

First plugin

A tutorial, from an empty folder to a signed package

One hour, ten steps, a plugin that reads your samples, draws a quick check out of them and shows it.

August 2026. Read alongside the kernel guide, which serves as the reference.

This tutorial takes the sandboxed route, the one recommended for any new plugin.

Contents

Before you start

1. What we are building	3
2. Setting up the machine	3

The plugin

3. The folder and the manifest	5
4. First load, and first error	6
5. Reading the samples	7
6. The computation, and where to put it	8
7. A panel showing the result	9
8. A formula for the spreadsheet	11
9. Remembering a setting	12

Publishing

10. Packaging and signing	14
11. What breaks, and why	15
12. Where to go next	16

Before you start

What the plugin will do, and the two settings without which nothing will load.

1. What we are building

A plugin called **Quick check**. It reads the loaded samples, counts measurements per element, flags those whose uncertainty looks out of the ordinary, and shows all of it in a panel. A spreadsheet formula comes as a bonus.

The subject was chosen for a reason: it touches everything a plugin usually does. It reads data, it computes, it displays, it adds a menu entry, it keeps a setting. If you can do those five things you can write a plugin, and the rest is vocabulary.

Step	What you add	Capability
5	Read the samples, say how many there are	<code>samples:read</code> , <code>ui:notify</code>
6	The computation, in a function of its own	<code>none</code>
7	A panel, opened from the menu	<code>ui:menu</code> , <code>ui:panel</code>
8	A formula in the spreadsheet	<code>ui:formula</code>
9	A threshold the plugin remembers	<code>config:read</code> , <code>config:write</code>
-	Check what the facade actually exposes	<code>none</code>

We take the sandboxed route. The plugin will run in an opaque origin iframe, with no access to the application DOM and none to `window.IsoFind`. It will have its own document though, the iframe's: you can create nodes in it and draw. It is the recommended route, and it is also the one that forces habits that will hold: everything goes through a facade, data arrives asynchronously, and nothing works by accident.

The finished plugin is about a hundred lines. You can read it whole at step 10, but write it as you go instead: the errors in step 4 and step 11 are half of what this tutorial has to teach you.

2. Setting up the machine

Two settings, and nothing to install. No npm, no build step, no development server.

Switching to the local zone

The local zone is what is called developer mode elsewhere. It is what allows unsigned code to run, hence yours until it is signed. It also opens the plugin workshop, which reads and writes files on your machine.

On a production install, entering the local zone asks for a confirmation that names things, and leaves a visible mark. That is deliberate: a dangerous mode you forget is on is worse than no mode at all. You can work this way on your everyday machine, as long as you know the mark is there and why.

The local zone does not open the catalogue. An unsigned plugin coming from the catalogue stays refused, whatever the zone. That is the protection which makes "turn on developer mode and install this" useless: the second half will never work.

Opening the workshop

The workshop is a panel. It can scaffold, reread your folder, validate the manifest, hot reload, compare what you declare with what the registry holds, and package. It polls your files' timestamps a little under twice a second, so in practice you save in your editor and the plugin reloads itself.

Keep the browser console open beside it. Almost everything that goes wrong in a plugin leaves a line there, and almost none of it shows anywhere else.

The plugin

Seven steps, each leaving you with something that works.

3. The folder and the manifest

Two files are enough: a manifest and a code file. The workshop can generate them, but write them by hand this time, it is the only way to know what each line does.

```
quick-check/  
  manifest.json  
  plugin.js
```

```
// manifest.json  
{  
  "manifest": 3,  
  "id": "quick-check",  
  "name": "Quick check",  
  "version": "0.1.0",  
  "description": "Counts measurements per element and flags unusual uncertainties.",  
  
  "engine": { "runtime": "js" },  
  
  "capabilities": ["samples:read", "ui:notify"],  
  
  "activation": ["onStartup"]  
}
```

Field	What it decides
<code>manifest: 3</code>	Without it your manifest is read as a v2, and none of what follows means anything.
<code>id</code>	It prefixes everything you will declare. Choose it once: changing it later amounts to publishing a different plugin.
<code>engine.runtime</code>	<code>js</code> for the sandboxed route. <code>python</code> for a Pyodide worker, <code>native</code> for the old route, deprecated.
<code>capabilities</code>	What the user will see at install time, and nothing more. A call without the matching capability throws.
<code>activation</code>	When your code runs. <code>onStartup</code> to begin with; lazy activation comes at step 12.

Do not ask for what you are not using yet

We will add `ui:menu`, `ui:panel`, `ui:formula`, `config:read` and `config:write` when the matching steps arrive. Asking for everything up front would save you two edits and teach you the wrong habit: an attentive user refuses a plugin that asks for write access to display a table, and they are right.

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

  console.log('[Quick check] loaded');
})();
```

That is all for now. A plugin that writes one line to the console is a plugin that loads, and that is exactly what you want to check before writing anything else.

4. First load, and first error

Open the folder from the workshop. Two things can happen, and both are instructive.

If all goes well

The workshop shows the manifest as valid, the plugin appears in the list, and your line shows in the console. Move on to step 5.

If the manifest is refused

The workshop lists the errors, one per line, with their reason. Cause one on purpose right now, while the plugin is still empty: add a badly named command.

```
"contributes": {
  "commands": [{ "id": "analyse",           // WITHOUT the quick-check. prefix
                 "title": { "fr": "Analyser", "en": "Analyse" } }]
}
```

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

This refusal is no formatting quibble

Without the prefix rule, a plugin would declare `core.go.dashboard` and take over the application's navigation. Naming is a boundary, not a tidiness convention.

The second line says something else: a contribution no activation targets is dead code. Here it is a side effect of the first error, but in a real manifest it is nearly always the sign of a hand patched file, where a command was added without its `onCommand`.

Remove that `contributes` block, we will come back to it at step 12. Check that the plugin reloads, and watch the console: it should say the manifest is valid again.

5. Reading the samples

The first real call. Two verbs, two capabilities, and a property that catches everyone out once: the facade is asynchronous.

```
(async function () {  
    'use strict';  
  
    const samples = await isofind.getSamples();  
    await isofind.notify(samples.length + ' samples loaded', 'info');  
})();
```

Every facade call is asynchronous, without exception

Your code runs in an opaque origin iframe. It has neither the application DOM, nor its variables, nor its token. Each call is a message the host receives, arbitrates, runs, then answers. There is no synchronous version, and there cannot be one.

A forgotten `await` does not throw: you get a promise, your `length` is `undefined`, and the notification reads "undefined samples loaded". It is the first mistake everybody makes.

What the facade exposes, and how to find out

This is the reflex to acquire before writing any call at all. The JS facade and the Python facade do not carry the same verbs, and one SDK version is not another. One line gives you yours:

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

There is no `getAnalyses` on the JS side

The Python route offers a verb that flattens measurements host side. The JS facade stops at `getSamples()`, and calling it gives a blunt error, `isofind.getAnalyses is not a function`.

So you flatten it yourself, which takes six lines and puts your computation beyond the reach of a facade change.

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

Two members of the list above are objects rather than functions: `isofind.storage`, the plugin's own storage, under `storage:read` and `storage:write`, and `isofind.r`, the sandboxed R engine, under `compute:r`. Their methods are enumerated the same way, `Object.keys(isofind.storage)`.

Remove `samples:read` from the manifest and reload. Do it once, for real: the call throws with `Capacite refusee: samples:read`, not with an empty array. This is worth seeing with your own eyes, because the difference between "I am not allowed" and "there is nothing" is exactly the one your users will report to you wrongly.

6. The computation, and where to put it

A pure function, knowing neither the facade nor the display. This is the part you will be able to test, rewrite and move without breaking anything.

```
// Takes only data, returns only data.
function check(rows, threshold) {
  const byElement = new Map();

  for (const m of rows) {
    if (!m.element) continue;
    if (!byElement.has(m.element)) {
      byElement.set(m.element, { element: m.element, n: 0, flagged: [] });
    }
  }
}
```

```

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

    const u = Number(m.uncertainty);
    const v = Number(m.ratio_value);
    // A relative uncertainty past the threshold deserves a look.
    if (isFinite(u) && isFinite(v) && v !== 0 && Math.abs(u / v) > threshold) {
        e.flagged.push({ name: m.sample_name, ratio: v, uncertainty: u });
    }
}

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

```

Separate the computation from the facade, from day one

A function calling `isofind.getAnalyses()` in the middle of its computation can only be tried inside IsoFind, with real data, by reloading the plugin. The same function taking an array as an argument can be tried in the browser console, in three seconds, with whatever data you like.

This is not elegance: it is the difference between fixing a computation in a minute and in a quarter of an hour, multiplied by the number of times you will fix it.

Wire it up, and look at what comes out before writing any interface at all.

```

const samples = await isofind.getSamples();
const result = check(rowsFrom(samples), 0.05);
console.table(result);

```

7. A panel showing the result

Two more capabilities, `ui:menu` and `ui:panel`, and one constraint to know before writing a line of HTML.

The render receives `{ root, id }`, and its return value is ignored. This is the mistake that costs the most time: returning an HTML string produces nothing, the panel opens and stays blank, and nothing in the console says so. It is `ctx.root` you have to populate.

The render is synchronous, even when the data is not

Nothing guarantees the host will await a promise. A render declared `async` returns a promise nobody reads, and the panel stays empty. So paint a waiting state straight away, then fill in: the container stays valid in between.

Good news on the other hand: the iframe has its own document. You can create nodes, attach listeners, draw. What you do not have is the application's document.

```
isofind.registerPanel('summary', 'Quick check', function (ctx) {
  var host = ctx.root;
  host.innerHTML = '<p style="color:var(--text-muted)">Working...</p>';

  isofind.getSamples().then(function (samples) {
    var lines = check(rowsFrom(samples), threshold);
    lastResult = lines;

    if (!lines.length) {
      // Finding nothing is an answer, and it gets said.
      host.innerHTML = '<p>No measurement loaded.</p>';
      return;
    }

    host.innerHTML = '<table style="width:100%;border-collapse:collapse">'
      + '<tr><th align="left">Element</th><th align="left">Measurements</th>'
      + '<th align="left">To review</th></tr>'
      + lines.map(function (l) {
        return '<tr><td>' + escape(l.element) + '</td>'
          + '<td>' + l.n + '</td>'
          + '<td>' + (l.flagged.length || '') + '</td></tr>';
      }).join('')
      + '</table>';
  }).catch(function (e) {
    // A refused capability THROWS. Say so in the panel rather than
    // leaving an empty surface, which looks like a broken plugin.
    host.innerHTML = '<p style="color:#b91c1c">' + escape(e.message) + '</p>';
  });
});

isofind.registerMenu('open', 'Quick check', function () {
  isofind.openPanel('summary');
});
```

Register before you call. Registrations are synchronous, data calls are not. A plugin that throws on a call placed before its registrations leaves no surface behind: the panel does not exist, and neither does the menu. Put your contributions first, make your calls afterwards.

Your colours come from the theme: write `var(--text-primary)`, `var(--border-subtle)`, `var(--surface-1)` like a core component. The bridge puts those variables in the sandbox document and updates them when the user changes theme.

Escape whatever you insert. A sample name comes from your data, hence from an imported file, hence potentially from anywhere. The HTML is displayed without scripts, which limits the damage, but a name containing an angle bracket will break your table and you will spend a while looking for it.

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

The menu callback lives in your sandbox. The host does not install your function, it places a trigger that sends you a message when it is clicked. That is why your callback can close over your variables: it never left your iframe.

Add both capabilities to the manifest, reload, and click your entry in the Plugins section.

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

The panel capability is checked at registration. Not only at opening. If you forget `ui:panel`, the call throws at load, not on the first click, and that is good news: the error arrives while you are watching.

8. A formula for the spreadsheet

The most rewarding thing a geochemistry plugin can do, and the one with the most hidden constraints.

```
await isofind.registerFormula('QC_N', function (element) {
  const line = lastResult.find(function (l) { return l.element === element; });
  return line ? line.n : 0;
}, 'Quick check', 'Number of measurements', 'QC_N("Sr")', 'QC_N("Sr")');
```

Constraint

Why

The name must be upper case, two to thirty-one characters, letters, digits and underscore

A formula puts a global name in the sheet. The imposed shape stops a plugin from accidentally overwriting a service object whose name looks like a formula. A refused name is reported in the console, and the formula does not exist.

Constraint

Why

It requires `ui:formula`

A formula is not side effect free: it appears in the function picker and becomes callable anywhere in the sheet.

The first evaluation returns `NaN`

The sheet evaluates synchronously, your sandbox answers asynchronously. The bridge serves the cached value if there is one, otherwise it requests the computation and returns `NaN`, then refreshes the sheet on the answer.

The NaN on the first pass is not a bug

It is the direct consequence of confinement: an exact value straight away would assume your code runs in the page, which is precisely what was taken away. Consistency is one recalculation cycle behind, and exact after that.

So write formulas whose result does not depend on evaluation order, and avoid any that would rely on their own previous value.

9. Remembering a setting

The threshold has been hard coded since step 6. Let us pull it out, and keep it between sessions.

```
let threshold = 0.05;

const kept = await isofind.getConfig('quick-check.threshold');
if (typeof kept === 'number') threshold = kept;

// ... and when the user changes it
await isofind.setConfig('quick-check.threshold', 0.02);
```

Two more capabilities, `config:read` and `config:write`. Prefix your keys with your plugin identifier: the configuration is shared, and a key called `threshold` will eventually meet somebody else's.

`config` and `storage` are not the same thing. `getConfig` and `setConfig` touch the application configuration, which others read too. `isofind.storage`, a sub-object rather than top level methods, is your own storage: prefixed with your identifier, subject to a quota, and purged on uninstall. That is what a user grants most readily, and the right choice for a simple threshold. Enumerate its methods before using it, as with the rest of the facade.

The complete manifest at this point:

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

Seven capabilities for a plugin that reads, computes and displays. That is a lot, and it is normal: each one matches something visible the plugin does. A user who reads them understands what they are installing, and that is the point.

Publishing

From your working folder to a package somebody else can install.

10. Packaging and signing

Three gestures, in this order: check, package, sign.

Check with the inspector

The workshop inspector compares what your manifest declares with what the registry actually holds. A gap is not always a fault: a contribution declared and not implemented may be work in progress. The reverse gap is more serious, a contribution registered without being declared escapes the user's consent, who will never have seen that it existed.

Package

The workshop produces the archive. Check that it contains the manifest, your code and any resources, and nothing else from your working folder.

Sign

Signing is not required for personal use: in the local zone, an unsigned plugin imported from a file will load. It becomes required as soon as somebody else has to install it on a production install, or you want to offer it in the catalogue.

Step	What happens
Create your identity	The local backend produces your key pair and writes it encrypted into the data directory.
Request a certificate	A request carrying your identity goes to the IsoFind authority.
Install the certificate you receive	It is filed next to the key.
Sign the archive	The signature is computed locally.

Your key never leaves your machine

It is the only property that matters. A signing key that exists in two copies proves nothing any more, and revoking a certificate whose author never held the key revokes nothing at all.

If you do not understand why your key must stay with you, you will end up emailing it believing you are being helpful. That is why the interface says so rather than hiding it.

A verified signature removes the need for the local zone. It does not remove consent: your seven capabilities will still be presented at install time, and the user will still be able to untick some.

11. What breaks, and why

The errors from this tutorial, in the order people make them.

Symptom	Cause	Fix
"undefined samples loaded"	A forgotten <code>await</code> : you are handling a promise	The whole facade is asynchronous, without exception.
<code>Capacite refusee: samples:read</code>	The capability is not in the manifest, or the user unticked it	Add it, reinstall. A refusal throws, it does not return an empty array.
The manifest is refused, without an obvious reason	Identifier prefix, unknown dock, invalid <code>when</code> expression, contribution never activable	The workshop lists every error with its reason; read them all, the first often causes the others.
The panel table is broken on one sample	A name containing an angle bracket, inserted unescaped	Escape everything that comes from the data.
The panel opens and stays blank	The render returned HTML instead of populating <code>ctx.root</code> , or it was declared <code>async</code>	Write into <code>ctx.root</code> , and keep the render synchronous.
<code>isofind.X is not a function</code>	The verb does not exist in this facade: wrong route, or wrong SDK version	<code>Object.keys(isofind)</code> gives the real list.
The panel and the menu have disappeared entirely	A call placed before the registrations threw, and the rest of the file never ran	Register first, call afterwards.
The formula returns <code>NaN</code> then the right value	Synchronous sheet, asynchronous sandbox	Expected. The value is exact after one recalculation cycle.
The formula does not appear at all	Name refused by the imposed shape, or <code>ui:formula</code> missing	Upper case, at least two characters, and check the console.
The plugin stops reloading after an edit	A syntax error in the manifest: it is no longer valid JSON	One comma too many, nearly always.
Nothing loads, and the console is silent	You are no longer in the local zone, or the plugin is disabled in the list	Both take ten seconds to check, and it is always one of the two.

The reflex that saves the most time. Keep the console open at all times. Almost everything that goes wrong in a plugin leaves a line there, with its reason, and almost none of it shows anywhere else. A plugin failing in silence is a plugin whose console you have not read.

12. Where to go next

Your plugin works and can be distributed. Four directions, in order of difficulty.

Lazy activation

You declared `onStartup`, so your code runs at every startup, including on days nobody uses it. Declare your contributions in `contributes` instead, and target them with an activation. The kernel registers them at install time, with stand-in handlers; your code will only load on the first invocation.

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

Your command then appears in the palette, in universal search and in the Actions panel, and can take a shortcut, without your code having run.

One thing to check before committing to it. The stand-in handler expects your code to re-register the contribution under the same identifier. From a sandbox, that re-registration goes through the bridge, and the kernel guide notes that this relay has not been read line by line. Try it on one command before converting your whole manifest.

A workflow task

Your check becomes useful at night, inside a workflow armed on an acquisition. A declared task appears in the workflow editor, in the capability computation and in the access log. The workflow guide covers the declaration, the `ecrit` and `outil` flags, and what the assistant sees or does not.

The domain rather than the interface

The contribution points in the kernel guide go well beyond a menu and a panel: an editor for a file format, an opener, a dashboard widget, a search provider, a status bar item, a project provider. They are all registries, and they are all written the same way.

The native route, and why not to take it

It gives you the DOM, the kernel and an interactive panel, and it is refused for any new plugin. If you need it, chances are what you are writing belongs in the core rather than in a plugin, and the right conversation to have is not a technical one.

What to take away from this tutorial

Everything goes through a registry, everything returns a token, everything is asynchronous from a sandbox, and a refused capability throws instead of returning emptiness. Those four sentences cover half the questions you will have while writing a plugin.

The other half is in the console.