

The kernel

Extending, contributing, publishing

Writing a plugin, adding a feature to the core, and understanding what the kernel grants, refuses and sandboxes.

August 2026. Describes contribution kernel 1.0.0 and manifest v3.

The first four sections read straight through. The rest is a catalogue of contribution points.

Contents

The model

1. Three ways to extend IsoFind	3
2. The contribution kernel	4
3. The life of a page	6
4. Manifest v3	7

Contribution points

5. Commands	9
6. Panels	10
7. Editors and the central area	11
8. Menus	12
9. Search, status bar, dashboard	13
10. Background tasks, notifications, modals	14
11. Saving, files, projects	15
12. Connectors, LIMS, workflows	16
13. Settings, context, events	18

Execution and security

14. The three runtimes	20
15. Zone, source, signature	21
16. Capabilities and consent	21
17. What the kernel refuses	24

Writing a plugin

18. From nothing to a loaded plugin	25
19. Signing and distributing	27
20. Four plugins to copy	27
21. Troubleshooting	29
22. API reference	30

The model

What a contribution is, how a page is built, and how a plugin declares what it brings before its code ever runs.

1. Three ways to extend IsoFind

They all go through the same registry, which is what makes the first one useful for learning the other two.

Route	What it is	What it assumes
User script	Code run in the page, registering contributions under the <code>user</code> source. It is also the only place from which an existing command can be wrapped.	The local zone. Nothing to package, nothing to sign.
Plugin	A manifest plus code, installed, consented to, possibly signed. The normal route for anything that will be distributed.	An identifier, declared capabilities, and a sandboxed runtime.
Core contribution	One more file in <code>static/js/kernel/</code> , loaded with the others. Same API as the two above, source <code>core</code> .	Access to the repository. The core is not subject to consent, since it is the application.

One point is worth making straight away: the core of IsoFind is written with the same API as a plugin. The product's panels, commands, menus and dashboard widgets all go through `K.contribute`. This is not a courtesy to plugin authors, it is what keeps the API honest: an extension surface used only by extensions decays quietly, because nobody uses it enough to notice what it lacks.

The rule that holds up everything else

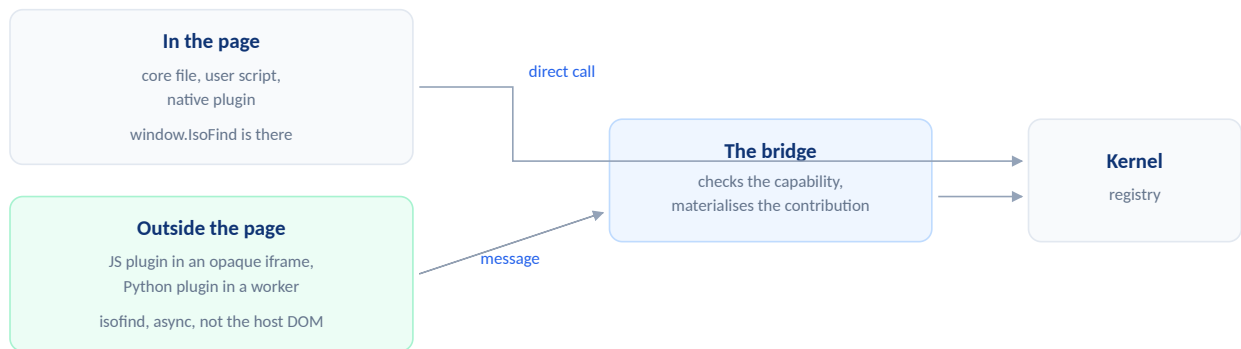
As soon as a feature is hard wired into a page, it escapes the registry. Nothing can list it, wrap it, run it from a console, bind it to a shortcut or show it in the palette. The boundary between core and plugins then reforms on its own, and it always reforms on the same side.

This is the one rule that, held everywhere, prevents that. It explains most of the choices described in this guide.

Where your code runs, and what that changes

This is the most important distinction in the guide, and the one that makes the examples readable. Anything running in the page sees `window.IsoFind` and can call the registries directly. A sandboxed plugin does not see it, and does not need to: it talks through a facade the host arbitrates.

Your code runs	K	What you call
Core file , in the page	yes	Every registry in this guide, directly and synchronously.
User script , in the page	yes	The same, plus <code>K.commands.override</code> .
Native plugin , in the page	yes	The same, plus <code>window.IsoFindPluginAPI</code> . That is what makes it dangerous, and deprecated.
Sandboxed JS plugin , in an iframe	no	The <code>isofind</code> object, installed by the sandbox bridge. The iframe has its OWN document: you can create nodes in it.
Python plugin , in a worker	no	The <code>isofind</code> object as well, but a worker has no document at all. The two facades do not carry the same verbs, see section 16.



Both paths end in the same registry. The second goes through an arbiter, and that is the only difference that matters.

The manifest crosses the boundary, your code does not

Contributions declared in the manifest are registered by the host, which reads that manifest in the page. A sandboxed plugin's command therefore exists in the kernel registry, shows up in the palette and can take a shortcut, without the sandbox ever seeing `K`.

What your code registers from the sandbox goes through the bridge instead: a message is sent, the host checks the capability, then materialises the contribution on its side. Sections 5 to 13 describe the registry as the host sees it; section 16 gives the facade's verbs and the capability each one requires.

The examples in this guide are written page side. They hold as they are for a core file, a user script and a native plugin. From a sandbox, `K` is undefined: the equivalent code calls the facade, whose surface is narrower and whose verbs differ from one runtime to the other. Section 16 gives both tables, section 18 both forms side by side.

What this guide assumes

JavaScript, with no dependency and no build step. The kernel is written in ES5, the core files too, and JS plugins run in an iframe where you are free to use whatever style you like. No npm package is required anywhere.

Domain vocabulary, sample, analysis, campaign, workflow, is not explained here. For workflows and their tasks, the dedicated guide goes further than section 12 of this one.

API names stay in French. `K.menus.declarer`, `K.taches.demarrer`, `entrees`, `sorties`, `executer`: these are the keys the kernel reads, so they are written that way whatever language the interface is in. This guide quotes them verbatim rather than translating them into names that would not work.

2. The contribution kernel

Four notions are enough: the contribution, the source, the token, the registry. Everything else follows from them.

Notion	What it is
Contribution	An object declared to the registry under a type and an identifier: a command, a panel, an editor, a status bar item. The type is not a closed list.
Source	Who contributes: <code>core</code> , <code>user</code> , or a plugin identifier. It is the key to everything else, from naming to release.

Notion	What it is
Token	What every registration returns. Calling it undoes exactly what the registration did.
Registry	One table per type. Display surfaces read the registry and never know their contributors.



A surface never holds a hard coded list. It reads the registry, and finds itself enriched by a plugin without a line of the kernel moving.

The token, and why it is not optional

Every registration returns a token. The pattern already existed in the application, but reimplemented by hand for each kind of contribution: lists of names on `window`, `__pluginId` markers, `data-plugin-id` attributes, swept one by one at teardown. Each variant could miss a sink, and reloading a plugin then left ghost contributions behind.

```

var K = window.IsoFind;

var token = K.commands.register({
  id: 'myplugin.greet',
  source: 'myplugin',
  title: { fr: 'Dire bonjour', en: 'Say hello' },
  handler: function () { K.notifications.info('Hello'); }
});

token.dispose();           // removes this command
K.releaseSource('myplugin'); // removes everything this plugin registered
  
```

One store per source accumulates the tokens, and release happens in reverse order of registration: a late contribution may depend on an earlier one, never the other way round.

Always declare your source. A contribution without `source` is filed under `core`, and `releaseSource` will never free it. It would survive its plugin being unloaded, emptied of its code, and its menu entry would lead nowhere. The recent registries warn in the console when the source is missing; the older ones stay silent.

An identifier belongs to a source

A source may replace its own contribution: that is what lets a plugin declare a command in its manifest and then register its real implementation under the same identifier once activated. A source can never overwrite another's. Naming is not a tidiness convention, it is the boundary that stops a plugin from taking over kernel navigation by declaring `core.go.dashboard`.

A refusal returns an inert token, marked `accepte: false`. Test it where silence would cost you: otherwise you would believe you had registered, and the absence would show up later, elsewhere, with no visible cause.

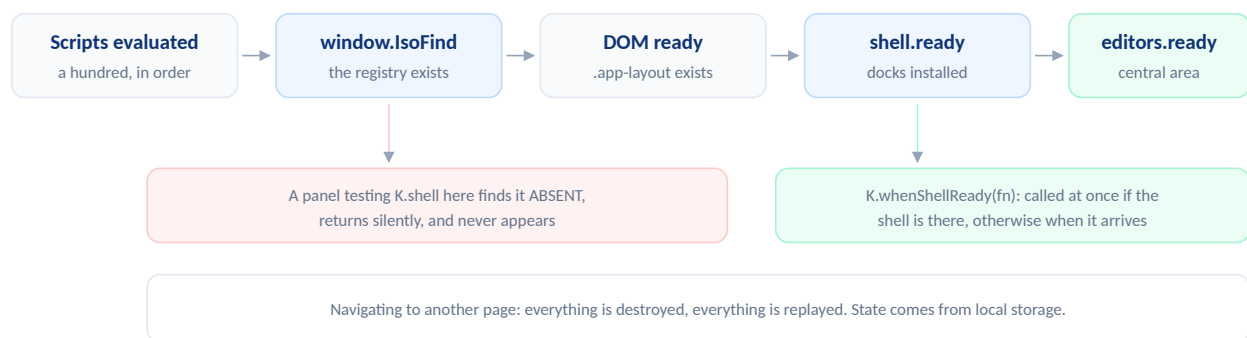
Bilingual titles

Two forms are accepted everywhere. An object `{ fr, en }` when the contributor carries its own strings, which is the case for plugins, since they cannot write into the application dictionary. A translation key when the contributor is the core and the string lives in `translations.js`. `K.resolveTitle` resolves both, and `K.currentLang()` returns the current language.

3. The life of a page

IsoFind is a multi-page application. Every link in the sidebar destroys the document and rebuilds it. That constraint explains the shape of every API that follows.

Nothing survives in memory from one page to the next. What survives is the registration, replayed at load, plus the layout state read back from local storage. Every contribution must therefore be cheap to rebuild, and any panel with heavy state must know how to serialise it rather than keep it in memory.



The hardest failure to see

Scripts are evaluated while the document is parsed; the shell installs only when the DOM is ready, because it needs `.app-layout`. A panel testing `K.shell` as it loads therefore finds it absent, and an early return makes it vanish with no message at all: the file did load, the function did run, and nothing appeared.

`K.whenShellReady(fn)` exists for this. The pattern was copied into every panel, so it was forgettable; putting it in the kernel is the only way to stop forgetting it.

```
(function () {
  'use strict';
  var K = window.IsoFind;
  if (!K) return; // loaded outside the application

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

The same pattern applies to the central area, which sets `editors.ready`, and to anything depending on a surface installed later. The standard context keys are listed in section 13.

Classic script, not a module

The kernel and the core files are classic scripts. A module would be deferred and would run after the hundred global scripts, hence too late for the core to register itself at load time. If you add a file to the core, keep the same shape: an immediately invoked function, `'use strict'`, and a clean exit when the kernel is absent.

4. Manifest v3

A v2 plugin installed itself: to know what it brought, you had to run it. The v3 reverses that, and it is what puts consent back the right way round.

Under the old model, permissions were asked for before anyone knew what the plugin would do, on the strength of a list of abstract capabilities. And every active plugin ran at startup, because that was the only way to discover what it brought. A plugin the user would not touch all day still ran, with all the attack surface that implies.

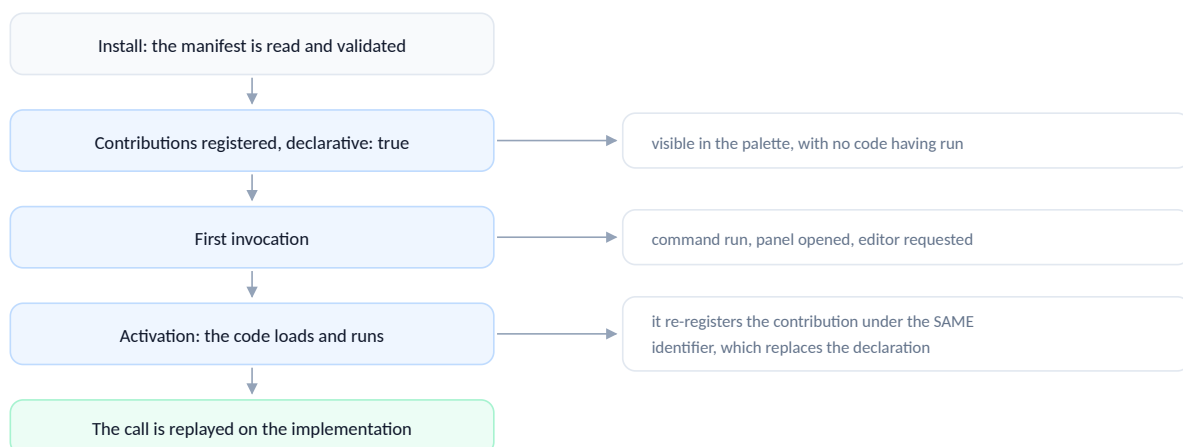
```
{
  "manifest": 3,
  "id": "myplugin",
  "name": "My plugin",
  "version": "1.0.0",

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

  "activation": [                        // when to run the code
    "onCommand:myplugin.analyse",
    "onPanel:myplugin.results"
  ],

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

Lazy activation, and the stand-in handler



A declared panel shows itself immediately, waiting, while the code loads. Loading before opening would give a window that stays empty with no way to tell whether it is working or has failed.

If your plugin declares `onStartup`, its code runs at startup as in v2. Keep that for what really must run without being called, and note it is the only case where validation does not flag a contribution that no activation can reach.

What validation refuses, and why

Refusal	What it closes
An identifier not starting with <code><plugin id>.</code>	The plugin would declare <code>core.go.dashboard</code> and take over kernel navigation.
An invalid <code>when</code> expression	A broken expression evaluates to false, the command vanishes with no message, and the author hunts for a kernel bug.
A dock other than <code>left</code> , <code>right</code> , <code>bottom</code>	The panel would belong nowhere.
A menu entry pointing at an undeclared command	A menu that leads nowhere.
A contribution no activation targets	Dead code, and usually the sign of a hand patched manifest.
The <code>native</code> runtime without the <code>runtime:native</code> capability	Exactly the manifest an attacker would write to obtain full trust without asking for it.

Context expressions

The `when` field of a declared command is evaluated by the kernel without running the plugin: that is what allows an irrelevant command to be hidden without ever loading its code. The grammar is deliberately poor, and there is no `eval`: keys, negation, conjunction, disjunction, equality. A manifest expression is data supplied by a third party, and evaluating it would be exactly the hole we spend our time closing elsewhere.

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

A v2 manifest is still accepted. It is exactly a v3 with `onStartup` and zero declared contributions. That is no coincidence: it is the definition of the model being replaced, and the fact that it expresses itself effortlessly in the new vocabulary is what shows the new one is more general, not merely different.

Contribution points

The catalogue: what each registry expects, what it returns, and the trap in each one.

5. Commands

The single dispatch path. Palette, shortcut, menu, toolbar, console, script, assistant: all of them converge here. It is what makes the application drivable, and it is the only place where capability and zone are checked, hence the only place to audit.

```
K.commands.register({
  id:      'myplugin.analyse',      // prefixed by the source, required
  source:  'myplugin',
  title:   { fr: 'Analyser', en: 'Analyse' },
  category: 'myplugin',           // groups it in the palette
  capability: 'samples:read',      // checked on every call
  zone:    null,                  // 'local' to restrict to the local zone
  when:    function (ctx) { return !!ctx.get('editor.active'); },
  keybinding: 'ctrl+alt+a',
  quickAccess: true,              // offered in quick access
  handler:  function (args, ctx) { return Promise.resolve(); }
});
```

Call

Effect

<code>K.commands.execute(id, args)</code>	Returns a promise. Checks zone, capability and <code>when</code> , wakes the source if needed, then calls the handler.
<code>K.commands.isAvailable(id)</code>	The same rule without running. This is what views use.
<code>K.commands.list(opts)</code>	Available commands, with resolved labels. Options: <code>quickAccess</code> , <code>source</code> , <code>category</code> , <code>available: false</code> to see everything.
<code>K.commands.override(id, fn)</code>	Wraps an existing command. Local zone, <code>user</code> source.
<code>K.keybindings.bind(combo, id)</code>	Binds a key. A table separate from commands, so remapping acts on the binding.

Three events accompany every call: `command.before`, `command.after` and `command.error`. They carry the identifier, the arguments, then the result or the error. That is how you instrument without touching the commands.

Overriding

The pattern is the one adopted for the IsoSupply CRM extensions: you wrap without touching the original, and releasing the token restores exactly the previous state. It is the primitive all advanced customisation rests on.

```
var token = K.commands.override('core.enregistrer', function (next, args, ctx) {
  if (!confirmed()) return Promise.resolve(); // we refuse, core untouched
  return next(args).then(function (r) {
    K.log('user', 'Save went through the override');
    return r;
  });
});
```

Overriding is not offered to plugins. It lives under the `user` source and assumes the local zone. A distributed plugin able to wrap core commands could do anything: that would be the native route under another name.

Shortcuts

A shortcut without a modifier does not fire while you type. Binding `n` to a command would otherwise make it impossible to type the letter `n` in a field, and the symptom would be blamed on the keyboard, never on the shortcut. With a modifier it goes through: `ctrl+f` in an editor must reach the application, which is the very point of a modifier.

A conflict is resolved by context: the first available binding wins, and a binding registered later takes precedence.

6. Panels

A panel lives in a dock, left, right or bottom. It can then be moved, pinned, opened as a central tab, detached into a window or floated, without its author planning for any of it.

```
K.shell.registerPanel({
  id:      'myplugin.results',
  source:  'myplugin',
  title:   { fr: 'Résultats', en: 'Results' },
  icon:    '<svg ...>',           // inline SVG, no file
  dock:    'right',              // left | right | bottom
  factory: function (ctx) {
    ctx.root.innerHTML = '<div class="mp-body"></div>';

    // The state from the previous page, if there was one.
    var state = ctx.state || { filter: '' };

    function keep() { ctx.saveState(state); }

    return {
      dispose: function () { /* listeners, timers, requests */ }
    };
  }
});
```

ctx	Content
ctx.panelId	The panel identifier
ctx.root	The node to build into. Empty on call.
ctx.state	Whatever <code>saveState</code> filed, or <code>null</code>
ctx.saveState(obj)	Files a serialisable state. It is through this call, and only this call, that a panel survives navigation.
ctx.close()	Closes the panel from inside

The factory returns a controller, of which only `dispose` is read. A panel that sets listeners on `window`, a timer or a bus subscription must undo them here, otherwise they outlive closing and stack up on every reopen.

Every panel becomes toggleable without you thinking about it. Registration automatically adds a `<id>.toggle` command, so the panel appears in the palette, in universal search and in the Panels menu. That is the kind of uniformity the registry gives away for free.

The four possible hosts

Call	Effect
<code>K.shell.open / close / toggle</code>	In its dock
<code>K.shell.move(id, dock)</code>	Moves dock
<code>K.shell.placer(id, host)</code>	Frees the other three hosts then installs in the one asked for: <code>dock</code> , <code>flottant</code> , <code>onglet</code> , <code>fenetre</code>
<code>K.zone.ouvrirPanneau(id)</code>	As a tab in the central area
<code>K.rail.epingler(...)</code>	On a side strip
<code>K.shell.signaler(id)</code>	Attention dot, unless the panel is already in view

Not every panel travels equally well. A panel whose factory builds into `ctx.root` moves through all four hosts without trouble. An adapter panel, which relocates a node built elsewhere, cannot: the node can only be in one place at a time. If you want your panel to travel, build into the root you are given.

The dot, and when it is warranted

A question that takes thirty seconds, you go and read a file, and you never come back: the answer is there and nobody knows.

`K.shell.signaler(id)` says so once, and the mark clears on opening. It does not show if the panel is already visible: blinking to announce what the user is currently reading is the surest way to teach them to ignore the dot.

Detached windows

A detached window is an entirely separate JavaScript context. The kernel bus does not cross that boundary: an emitter and its subscriber must live in the same window to see each other. A bridge carries a handful of explicitly listed subjects across, each with a direction. Relaying everything would be a mistake: most bus subjects describe the internal state of one window, and relaying them would make each window believe the registry next door is its own.

If your panel needs to talk to another across a window, the line is added in `pont-fenetres.js`, and that is a deliberate act: every line is a coupling between two windows.

7. Editors and the central area

The central area is a tabbed area. The first tab is the page itself; the others are editors open on some content: a CSV, a database table, an ISO document, whatever your plugin can display.

Treating the page as one tab among others is the move that makes the rest coherent. Stacking editors on top of the page as an overlay would have given two notions of central display, and every question of focus, shortcuts and saving would arise twice.

```
K.editors.register({
  kind: 'myplugin.spectrum',      // prefixed by the source
  source: 'myplugin',
  title: { fr: 'Spectre', en: 'Spectrum' },
  icon: '<svg ...>',

  // Does this reference concern me?
  match: function (ref) { return /\s.spç$/i.test(ref.chemin || ''); },
  // The tab label
  label: function (ref) { return ref.nom || 'Spectrum'; },
});
```

```

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

K.editors.open({ kind: 'myplugin.spectrum', chemin: '/data/s01.spc' });

```

A reference is resolvable, or it is not

That is what decides whether it persists. A database table reopens from its name: it survives navigation. A dropped file has no path the application can read back; it survives only if its content fits in the serialised state, and the editor decides that for itself.

The registry does not pretend to restore what it cannot find again. Beyond roughly four hundred thousand characters a state is no longer persisted: the local storage quota would otherwise take down the whole layout, including that of other panels.

Call	Effect
<code>K.editors.open(ref)</code>	Opens or activates the tab
<code>K.editors.close / activate</code>	Closes, brings to front
<code>K.editors.detach / reattach</code>	To a dock, and back
<code>K.editors.isOpen(ref)</code>	A boolean
<code>K.zone.scinder / plein / epingler</code>	Split panes, full screen, pinned tab

`ctrl+w` closes the active tab, and the matching command (`core.editor.close`) is only available when the `editor.active` context key is set. It is a good example of the pattern: the shortcut is not guarded by a condition written into a listener, it is guarded by the availability of the command.

8. Menus

Two separate registries: the menu bar at the top, and context menus attached to object types.

The context menu

The target is an object type, not a CSS selector. `db.ligne` rather than `.db-row`: a selector would tie menus to the DOM structure of a panel, renaming a class would break a plugin, and the plugin would have no way to know. The type is a contract, the class is a detail.

```

K.menus.declarer({
  id: 'myplugin.db.simulate',
  source: 'myplugin',
  cible: 'db.ligne', // the object TYPE
  ordre: 200, // low numbers first
  items: function (ctx) {
    return [{
      id: 'simulate',
      titre: { fr: 'Simuler dans PHREEQC', en: 'Simulate in PHREEQC' },
      groupe: 'analysis', // groups are separated by a rule
      ecrit: false,
      actif: function () { return !!ctx.ligne; },
    ]
  }
});

```

```

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

```

An item may instead carry `commande: 'myplugin.analyse'`, and its title and shortcut then come from the command registry: one definition, and the menu follows a rename on its own.

What is unavailable is greyed out, never hidden

Hiding an item according to state makes people believe it does not exist, and they go looking elsewhere for a function they had at hand. An inactive item stays visible, with its reason.

On software whose provenance is signed, "Pull now" and "Copy path" do not commit to the same thing. Items marked `ecrit` are separated from the rest and pushed to the bottom of the menu, and `danger` turns them red. This is not decoration: it is what keeps a quick gesture, made without reading, a safe one. `confirmation` adds a question before acting.

Submenus go one level deep, and that is enough. `K.menus.ouvrir(target, ctx, x, y)` allows imperative opening, for a panel whose items are only known at the moment of the click, such as the file explorer whose list of openers depends on the file.

The menu bar

```

K.menubar.register({
  id: 'myplugin.menu',
  title: { fr: 'Mon plugin', en: 'My plugin' },
  ordre: 60, // File 5, Go 8, Panels 10, Journal 20
  items: function () { return [ /* same items */ ]; }
});

```

9. Search, status bar, dashboard

Three surfaces with nothing hard coded, which a plugin enriches the same way it enriches a menu.

Universal search

It opens on a double press of Shift. A provider receives the query and returns results; the kernel groups them, orders them and handles keyboard navigation.

```

K.recherche.enregistrer({
  id: 'myplugin.spectra',
  source: 'myplugin',
  title: { fr: 'Spectres', en: 'Spectra' },
  order: 120, // low numbers first
  icone: '<svg ...>',
  chercher: function (query) {
    // returns an array, or a promise of one
    return [{ id: 's1', libelle: 'Spectrum 1', sousTitre: 's-01',
      ouvrir: function () { /* ... */ } }];
  }
});

```

The status bar

```
K.statut.enregistrer({
  id:      'myplugin.state',
  source:  'myplugin',
  title:   { fr: 'Moteur', en: 'Engine' },
  side:    'right',           // left | right
  order:   100,
  defaultOn: true,           // visible by default
  render:  function (host) { host.textContent = 'ready'; },
  onClick: function () { K.shell.toggle('myplugin.results'); }
});
```

An item writes its labels when it is built. The registry rebuilds it on a language change, so do not keep state there that you could not rebuild. The user can hide any item, including yours, and `K.statut.disponibles()` returns the real list, plugins included, for the preferences screen.

The dashboard

The dashboard has stopped being a page written in advance and become a surface. A hard coded dashboard can only show what its author foresaw: it will never show a plugin's last simulation, since the page does not know the plugin.

```
K.dashboard.registerWidget({
  id:      'myplugin.recent',
  source:  'myplugin',
  order:   50,
  factory: function (host) {
    host.innerHTML = '<h3>Recent simulations</h3>';
    return { dispose: function () {} };
  }
});
```

What a dashboard has to answer. One question: "where was I". Not "how many samples do I have", which is a statistic and gets looked at once. Picking work back up is why people open software in the morning, and it is the only thing a home page can do better than a menu.

`K.dashboard.remonter()` repaints the dashboard where it is already mounted. Without it, a user who launches an export and comes back to the dashboard would see nothing, and would conclude nothing had been recorded.

10. Background tasks, notifications, modals

Three ways to speak to the user, and they are not interchangeable. A long operation declares a task, a completed fact produces a notification, a question blocks and waits.

Background tasks

An import takes two minutes, a simulation longer, a text generation drags on. Without a registry each one reports progress its own way, or not at all, and the user cannot tell whether the application is thinking or stuck.

```
var t = K.taches.demarrer({
  titre:      'PHREEQC simulation',
  sujet:      's-01',
  source:     'myplugin',
  determinee: true,           // false: indeterminate bar
});
```

```

    onAnnuler: function () { engine.stop(); }
  });

  t.progresser(0.34, '340 rows of 1000');
  t.message('Equilibrating');
  t.reussir('12 species computed');
  // or t.echouer(error), or t.annuler()

```

Two kinds of progress, and you need both

Some tasks know how far along they are. Others do not: a generation does not know when it will finish. The first gets a bar that fills, the second an indeterminate bar. Forcing a percentage on the second would display an invented figure, worse than no figure at all, since you would believe you could estimate the end.

A finished task does not vanish at once. A task that disappears the millisecond it ends does not let you see that it succeeded, and above all that it failed. Failures stay until acknowledged: a failed import that clears itself is an import you will never know failed.

Notifications

```

K.notifications.info('Export finished');
K.notifications.attention('3 files skipped');
K.notifications.erreur('Source unreachable');
window.showNotification(message, 'error'); // historical form, equivalent

```

A notification keeps nothing. For what must survive the night there are workflow alerts, which the Workflows panel presents at the next startup. The notification system relays them in a single line rather than stacking forty: forty banners would be forty times less read than one.

Shared floating surfaces

`window.IsoFindModales` carries the backdrop, the modal, the buttons, the popups and the dropdowns the whole application uses. Three forms cover nearly every case, and they return promises.

```

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

IsoFindModales.informer({ message: 'Done' }); // what alert() did
IsoFindModales.demander({ titre: 'Batch name' }); // what prompt() did

```

`window.confirm` and `window.prompt` are broken under Tauri. They either do not appear or freeze the window. The whole application has moved to these promise based modals; a plugin calling the native functions would see its code stop with no message.

11. Saving, files, projects

Declaring what is unsaved

The kernel keeps a single notion of unsaved state, which every editor, panel and console registers with. That is what lets

`core.enregistrer` save everything, and a close warn about the rest.

```
K.enregistrement.declarer({
  id:      'myplugin.spectrum',
  source:   'myplugin',
  title:    { fr: 'Spectre', en: 'Spectrum' },
  modifie:  function () { return dirty; },
  enregistrer: function () { return write(); } // a promise is accepted
});
```

File openers

What happens when a file is clicked in the explorer. The temptation would be to write a `switch` on the extension inside the panel: that would be the ninth hard coded list, and the ninth time a plugin could add nothing to it.

```
K.openers.declare({
  id:      'myplugin.spectrum',
  source:   'myplugin',
  title:    { fr: 'Ouvrir comme spectre', en: 'Open as spectrum' },
  extensions: ['spc', 'sp'], // or accepte: function (path) -> bool
  default:  true,           // offered first
  ouvrir:   function (path, name) { return K.editors.open({ /* ... */ }); }
});
```

Several openers may accept the same extension, and that is intended. A `.csv` can be opened in the spreadsheet, attached to the assistant, or ingested by the LIMS: these are not variants of one action, they are three different intentions, and only the user knows which is theirs. Every opener that accepts the file is offered, the first being the default. Core defaults come first: a plugin loaded late must not steal the behaviour of a core format, or the user would see their double click change meaning after an install without understanding why.

Project providers

A project tree is not written into the panel. Each family of objects registers itself, and a plugin can add its own.

```
K.projet.fournir({
  id:      'myplugin.spectra',
  titre:   { fr: 'Spectres', en: 'Spectra' },
  ordre:  40,
  compter: function (project) { return 12; },
  lister:  function (project) { return [{ id: 's1', libelle: 'Spectrum 1' }]; },
  ouvrir:  function (item) { /* double click on a leaf */ },
  rattachable: true
});
```

The tree follows the declared order, not the alphabet: "Samples" before "Analyses" because the latter follow from the former, and the order tells that dependency. Sorting alphabetically would scramble the logic of the work.

12. Connectors, LIMS, workflows

Reasoning connectors

A connector links IsoFind to a local engine. The abstraction exists so that IsoFind depends on none of them, and a plugin may declare one.

The connector does not make the network call

If it performed its own `fetch`, it would call whatever it liked, and IsoFind's sovereignty would be no more than a promise kept by someone else's code. The connector declares an address and formats its requests; the kernel does the transport, and refuses any address that is not the local machine.

The difference is not rhetorical: a promise is verified by rereading all the code, a constraint is verified by rereading one file.

```
K.connectors.declare({
  id:      'myplugin.engine',
  title:   { fr: 'Mon moteur', en: 'My engine' },
  base:    'http://127.0.0.1:11434', // refused if not local
  dialoguer: function (messages, options) { /* formats, returns a request */ }
});
```

Three traps are closed by the transport, and none is theoretical. Redirects: a local server can answer 302 towards the outside, so the transport refuses to follow. Names that resolve elsewhere: only local address literals and `localhost` are admitted. Disguised addresses: `2130706433` and `127.1` are normalised by `new URL()`, but `127.0.0.1.evil.com` stays a domain name, which is why the comparison is on the whole host, never on its prefix.

LIMS connectors

Same principle for acquisition: a connector declares how to read an instrument source, and the registry checks the transport lock at the last moment. A plugin writing a direct `fetch` to the acquisition route would bypass the rule forbidding network connectors from leaving the machine. `K.lims` carries `tirer`, `tester`, `transportDe` and `verrou`.

Workflow tasks

A task declared by a plugin appears in the workflow editor, in the capability computation, in the access log, and, if it allows it, among the assistant's tools. Nothing to wire: the registry is enough.

```
K.workflows.declarerTache({
  id:      'myplugin.file',
  source:  'myplugin',
  ecrit:   true, // missing means « writes »: the careful default
  outil:   false, // invisible to the model
  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 }); }
});
```

The `ecrit` flag decides visibility in conversation mode and skipping in a dry run; `outil: false` withholds the task from the model in every mode. The workflow guide covers both, along with capabilities, the access log and the full catalogue of core tasks.

The assistant's tools

`K.tools` turns tasks into tool schemas for the model: `schemas('agent')` or `schemas('conversation')`, `appeler(name, args)` to run one, `lisible` for display. A plugin normally has nothing to do here: it declares a task, and the assistant sees it or not according to the two flags.

13. Settings, context, events

Layered settings

Four layers, from weakest to strongest: `default`, `edition`, `user`, `project`. The effective value is computed on read, never frozen on write; otherwise an override installed late would have no effect on anything that had already read.

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

K.settings.get('myplugin.threshold');
K.settings.set('myplugin.threshold', 5);           // user layer, persisted
K.settings.onChange('myplugin.threshold', function (v) { /* ... */ });
```

Context keys

Context mirrors the current state against which availability predicates are evaluated. It is what lets a command exist only where it makes sense, without the trigger having to know.

Key	Set by
<code>shell.ready</code>	The docks are installed
<code>editors.ready</code> , <code>zone.ready</code>	The central area is ready
<code>actions.ready</code>	The Actions panel is mounted
<code>editor.active</code>	An editor tab has focus
<code>core.page</code>	The current application page
<code>core.hasSidebar</code>	The page carries a sidebar
<code>kernel.zone</code>	<code>distribution</code> or <code>local</code>
<code>dev</code> , <code>dev.build</code>	Developer mode, development build
<code>plugins.unconfined</code>	An unconfined plugin is loaded
<code>plugin.<id>.active</code>	That plugin has been activated
<code>fichier.ouvrir</code> , <code>isof.importer</code> , <code>isof.verifier</code>	Domain hook points

Set your own with `K.context.set('myplugin.ready', true)`, and read them in a `when`. A change notifies every listener, which lets a surface repaint without polling.

The bus, and blocking hooks

Two regimes. The plain notification, after the fact, not cancellable. And the blocking hook, before the fact, which a subscriber may refuse: that is the clean injection point for override scripts, which can then refuse a save or an engine launch without touching the core.

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

K.events.emit('myplugin.compute.done', { n: 12 });
```

```
K.events.emitBlocking('myplugin.before.write', payload)
  .then(function (ok) { if (ok) write(); }); // false = someone refused
```

A subscriber that throws does not break the chain: the error is logged and the others carry on. The kernel's most useful subjects are `registry.changed`, `setting.changed`, `command.before`, `command.after`, `command.error`, plus those of the registries: `dashboard.changed`, `openers.changed`, `connectors.changed`, `projet.changed`.

The journal

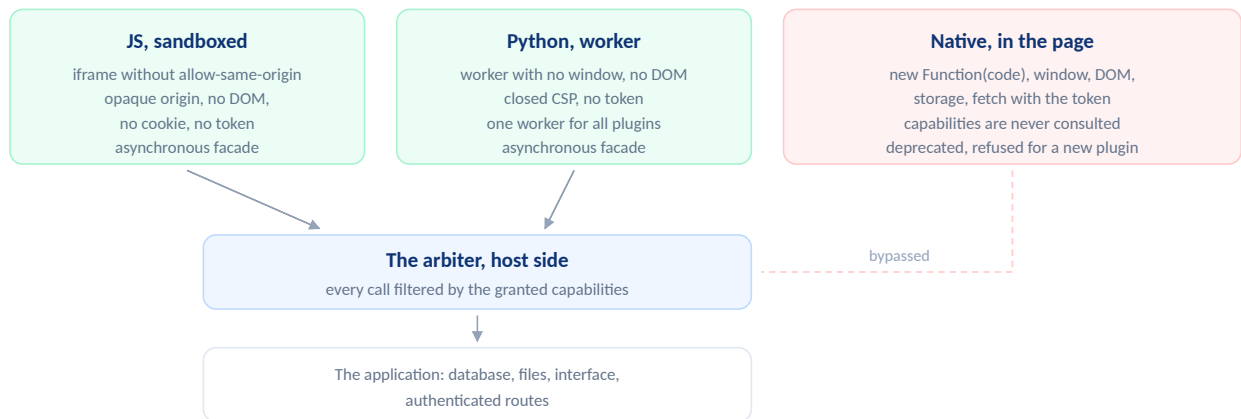
`K.log(source, text, level)` writes to the Journal panel, which is the record of intent: what the user set out to do. Not to be confused with the change registry, which says what changed in the data, durably and under signature. Both are necessary, and a traceability tool showing only the first would be lying by omission.

Execution and security

Where a plugin's code runs, what decides whether it loads, and what the kernel refuses whatever it is asked.

14. The three runtimes

Three routes coexist, and they do not offer the same confinement. The difference is not a language preference, it is what decides what a plugin can reach.



The JS route

It is already confined, and well. A sandboxed iframe without `allow-same-origin`, hence an opaque origin: no access to the host DOM, no cookie, no token, and a call surface filtered by capabilities. This is the recommended route for any new plugin.

The Python route

It was not, and that was a hole we had left ourselves: the worker existed, worked, and nobody called it. A Python plugin therefore had `window`, the DOM and the API token. Confinement was written and unplugged, which is worse than absent, since we believed we had it.

The worker is now the only path. Accepted consequence: the Python facade has become asynchronous. Synchronous calls were only possible because nothing separated Python from the page.

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

    isofind.register_formula("delta_norm", compute,
                             group="My plugin", syntax="delta_norm(x)")
    isofind.register_panel("results", "Results", render)
    isofind.register_menu("analyse", "Analyse", action)
```

The facade offers `get_samples`, `get_samples_df`, `get_analyses`, `add_sample`, `run_analysis`, `notify`, `get_config`, `set_config`, `open_panel`, and the three registrations above. Each becomes a message the host arbitrates, and that is where, and only where, capabilities are checked.

The native route cannot be confined. It runs `new Function(code)` in the page, with `window`, the DOM, storage and the authenticated `fetch`. No guard changes that: governing total access is still total access. It still works, it warns, and it is refused for any new plugin. Removing it outright would break the existing consoles, which would make security responsible for a regression, and that is the surest way to get security abandoned.

15. Zone, source, signature

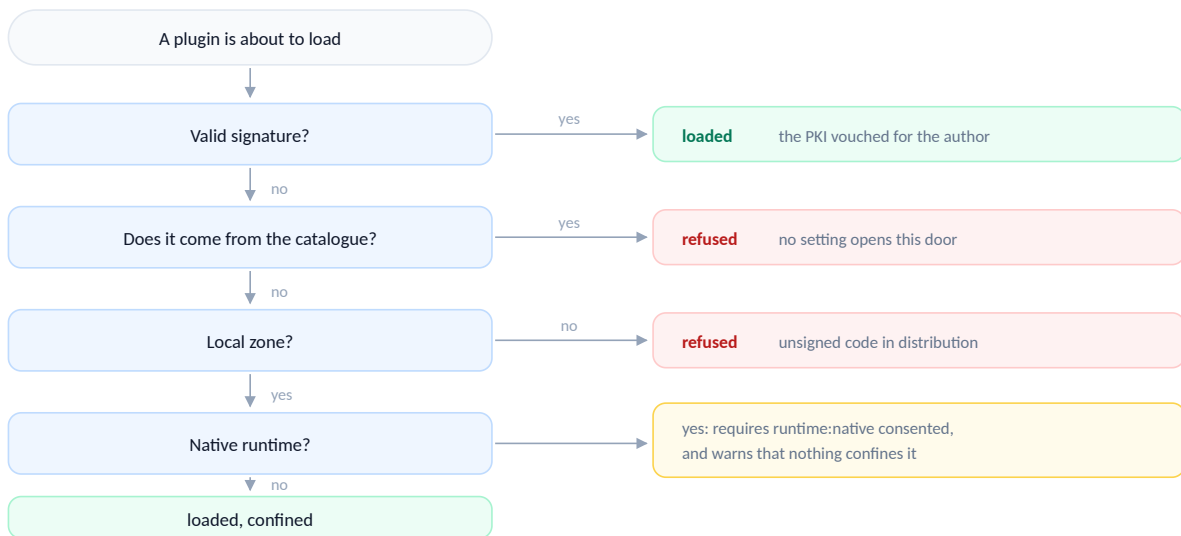
Two independent axes, and confusing them is what makes a checkbox dangerous.

The zone controls the execution of unsigned code. The source says where the plugin came from. An unsigned plugin imported from a local file, in the local zone, is allowed. An unsigned plugin coming from the catalogue is refused, whatever the zone.

Why the second invariant matters most

You can talk someone into ticking a box. You cannot make an unsigned remote plugin installable, because no setting opens that door. The attack scenario, which is always "turn on developer mode and install this from the catalogue", loses its second half.

The check happens at load, not at install. A plugin record lives in local storage, which is editable; a check at load runs at every startup, on what is really about to execute, and that is the only moment that counts.



Entering the local zone on a distribution build is still a serious act. It is not forbidden, and forbidding it would be a mistake: a lab technician writing their own plugin on a production install is exactly the audience, and refusing them sideloading would empty the product of its point. The risk is not the developer, it is the manipulated user. Entering therefore requires a confirmation that names things, and leaves a permanent, visible mark: a dangerous mode you forget is on is worse than no mode at all.

16. Capabilities and consent

The kernel decides nothing here, it asks. Policy stays with the host, which alone knows the PKI, the consent granted at install and developer mode. The kernel merely refuses a dispatch when the host says no, which guarantees there is no side path.

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

A command carrying a `capability` is refused if its source has not been granted it. The core is not subject to consent: it is the application. Every other source must prove its capability, and a missing provider counts as refusal, never as authorisation.

The vocabulary, in full

Thirteen capabilities, and not one more: a capability outside this list is ignored at install time, with a console warning. It opens nothing, and filing it with the others would suggest otherwise.

Capability	What the user reads
<code>samples:read</code>	Read samples: access loaded sample data
<code>samples:write</code>	Add samples: create new samples in the project
<code>analysis:run</code>	Run analyses: trigger the analysis engine
<code>config:read</code>	Read configuration: view application settings
<code>config:write</code>	Modify configuration: save settings
<code>ui:notify</code>	Show notifications: emit on-screen messages
<code>ui:menu</code>	Add menu entries: extend application menus
<code>ui:panel</code>	Show panels: open a dedicated interface
<code>ui:formula</code>	Add formulas: extend the spreadsheet functions
<code>storage:read</code>	Read its storage: read back its own settings
<code>storage:write</code>	Write its storage: keep its own settings
<code>compute:r</code>	Run R code: compute in an isolated R engine, with no data access
<code>network</code>	Access the network: open outbound connections

At install time the consent screen shows these labels, boxes ticked, and grants only what stays ticked. In air-gap mode, `network` is greyed out and refused. Two kernel capabilities live apart because they are not asked for the same way: `runtime:native`, which must be declared and signed for the native route to be admitted, and the workflow capabilities, which are granted when a flow is imported rather than when a plugin is installed.

The facades, and why there are two

A sandboxed plugin does not call the registries, it calls an object named `isofind` that its runtime installs for it. That object is not the same on each route, and confusing them costs a debugging session: the host backend table serves the Python and native routes, the bridge table serves the JS route.

Verb	Capability	JS	Python
<code>getSamples()</code>	<code>samples:read</code>	yes	yes
<code>getAnalyses()</code>	<code>samples:read</code>	no	yes
<code>addSample(object)</code>	<code>samples:write</code>	yes	yes
<code>runAnalysis(params)</code>	<code>analysis:run</code>	yes	yes
<code>getConfig / setConfig</code>	<code>config:read</code> , <code>config:write</code>	yes	yes
<code>notify(message, type)</code>	<code>ui:notify</code>	yes	yes

Verb	Capability	JS	Python
<code>registerMenu(id, label, callback)</code>	<code>ui:menu</code>	yes	yes
<code>registerPanel(id, label, render)</code>	<code>ui:panel</code>	yes	yes
<code>registerFormula(name, fn, ...)</code>	<code>ui:formula</code>	yes	yes
<code>openPanel(id)</code>	<code>ui:panel</code>	yes	yes
<code>log(...)</code>	<code>none</code>	yes	yes
<code>storage</code>	<code>storage:read, storage:write</code>	yes, sub-object	no
<code>r</code>	<code>compute:r</code>	yes, sub-object	no
<code>version</code>	<code>none</code>	string, 2.0.0	-

The absence of `getAnalyses` on the JS side is not an oversight to work around

A JS plugin that wants flat measurements takes them from `getSamples()` and flattens `isotope_data` itself: each measurement carries `element`, `ratio_value` and `uncertainty`, and the sample context is reattached at the same time.

The general rule: before writing a call, enumerate what the facade actually exposes in your runtime. The table above describes one version, and `Object.keys(isofind)` describes yours.

Two kinds of verb, two shapes. Contribution registrations are synchronous and return nothing. Data calls return promises, and a forgotten `await` gives you a promise where you expected an array. A refused capability throws, naming the capability, rather than returning emptiness.

Rendering a panel

This is where the routes differ most, and for a fundamental reason: an iframe has its own document, a worker has none.

Route	The render receives	It must
Sandboxed JS	<code>{ root, id }</code>	Populate <code>ctx.root</code> . Its return value is ignored, so returning an HTML string produces nothing: the panel opens and stays blank.
Python	Nothing	Return HTML, which the host displays inert in a sandboxed iframe without <code>allow-scripts</code> .
Native, in the page	The real container	Populate it, with the whole application DOM within reach.

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. Paint a waiting state straight away, then fill in: the container stays valid in between.

A sandboxed JS plugin can therefore create nodes, attach listeners and draw, in its own document. What it does not have is the host's document.

The theme

A plugin does not hard code colours. It writes `var(--surface-1)` like a core component, and the bridge guarantees those variables exist in the sandbox document, then updates them on every theme change. The exposed list is deliberately closed: surfaces, text, borders, shadows and brand, plus a few inherited names in transition. Exposing everything would make the whole design system a public surface that could no longer evolve without breaking third party plugins.

Ask only for what you use. The consent screen ticks reads in advance and leaves writes to be ticked. A plugin asking for `samples:write` to draw a chart gets refused by attentive users, and that is the intended behaviour.

17. What the kernel refuses

Six refusals, each closing one door. Knowing them saves you from hunting for a bug where there is a rule.

1. **An identifier belonging to another source.** The token returned is inert and carries `accepte: false`, and an error names the owner. Without this rule a plugin would declare `core.go.dashboard` and take over navigation.
2. **A plugin contribution whose identifier does not start with the plugin id.** The same boundary, checked while validating the manifest, hence before any code runs.
3. **A command with no handler after activation.** The plugin declares what it does not implement. Without this guard we would call the stand-in handler again, hence activation, indefinitely.
4. **A connector whose address is not local.** Refused at registration, not at call time: letting it appear in the interface only to refuse it when the user needs it would be a way of lying on a delay.
5. **The native runtime without a consented capability,** and the native runtime unsigned in the distribution zone.
6. **Any evaluation of an expression supplied by a manifest.** The `when` grammar is poor by construction, and there is no `eval`.

A refusal has to be visible

The kernel returns an inert token rather than throwing: the caller should not have to guard against an exception it does not cause. But it does not return something that looks like success either, and it writes to the console who owns the identifier.

A contribution that vanishes with no message is the most expensive failure to diagnose, because nothing invites you to look in the right place.

Writing a plugin

From the workshop to the signature, four complete plugins, troubleshooting and the API reference.

18. From nothing to a loaded plugin

The plugin workshop is a development panel: working folder, manifest validation, hot reload, contribution inspector, packaging. It exists only in the local zone.

That is not a display convenience. The workshop backend reads and writes files on the machine; exposing it outside developer mode would amount to offering arbitrary file reading from the interface.

The five gestures

Gesture	What it does
Scaffold	Writes a valid v3 manifest and a code file into an empty folder. The shortest path to a plugin that loads.
Open a folder	The workshop rereads the manifest, validates it, and shows errors with their reason.
Hot reload	Releases the source, rereads, re-registers. A timestamp poll every 1.2 seconds does it for you if you leave automatic mode on.
Inspect	Compares what the manifest declares with what the registry actually holds.
Package	Produces the distributable archive.

The inspector, and why it could not exist before

Manifest v3 declares what a plugin brings, and the code implements it. Two distinct things, hence two things that can diverge. A plugin can declare a command it never implements, or register a contribution it never declared.

A gap is not always a fault. A contribution declared but not implemented may be work in progress. A contribution registered without being declared is more serious: it will escape the user's consent, who will never have seen that it existed.

This tool was impossible in v2: there was nothing to compare, since the surface was only known after execution, hence too late.

Two shapes, depending on where the code runs

A native plugin runs in the page and sees the kernel. A sandboxed plugin does not: it declares what it brings in its manifest, then calls the facade. Both forms below do the same thing.

```
// Page side: core file, user script, native plugin
(function () {
  'use strict';
  var K = window.IsoFind;

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

```
// Sandbox side: K is UNDEFINED. You declare in the manifest, and call
// the facade. Registrations are SYNCHRONOUS, data calls return promises.
(function () {
  'use strict';

  // Register BEFORE any call: a plugin that throws before registering
  // leaves no surface behind it at all.
  isofind.registerPanel('results', 'Results', function (ctx) {
    var host = ctx.root; // { root, id }, not the container
    host.innerHTML = '<p>Working...</p>';

    isofind.getSamples().then(function (samples) {
      host.innerHTML = render(samples); // the iframe has its own DOM
    }).catch(function (e) {
      host.textContent = e.message; // a refused capability throws
    });
  });

  isofind.registerMenu('open', 'Analyse', function () {
    isofind.openPanel('results');
  });
})();
```

Check the surface before writing a call. The JS facade and the Python facade do not carry the same verbs, and one SDK version is not another: `getAnalyses` exists on the Python side and not on the JS side, storage and R are sub-objects on the JS side only. `Object.keys(isofind)` describes yours in one line, and `isofind.version` dates it.

The shape of a native plugin

```
// myplugin.js, loaded in the sandbox
(function () {
  'use strict';
  var K = window.IsoFind;

  // The code runs only on the first invocation of a declared contribution.
  // It RE-REGISTERS what the manifest declared, under the same identifiers.

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

  K.shell.registerPanel({
    id: 'myplugin.results',
    source: 'myplugin',
    title: { fr: 'Résultats', en: 'Results' },
    dock: 'right',
    factory: function (ctx) { return build(ctx); }
  });
})();
```

Nothing to undo by hand at unload. All your registrations carry your source, so `K.releaseSource('myplugin')` removes them at once, including those in registries you never wrote. What you must undo yourself is what does not go through the registry: a listener on `window`, a timer, a request in flight. That is the job of the `dispose` your factories return.

19. Signing and distributing

An author signature lets a plugin load without the local zone, and lets it be offered in the catalogue. It rests on a key that never leaves your machine.

Step	Route	What happens
Create your identity	/plugins/author/identity	The local backend produces the key pair and writes it encrypted into the data directory.
Request a certificate	/plugins/author/request	A request carrying your identity goes to the IsoFind authority.
Install the certificate	/plugins/author/import	The certificate comes back and is filed next to the key.
Sign	/plugins/author/sign	The archive is signed locally.
Package	/plugins/dev/package	The workshop produces the archive to distribute.

The key travels over no network

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.

The interface exposes that constraint rather than hiding it. An author who does not understand why their key must stay with them will end up emailing it, believing they are being helpful.

A verified signature removes the need for the local zone and lifts the ban on unsigned plugins from the catalogue. It does not remove consent: capabilities still have to be granted, and a signed plugin asking for `runtime:native` still shows its warning.

20. Four plugins to copy

Four complete shapes, from manifest to code. Replace the identifiers, keep the structure.

These four examples are written page side. They hold for a core file, a user script or a native plugin. For a sandboxed plugin the manifest is unchanged, and the code calls the facade from section 16 instead of `K`.

A command and its panel

The most common shape: a computation launched from the palette, a result shown on the right.

```
// manifest
{ "manifest": 3, "id": "spectro", "name": "Spectro", "version": "1.0.0",
  "engine": { "runtime": "js" },
  "capabilities": ["samples:read", "ui:panel"],
  "activation": ["onCommand:spectro.analyse", "onPanel:spectro.results"],
  "contributes": {
    "commands": [{ "id": "spectro.analyse",
      "title": { "fr": "Analyser le spectre", "en": "Analyse spectrum" },
      "category": "Spectro" }],
    "panels": [{ "id": "spectro.results", "dock": "right",
      "title": { "fr": "Spectro", "en": "Spectro" } }]
  }
}

// code
K.commands.register({
```

```

    id: 'spectro.analyse', source: 'spectro',
    title: { fr: 'Analyser le spectre', en: 'Analyse spectrum' },
    handler: function () {
        var t = K.taches.demarrer({ titre: 'Analysis', source: 'spectro' });
        return analyse()
            .then(function (r) {
                last = r;
                t.reussir(r.peaks.length + ' peaks');
                K.shell.open('spectro.results');
                K.shell.signaler('spectro.results');
            })
            .catch(function (e) { t.echouer(e); throw e; });
    }
});

```

An editor for a format

An opener so that a double click leads somewhere, and an editor to show it as a central tab.

```

K.editors.register({
    kind: 'spectro.file', source: 'spectro',
    title: { fr: 'Spectre', en: 'Spectrum' },
    label: function (ref) { return ref.nom; },
    factory: function (ctx) {
        var view = draw(ctx.root, ctx.ref, ctx.state);
        // What fits in the state survives navigation; the rest is read again.
        ctx.saveState({ zoom: view.zoom });
        return { dispose: view.destroy };
    }
});

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

```

A dashboard widget fed by a task

```

K.dashboard.registerWidget({
    id: 'spectro.recent', source: 'spectro', order: 60,
    factory: function (host) {
        host.innerHTML = '<h3>Recent spectra</h3><ul></ul>';
        var list = host.querySelector('ul');

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

        // A widget repaints when something moves, without polling.
        var off = K.events.on('spectro.analysis.done', render);
        return { dispose: function () { off.dispose(); } };
    }
});

```

A workflow task and its context menu

The task makes the plugin usable from a nightly workflow, the menu makes it usable with the mouse. Both point at the same code.

```
K.workflows.declarerTache({
  id: 'spectro.analyse_batch', 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 analyseBatch(args.lot); }
});

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

21. Troubleshooting

Common failures, and where to start. Most of them are silences rather than errors, which is what makes them slow to find.

Symptom	Likely cause	Where to look
The panel does not appear, no message	<code>K.shell</code> was absent when the file loaded	Wrap the registration in <code>K.whenShellReady</code> .
The contribution disappears after a plugin reload	It was registered without <code>source</code> , hence filed under <code>core</code> and never released, then replaced	The registry warnings in the console.
Registration seems to succeed but nothing shows	The identifier belongs to another source, the token is inert	The console error names the owner. Test <code>token.accepte</code> .
The command is never offered	Its <code>when</code> is false, its capability was not granted, or it is restricted to the local zone	<code>K.commands.list({ available: false })</code> shows it anyway.
The manifest is refused at load	Identifier prefix, unknown dock, invalid expression, contribution never activable	The console lists every error with its reason; the workshop shows them.
"declares X but does not implement it"	The code loaded without re-registering the declared contribution	The workshop inspector compares the two lists.
Listeners pile up on every opening	The factory returns no <code>dispose</code> , or it does not undo everything	Open and close the panel ten times and watch what accumulates.
The panel state is lost on every navigation	Nothing was filed through <code>ctx.saveState</code>	Reminder: no memory survives a page.
The connector appears nowhere	Its address is not local, registration was refused	The console, and the security journal.
The Python plugin hangs on a call	The facade has been asynchronous since the move to the worker	<code>await</code> in front of every <code>isofind</code> call.

Symptom	Likely cause	Where to look
The plugin refuses to load in production	Unsigned, and either from the catalogue or outside the local zone	Section 15, and the decision tree.
A modal does not show and the code stops	A call to <code>window.confirm</code> or <code>window.prompt</code>	Go through <code>IsoFindModales</code> .

The useful reflex. Open the console and run `K.list('command', function (c) { return c.source === 'myplugin'; })`. What the registry holds is the truth; what you believe you registered is not always the same thing.

22. API reference

The public surfaces, and the file that installs each one. They all live under `window.IsoFind`, which this guide calls `K`.

The kernel

Surface	Contents
<code>K.contribute(type, entry)</code>	Generic registration, returns a token
<code>K.get(type, id) / K.list(type, filter)</code>	Reading the registry
<code>K.releaseSource(source)</code>	Frees everything a source registered
<code>K.whenShellReady(fn)</code>	Waits for the shell to install
<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>	Bilingual titles
<code>K.Disposable / K.DisposableStore</code>	Tokens, and stores of tokens

The surfaces

Everything below lives in the page. From a sandboxed runtime none of it is reachable directly: you go through the facade in section 16, and the bridge materialises the contribution here.

Surface	Entry points	File
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
K.notifications	info, attention, erreur, taire, rouvrir	notifications.js

Surface	Entry points	File
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	Notebook kernels, assistant agents, campaigns	notebook-kernels.js, agent-manager.js, campagnes.js
K.log(source, text, level)	Writes to the journal	panel-journal.js

Interface services

Global	What it provides
IsoFindModales	confirmer, informer, demander, popup, deroulant, ouvrir
K.DataGrid	creer, typeDe, TYPES: the shared data grid
IsoFindGraphiques	The chart theme

Global	What it provides
IsoFindColoration	Syntax highlighting
IsoFindListes	The dropdowns used across the application
IsoFindPalette / IsoFindRaccourcis / IsoFindLayouts	Palette, shortcuts, named layouts

Use the shared services rather than your own. A grid hand written inside a plugin will not follow the theme, will not know how to export, will not copy in spreadsheet format, and will drift at the first change. The same holds for dropdowns, modals and highlighting.

What stays outside the kernel

The plugin manager lives in `window.pluginsHandlers` : installation, signature, known capabilities, code loading. The kernel never calls it directly, it delegates through the three hooks in section 16. That is what lets each evolve without the other, and lets a security review concentrate on one file at a time.