend que des donnees, ne rend que des donnees.
function controler(mesures, seuil) {
  const parElement = new Map();

  for (const m of mesures) {
    if (!m.element) continue;
    if (!parElement.has(m.element)) {
      parElement.set(m.element, { element: m.element, n: 0, suspectes: [] });
    }
  }
}
```

```

const e = parElement.get(m.element);
e.n += 1;

const u = Number(m.uncertainty);
const v = Number(m.ratio_value);
// Une incertitude relative au-dela du seuil merite un oeil.
if (isFinite(u) && isFinite(v) && v !== 0 && Math.abs(u / v) > seuil) {
    e.suspectes.push({ nom: m.sample_name, ratio: v, incertitude: u });
}
}

return Array.from(parElement.values())
    .sort((a, b) => b.n - a.n);
}

```

Séparez le calcul de la façade, dès le premier jour

Une fonction qui appelle `isofind.getAnalyses()` au milieu de son calcul ne peut être essayée que dans IsoFind, avec des données réelles, en rechargeant le plugin. La même fonction qui prend un tableau en argument s'essaie dans la console du navigateur, en trois secondes, avec les données que vous voulez.

Ce n'est pas une élégance : c'est la différence entre corriger un calcul en une minute et en un quart d'heure, multipliée par le nombre de fois où vous le corrigerez.

Branchez-la, et regardez ce qui sort avant d'écrire la moindre interface.

```

const echantillons = await isofind.getSamples();
const resultat = controler(mesuresDe(echantillons), 0.05);
console.table(resultat);

```

7. Un panneau qui montre le résultat

Deux capacités de plus, `ui:menu` et `ui:panel`, et une contrainte à connaître avant d'écrire une ligne de HTML.

Le rendu reçoit `{ root, id }`, et sa valeur de retour est ignorée. C'est l'erreur qui coûte le plus de temps : rendre une chaîne de HTML ne produit rien, le panneau s'ouvre et reste blanc, et rien dans la console ne le signale. C'est `ctx.root` qu'il faut peupler.

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. On peint donc un état d'attente tout de suite, puis on remplit : le conteneur reste valide entre les deux.

Bonne nouvelle en revanche : l'iframe a son propre document. Vous pouvez créer des nœuds, poser des écouteurs, dessiner. Ce que vous n'avez pas, c'est le document de l'application.

```
isofind.registerPanel('resume', 'Contrôle rapide', function (ctx) {
  var hote = ctx.root;
  hote.innerHTML = '<p style="color:var(--text-muted)">Calcul en cours...</p>';

  isofind.getSamples().then(function (echantillons) {
    var lignes = controler(mesuresDe(echantillons), seuil);
    dernierResultat = lignes;

    if (!lignes.length) {
      // Ne rien trouver est une reponse, et elle se dit.
      hote.innerHTML = '<p>Aucune mesure chargée.</p>';
      return;
    }

    hote.innerHTML = '<table style="width:100%;border-collapse:collapse">'
      + '<tr><th align="left">Élément</th><th align="left">Mesures</th>'
      + '<th align="left">À revoir</th></tr>'
      + lignes.map(function (l) {
        return '<tr><td>' + echapper(l.element) + '</td>'
          + '<td>' + l.n + '</td>'
          + '<td>' + (l.suspectes.length || '') + '</td></tr>';
      }).join('')
      + '</table>';
  }).catch(function (e) {
    // Une capacite refusee LEVE. Le dire dans le panneau plutot que de
    // laisser une surface vide, qui ressemble a un plugin casse.
    hote.innerHTML = '<p style="color:#b91c1c">' + echapper(e.message) + '</p>';
  });
});

isofind.registerMenu('ouvrir', 'Contrôle rapide', function () {
  isofind.openPanel('resume');
});
```

Enregistrez avant d'appeler. Les enregistrements sont synchrones, les appels de données ne le sont pas. Un plugin qui lève sur un appel placé avant ses enregistrements ne laisse aucune surface derrière lui : le panneau n'existe pas, et le menu non plus. Posez vos contributions en premier, faites vos appels ensuite.

Vos couleurs viennent du thème : écrivez `var(--text-primary)`, `var(--border-subtle)`, `var(--surface-1)` comme un composant du cœur. Le pont pose ces variables dans le document du bac à sable et les met à jour quand l'utilisateur change de thème.

Échappez ce que vous insérez. Un nom d'échantillon vient de vos données, donc d'un fichier importé, donc potentiellement de n'importe où. Le HTML est affiché sans scripts, ce qui limite les dégâts, mais un nom contenant un chevron cassera votre tableau et vous chercherez longtemps.

```
function echapper(s) {
  return String(s == null ? '' : s)
    .replace(/&/g, '&amp;').replace(/</g, '&lt;').replace(/>/g, '&gt;');
}
```

Le rappel du menu vit dans votre bac à sable. L'hôte n'installe pas votre fonction, il pose un déclencheur qui vous envoie un message quand on clique. C'est pourquoi votre rappel peut fermer sur vos variables : il n'a jamais quitté votre iframe.

Ajoutez les deux capacités au manifeste, rechargez, et cliquez votre entrée dans la section Plugins.

```
"capabilities": ["samples:read", "ui:notify", "ui:menu", "ui:panel"]
```

La capacité du panneau est vérifiée à l'enregistrement. Pas seulement à l'ouverture. Si vous oubliez `ui:panel`, l'appel lève au chargement, pas au premier clic, et c'est une bonne nouvelle : l'erreur arrive pendant que vous regardez.

8. Une formule pour la feuille

Le geste le plus rentable d'un plugin de géochimie, et celui qui a le plus de contraintes cachées.

```
await isofind.registerFormula('CTRL_N', function (element) {
  const ligne = dernierResultat.find(function (l) { return l.element === element; })
  return ligne ? ligne.n : 0;
}, 'Contrôle rapide', 'Nombre de mesures', 'CTRL_N("Sr")', 'CTRL_N("Sr")');
```

Contrainte

Pourquoi

Le nom doit être en capitales, deux à trente et un caractères, lettres, chiffres et tiret bas

Une formule pose un nom global dans la feuille. La forme imposée évite qu'un plugin écrase par mégarde un objet de service dont le nom ressemble à celui d'une formule. Un nom refusé est signalé en console, et la formule n'existe pas.

Contrainte

Pourquoi

Elle exige `ui:formula`

Une formule n'est pas sans effet de bord : elle apparait dans le sélecteur de fonctions et devient appellable partout dans la feuille.

La première évaluation rend `NaN`

La feuille évalue en synchrone, votre bac à sable répond en asynchrone. Le pont sert la valeur en cache si elle existe, sinon il demande le calcul et rend `NaN`, puis rafraichit la feuille à la réponse.

Le NaN du premier passage n'est pas un bogue

C'est la conséquence directe du confinement : une valeur exacte tout de suite supposerait que votre code tourne dans la page, ce qui est précisément ce qu'on a retiré. La cohérence est différée d'un cycle de recalcul, et exacte ensuite.

Écrivez donc des formules dont le résultat ne dépend pas de l'ordre d'évaluation, et évitez celles qui compteraient sur leur propre valeur précédente.

9. Se souvenir d'un réglage

Le seuil est écrit en dur depuis l'étape 6. Sortons-le, et gardons-le entre deux sessions.

```
let seuil = 0.05;

const garde = await isofind.getConfig('controle-rapide.seuil');
if (typeof garde === 'number') seuil = garde;

// ... et quand l'utilisateur le change
await isofind.setConfig('controle-rapide.seuil', 0.02);
```

Deux capacités de plus, `config:read` et `config:write`. Préfixez vos clefs par votre identifiant de plugin : la configuration est partagée, et une clef nommée `seuil` finira par rencontrer celle de quelqu'un d'autre.

`config` et `storage` ne sont pas la même chose. `getConfig` et `setConfig` touchent à la configuration de l'application, que d'autres lisent aussi. `isofind.storage`, un sous-objet et non des méthodes de premier niveau, est votre stockage à vous : préfixé par votre identifiant, soumis à un quota, et purgé à la désinstallation. C'est ce qu'un utilisateur accorde le plus volontiers, et le bon choix pour un simple seuil. Énumérez ses méthodes avant de l'employer, comme pour le reste de la façade.

Le manifeste complet, à ce stade :

```
"capabilities": [
  "samples:read",
  "ui:notify", "ui:menu", "ui:panel", "ui:formula",
  "config:read", "config:write"
```

Sept capacités pour un plugin qui lit, calcule et affiche. C'est beaucoup, et c'est normal : chacune correspond à une chose visible que le plugin fait. Un utilisateur qui les lit comprend ce qu'il installe, et c'est le but.

Publier

De votre dossier de travail à un paquet que quelqu'un d'autre peut installer.

10. Empaqueter et signer

Trois gestes, dans cet ordre : vérifier, empaqueter, signer.

Vérifier avec l'inspecteur

L'inspecteur de l'atelier confronte ce que votre manifeste déclare à ce que le registre contient réellement. Un écart n'est pas toujours une faute : une contribution déclarée et non implémentée peut être un travail en cours. L'écart inverse est plus grave, une contribution enregistrée sans avoir été déclarée échappe au consentement de l'utilisateur, qui n'aura jamais vu qu'elle existait.

Empaqueter

L'atelier produit l'archive. Vérifiez qu'elle contient le manifeste, votre code et vos éventuelles ressources, et rien de votre dossier de travail.

Signer

Signer n'est pas obligatoire pour un usage personnel : en zone locale, un plugin non signé importé depuis un fichier se charge. Cela le devient dès que quelqu'un d'autre doit l'installer sur une installation de production, ou que vous voulez le proposer au catalogue.

Étape	Ce qui se passe
Créer son identité	Le backend local produit votre paire de clefs et l'écrit chiffrée dans le répertoire de données.
Demander un certificat	Une demande portant votre identité part vers l'autorité IsoFind.
Installer le certificat reçu	Il se range à côté de la clef.
Signer l'archive	La signature est calculée localement.

Votre clef ne quitte jamais votre machine

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

Si vous ne comprenez pas pourquoi votre clef doit rester chez vous, vous finirez par l'envoyer par courriel en croyant rendre service. C'est pour cela que l'interface le dit plutôt que de le cacher.

Une signature vérifiée dispense de la zone locale. Elle ne dispense pas du consentement : vos sept capacités seront toujours présentées à l'installation, et l'utilisateur pourra toujours en décocher.

11. Ce qui casse, et pourquoi

Les erreurs de ce tutoriel, dans l'ordre où on les commet.

Symptôme	Cause	Correction
« undefined échantillons chargés »	Un <code>await</code> oublié : vous manipulez une promesse	Toute la façade est asynchrone, sans exception.
<code>Capacite refusee:</code> <code>samples: read</code>	La capacité n'est pas dans le manifeste, ou l'utilisateur l'a décochée	Ajoutez-la, réinstallez. Un refus lève, il ne rend pas un tableau vide.
Le manifeste est refusé, sans que la raison saute aux yeux	Préfixe d'identifiant, dock inconnu, expression <code>when</code> invalide, contribution jamais activable	L'atelier liste chaque erreur avec sa raison ; lisez-les toutes, la première en entraîne souvent d'autres.
Le tableau du panneau est cassé sur un échantillon	Un nom contenant un chevron, inséré sans échappement	Échappez tout ce qui vient des données.
Le panneau s'ouvre et reste blanc	Le rendu a retourné du HTML au lieu de peupler <code>ctx.root</code> , ou il était déclaré <code>async</code>	Écrire dans <code>ctx.root</code> , et garder le rendu synchrone.
<code>isofind.X is not a function</code>	Le verbe n'existe pas dans cette façade : voie, ou version du SDK	<code>Object.keys(isofind)</code> donne la liste réelle.
Le panneau et le menu ont entièrement disparu	Un appel placé avant les enregistrements a levé, et le reste du fichier n'a jamais été exécuté	Enregistrer d'abord, appeler ensuite.
La formule rend <code>NaN</code> puis la bonne valeur	Feuille synchrone, bac à sable asynchrone	Attendu. La valeur est exacte après un cycle de recalcul.
La formule n'apparaît pas du tout	Nom refusé par la forme imposée, ou <code>ui:formula</code> manquante	Capitales, deux caractères au moins, et regardez la console.
Le plugin ne se recharge plus après une modification	Une erreur de syntaxe dans le manifeste : ce n'est plus du JSON valide	Une virgule en trop, presque toujours.
Rien ne se charge, et la console est muette	Vous n'êtes plus en zone locale, ou le plugin est désactivé dans la liste	Les deux se vérifient en dix secondes, et c'est toujours l'un des deux.

Le réflexe qui fait gagner le plus de temps. Gardez la console ouverte en permanence. Presque tout ce qui rate dans un plugin y laisse une ligne, avec la raison, et presque rien ne se voit ailleurs. Un plugin qui échoue en silence est un plugin dont vous n'avez pas lu la console.

12. Et ensuite

Votre plugin fonctionne et se distribue. Quatre directions, par ordre de difficulté.

L'activation paresseuse

Vous avez déclaré `onStartup`, donc votre code tourne à chaque démarrage, même les jours où personne ne s'en sert. Déclarez plutôt vos contributions dans `contributes`, et visez-les par une activation. Le noyau les enregistre à l'installation, avec des gestionnaires de remplacement ; votre code ne se chargera qu'à la première invocation.

```
"activation": ["onCommand:controle-rapide.ouvrir"],
"contributes": {
  "commands": [{ "id": "controle-rapide.ouvrir",
                  "title": { "fr": "Contrôle rapide", "en": "Quick check" },
                  "category": "Contrôle rapide" }]
}
```

Votre commande apparaît alors dans la palette, dans la recherche universelle et dans le panneau Actions, et peut recevoir un raccourci, sans que votre code ait tourné.

Un point à vérifier avant de vous y engager. Le gestionnaire de remplacement attend que votre code réenregistre la contribution sous le même identifiant. Depuis un bac à sable, cette réinscription passe par le pont, et le guide du noyau signale que ce relais n'a pas été relu ligne à ligne. Essayez-le sur une commande avant d'y convertir tout votre manifeste.

Une tâche de workflow

Votre contrôle devient utile la nuit, dans un workflow armé sur une acquisition. Une tâche déclarée apparaît dans l'éditeur de workflow, dans le calcul des capacités et dans le journal des accès. Le guide des workflows décrit la déclaration, les drapeaux `ecrit` et `outil`, et ce que l'assistant voit ou non.

Le domaine plutôt que l'interface

Les points de contribution du guide du noyau vont bien au delà du menu et du panneau : éditeur pour un format de fichier, ouvrier, widget de tableau de bord, fournisseur de recherche, indicateur de barre d'état, fournisseur de projet. Ce sont tous des registres, et ils s'écrivent tous de la même façon.

La voie native, et pourquoi ne pas la prendre

Elle donne le DOM, le noyau et un panneau interactif, et elle est refusée pour tout nouveau plugin. Si vous en avez besoin, c'est probablement que ce que vous écrivez relève du cœur plutôt que d'un plugin, et la bonne conversation à avoir n'est pas technique.

Ce qu'il faut retenir de ce tutoriel

Tout passe par un registre, tout rend un jeton, tout est asynchrone depuis un bac à sable, et une capacité refusée lève au lieu de rendre du vide. Ces quatre phrases couvrent la moitié des questions qu'on se pose en écrivant un plugin.

L'autre moitié se trouve dans la console.