

Les workflows

De la tâche au passage signé

Référence technique du moteur, du registre des capacités, du journal des accès et des soixante et une tâches du noyau.

Sommaire

Comprendre

1. Modèle mental	3
2. Le modèle d'exécution	4
3. Anatomie d'un workflow	5
4. L'étape et ses réglages	7
5. Substitution et contexte	9
6. Contrat et implémentation	11

Exécuter

7. Les états	13
8. Lancer, simuler, déboguer	14
9. Le débogueur	15
10. Déclencheurs et armement	16
11. Composition et itération	18

Gouverner

12. Capacités et consentement	20
13. Journal des accès	22
14. Le panneau et les commandes	23

Référence

15. Catalogue des tâches	26
16. Ce que voit l'assistant	33
17. Quatre workflows types	34
18. Dépannage	37
19. Bornes et persistance	38
20. Déclarer une tâche	40

Comprendre

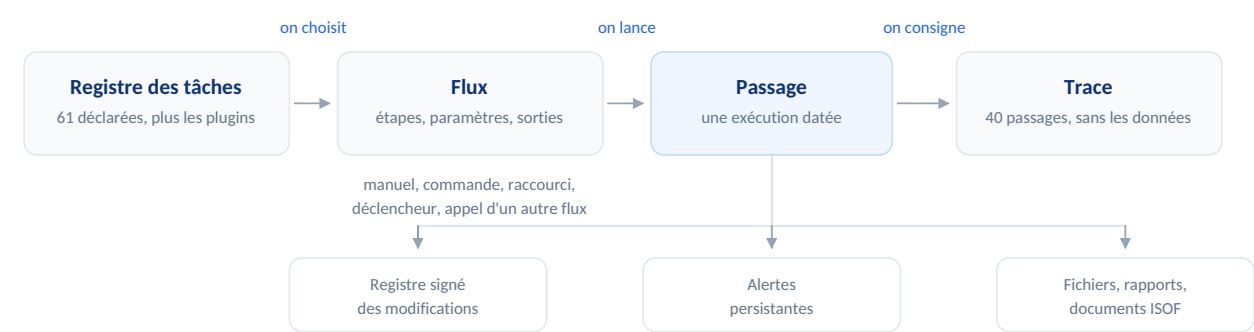
Ce qu'est un workflow, comment il est écrit, et sur quoi il est permis de s'appuyer quand on en écrit un.

1. Modèle mental

Cinq mots suffisent à décrire tout le système, à condition de ne pas les confondre. Deux d'entre eux, le flux et le passage, portent la distinction la plus utile : l'un est un texte, l'autre est un évènement daté.

Mot	Ce que c'est	Où il vit
Tâche	Une capacité d'action déclarée par le noyau ou par un plugin, avec ses entrées, ses sorties et son drapeau d'écriture. Elle ne s'écrit pas dans l'interface, elle s'y choisit.	En mémoire, dans le registre des tâches, reconstruit à chaque chargement de page.
Flux	Ce qu'on appelle un workflow : une suite d'étapes, des paramètres, des sorties déclarées, éventuellement un déclencheur. Un texte, qui ne fait rien tant qu'on ne le lance pas.	Dans le stockage local, sous <code>isofind_workflows</code> .
Étape	Un maillon du flux : une tâche, ses arguments, une condition, un repli. Son identifiant sert d'adresse pour lire ses sorties.	Dans la définition du flux.
Passage	Une exécution du flux, à une date, avec des paramètres, dans un mode. Deux passages du même flux ne se ressemblent pas nécessairement.	En mémoire pendant l'exécution, puis dans la trace, allégée de ses données.
Tour	Une itération de <code>core.pour_chaque</code> . Chaque tour est un passage complet du flux appelé, avec sa propre trace, repliée sous celle de son parent.	Comme un passage.

Le cycle de vie



Un flux importé passe en plus par l'écran de consentement avant son premier lancement, et la liste des accès accordés voyage ensuite avec lui, y compris dans les flux qu'il appelle.

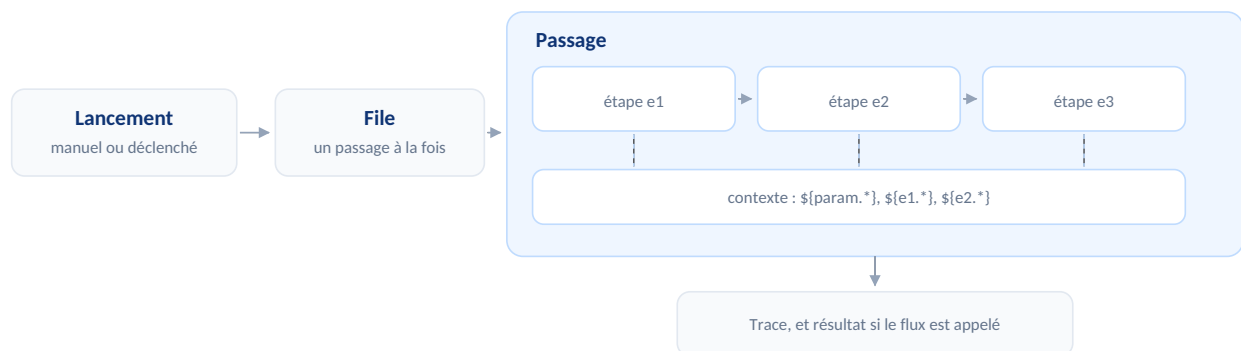
Trois idées qui expliquent la plupart des choix

1. **Le flux ne fait rien, le passage fait tout.** Modifier un flux ne modifie aucun passage déjà enregistré ; relire un passage ne relance rien. C'est aussi pourquoi la trace conserve les paramètres du passage et non ceux du flux au moment où on la relit.
2. **Le moteur ne fait confiance ni au modèle ni au fichier reçu.** Ce qui protège n'est jamais une consigne écrite quelque part, mais un outil qu'on n'a pas donné, une capacité qu'on n'a pas accordée, ou une substitution qui n'évalue rien.
3. **Ce qui tourne sans témoin doit laisser quelque chose derrière lui.** C'est ce qui explique la trace persistante, les alertes qui survivent au rechargement, le détail des étapes dans le panneau, et le fait qu'une tâche fasse remonter le message d'erreur du serveur plutôt qu'un code.

2. Le modèle d'exécution

Une tâche se distingue d'une commande sur un point qui gouverne tout le reste : elle s'exécute sans écran.

Les commandes du domaine ouvrent des fenêtres. Un import demande un fichier, une fiche demande un formulaire, et c'est très bien tant qu'un utilisateur est là pour répondre. Un moteur qui les enchaînerait produirait un diaporama de modales, avec un clic à chaque étape, ce qui ne ferait gagner à peu près rien. Les tâches sont donc déclarées à part : elles acceptent des paramètres, elles rendent un résultat, et rien ne les oblige à s'adresser à quelqu'un pendant qu'elles tournent.



Les étapes s'enchainent dans un seul sens. Chacune complète le contexte, que les suivantes lisent, et aucune ne peut renvoyer vers une étape déjà passée.

Le workflow entre dans le registre comme un agent

Le registre des modifications sépare l'acteur, qui décide, de l'agent, qui exécute. Une écriture faite par un workflow s'y inscrit sous la forme `Colin Ferrari, via workflow « Correction blanc v3 »`, et la mention s'allonge quand il y a lieu : le déclencheur qui l'a lancé, et le signataire chez qui le flux a été écrit s'il vient d'ailleurs.

Devant quarante corrections identiques, la première question d'un auditeur est de savoir s'il regarde un geste répété quarante fois ou une règle automatique, puisque les deux n'engagent pas la même responsabilité.

Quelles tâches sont retenues

Une tâche entre dans le registre si elle est sans écran, déterministe, et si ses paramètres suffisent à décrire son effet. Tout ce qui suppose un choix humain reste une commande. On automatise l'exécution d'une décision, pas la décision elle-même.

Ce qui manque volontairement

- **La suppression.** Un workflow se reçoit, se copie, s'importe. Lui donner le pouvoir d'effacer reviendrait à accepter qu'un fichier arrivé par courriel vide une base. Le registre garderait la trace de la suppression, ce qui est déjà beaucoup, mais une trace ne remplace pas une donnée.
- **L'exécution de code.** Elle rendrait décoratif le confinement des plugins, puisqu'il suffirait d'écrire un workflow pour en sortir. C'est aussi la raison pour laquelle l'écriture de fichiers reste hors de portée du modèle : un fichier écrit peut être un script.
- **L'évaluation d'expressions.** La substitution est littérale, sans `eval` nulle part, y compris dans le débogueur.
- **Le retour arrière.** L'enchaînement est linéaire, avec des conditions. Écrire un programme dans une interface graphique donne quelque chose qu'on ne peut ni relire, ni tester, ni versionner ; quand une boucle devient nécessaire, IsoFind sait déjà exécuter du Python et du R.

L'itération de `core.pour_chaque` ne revient pas sur ce dernier point. Le nombre de tours vaut la longueur d'une liste connue avant de partir, il est plafonné, et rien à l'intérieur ne décide de continuer.

3. Anatomie d'un workflow

Un workflow est un objet JSON, conservé dans le stockage local sous la clef `isofind_workflows` et rechargé au démarrage.

Le plus petit workflow qui serve à quelque chose

Trois choses sont indispensables : un identifiant, un nom, et au moins une étape portant une tâche. Tout le reste a un défaut.

```
{
  id: "wf-inventaire",
  nom: "Inventaire du jour",
  etapes: [
    { id: "compte", tache: "core.inventaire", args: { par: "element" } },
    { id: "dire", tache: "core.alerter",
      args: { message: "${compte.nombre} lignes dans l'inventaire" } }
  ]
}
```

Il se lance, il se simule, il apparaît dans la palette, et il peut recevoir un déclencheur sans qu'on lui ajoute rien. L'identifiant `compte` de la première étape est ce qui rend sa sortie lisible par la seconde : sans lui, l'étape tournerait quand même et le message serait vide.

La forme complète

```
{
  id: "wf-lx8k2",           // stable, sert de clef et de cible d'appel
  nom: "Correction de blanc", // ce que l'utilisateur voit et peut appeler
  description: "...",

  parametres: [             // lus par ${param.nom} dans les etapes
    { nom: "element", default: "Sr" }
  ],

  sorties: [                // ce qu'il rend a un workflow appelant
    { nom: "total", valeur: "${e2.lignes}" }
  ],

  etapes: [
    { id: "e1", tache: "core.sql", args: { sql: "SELECT ..." },
      si: null, continuerSiEchec: false, repli: null }
  ],

  declencheur: {            // declarer ne suffit pas : voir « arme »
    type: "evenement" | "horaire",
    evenement: "lims.pull.done",
    si: "evenement.resultat.fichiers",
    heure: "03:00", jours: [1,2,3,4,5],
    simulation: false
  },
  arme: false,              // le second geste, jamais importe

  origine: { signataire: "...", valide: true }, // present si importe
  capacites: ["samples:read", "files:write"]    // accordees a l'import
}
```

Chaque workflow devient une commande. À l'enregistrement, le moteur inscrit `workflow.<id>` dans le registre des commandes. Le flux apparait donc dans la palette, dans le panneau Actions et dans la recherche universelle, et il peut recevoir un raccourci clavier, au lieu de rester un réglage rangé au fond d'un menu.

Paramètres

Deux champs seulement, un nom et une valeur par défaut. Le défaut sert moins au confort qu'au lancement automatique : une commande, un raccourci ou un déclencheur partent sans personne pour répondre à une boîte de dialogue, et le workflow partirait alors avec des variables vides, échouerait à la première substitution, et l'utilisateur en tiendrait le workflow pour responsable plutôt que son mode de lancement.

C'est aussi ce qui évite les copies. « Correction blanc » existait en quatre exemplaires parce qu'elle différait d'un élément ; un paramètre lu par `${param.element}` en supprime trois.

Au lancement manuel, un workflow qui déclare des paramètres demande leurs valeurs, et un workflow qui n'en déclare aucun part sans rien demander. Une confirmation avant chaque lancement transformerait une automatisation en formulaire.

Sorties déclarées

Elles disent ce que le workflow rend quand un autre l'appelle. Sans déclaration, l'appelant reçoit les sorties de toutes les étapes nommées, rangées sous leurs identifiants. Cela fonctionne, mais l'appelant se retrouve lié à la structure interne de l'appelé, et renommer une étape casse alors un flux qu'on n'a pas ouvert.

La déclaration a un second effet, dans les boucles. Un flux qui déclare ses sorties voit ses résultats conservés tour par tour ; sans déclaration, seul l'état de chaque tour est gardé, faute de quoi cinq cents tours ramèneraient en mémoire toutes leurs étapes.

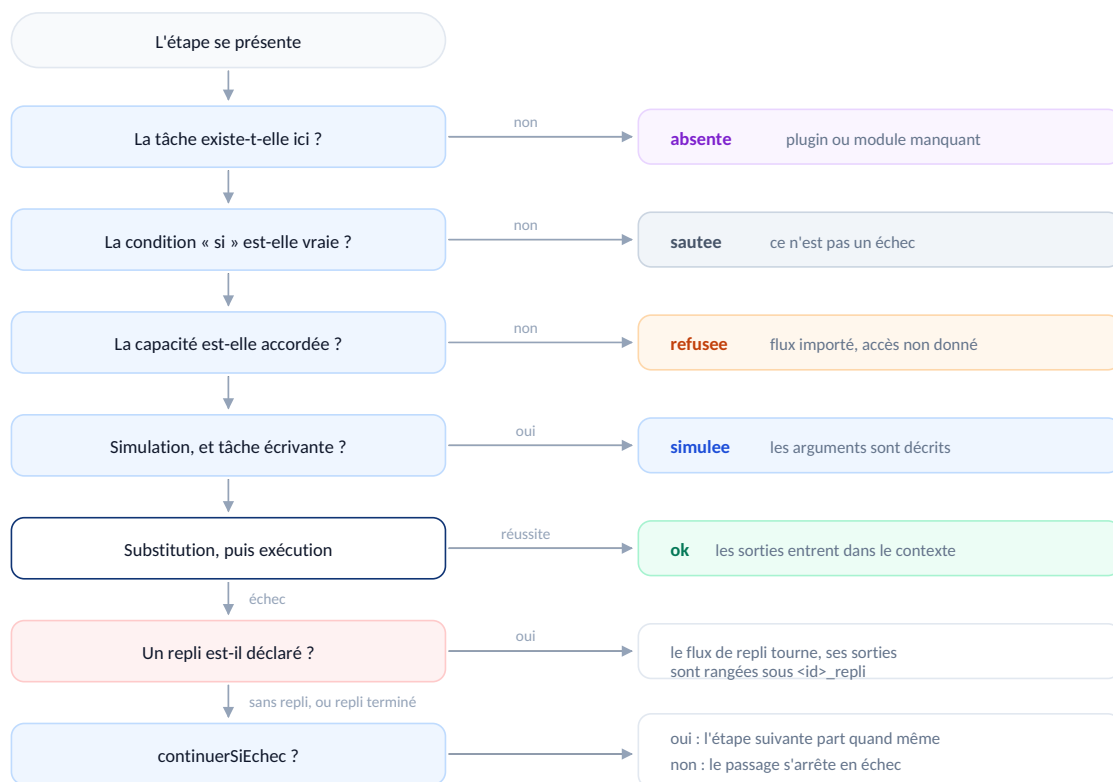
Le résultat n'est calculé que pour un flux appelé. À la racine, personne ne le lirait, et il retiendrait les sorties des quarante passages conservés.

4. L'étape et ses réglages

L'éditeur ne connaît aucune tâche en particulier : il dérive les champs d'une étape du schéma `entrees` de la tâche choisie. Une tâche apportée par un plugin devient donc éditable sans qu'on touche au panneau.

Champ	Rôle	Défaut
<code>id</code>	Nom sous lequel les sorties de l'étape deviennent lisibles, par <code>\${id.sortie}</code> . Une étape sans identifiant s'exécute, mais ne laisse rien derrière elle.	<code>e1</code> , <code>e2</code> , ...
<code>tache</code>	Identifiant de la tâche. La liste déroulante vient du registre et se groupe par capacité, ce qui indique, au moment du choix, ce que l'étape ajoutera à l'écran de consentement.	<code>requis</code>
<code>args</code>	Un champ par entrée déclarée. Changer de tâche vide les arguments, sans quoi l'ancien contenu resterait dans le fichier sans être affiché nulle part.	<code>{}</code>
<code>si</code>	Un chemin de variable, pas une expression. L'étape est sautée quand la valeur désignée est fausse. Pour comparer deux nombres, il faut une étape <code>core.condition</code> en amont et lire son <code>vrai</code> .	<code>null</code>
<code>continuerSiEchec</code>	Décide si l'échec arrête le passage. S'applique aussi aux états <i>absente</i> et <i>refusée</i> .	<code>false</code>
<code>repli</code>	Workflow lancé si l'étape échoue, choisi dans une liste. Il reçoit <code>\${param.erreur}</code> , <code>\${param.etape}</code> et <code>\${param.tache}</code> .	<code>null</code>

Ce que le moteur décide, dans l'ordre



L'ordre compte : la capacité est vérifiée avant que les arguments ne soient préparés, et une étape refusée ou sautée n'a donc jamais commencé.

Le repli est un workflow, et non un renvoi vers une autre étape

Renvoyer une étape en échec vers une étape précédente du même flux aurait introduit un branchement arrière, donc la boucle que le moteur refuse par ailleurs. Un workflow de repli garde l'enchaînement linéaire : il s'écrit ailleurs, se relit seul, se simule seul, et la pile d'appel empêche déjà qu'il se rappelle lui-même.

Ses sorties sont rangées sous `<id>_repli`, jamais sous l'identifiant de l'étape qui a cédé. Si on les y mettait, une étape suivante lisant `#{e2.lignes}` recevrait selon les jours le résultat ou le rattrapage, sans rien pour les distinguer, et le workflow paraîtrait marcher jusqu'au moment où il compterait faux.

Dans la trace, le repli ne compte pas comme une étape de plus : il porte le rang de celle qu'il rattrape, ce qui garde « étape 3 sur 5 » exact même quand trois étapes ont eu un repli.

Repli et poursuite se règlent séparément. Le premier dit quoi faire de l'échec, le second s'il arrête le passage. Les réunir ferait tourner la suite du workflow sur des variables absentes chaque fois qu'on veut seulement être prévenu. Un repli en échec ne masque pas l'échec d'origine : c'est celui-là qui continue de remonter.

Types d'entrée reconnus par l'éditeur

type	Contrôle affiché	Note
texte	Champ libre	Accepte les substitutions

nombre	Champ libre	Converti par la tâche
booléen	Case à cocher	La substitution rend du texte, donc les tâches testent <code>true</code> et <code>'true'</code>
objet	Champ libre	Objet ou texte JSON, les deux sont acceptés
liste	Champ libre	En pratique une référence à une sortie
sql	Champ en chasse fixe	Placeholder <code>SELECT ...</code>
workflow	Liste déroulante des flux	Un champ libre donnerait une étape qui n'échoue qu'à l'exécution, souvent la nuit
commande	Champ libre	Identifiant de commande

Le flux en cours d'édition ne figure jamais dans les listes de workflows, ni pour le repli ni pour un appel : le moteur refuserait l'appel circulaire à l'exécution, et le proposer reviendrait à offrir une erreur.

5. Substitution et contexte

Une étape lit les sorties des précédentes par `${chemin.pointe}`. La substitution reste littérale, et rien de ce qui est écrit dans un workflow n'est évalué comme du code.

Référence seule ou référence dans un texte

Quand la valeur vaut exactement `${e1.lignes}`, le moteur rend la valeur elle-même. Sinon un tableau de trois identifiants arriverait sous la forme `1,2,3` à une tâche qui attend une liste.

Quand la référence se trouve au milieu d'un texte, chaque occurrence est remplacée par sa forme textuelle, et une valeur absente laisse une chaîne vide. C'est ce qui permet d'écrire un message d'alerte comme `${e2.erreurs} fichiers rejetés cette nuit`.

Ce que contient le contexte

Chemin	Contenu
<code>\${param.nom}</code>	Les paramètres du passage : les défauts déclarés, écrasés par les variables de lancement.
<code>\${e2.lignes}</code>	Les sorties de l'étape <code>e2</code> , rangées sous son identifiant plutôt que fusionnées à plat, faute de quoi deux étapes produisant toutes deux <code>lignes</code> s'écraseraient.
<code>\${e2_repli.x}</code>	Les sorties du workflow de repli de l'étape <code>e2</code> .
<code>\${e3.sorties.total}</code>	Pour une étape <code>core.workflow</code> : ce que le flux appelé a déclaré rendre.
<code>\${evenement...}</code>	La charge de l'évènement, pour un passage déclenché.

`${param.element}`, Dans un corps de boucle : l'élément du tour, son rang et le nombre de tours.
`${param.index}`,
`${param.total}`

Cas de substitution

En supposant que l'étape `e1` ait rendu `{ nombre: 238, lignes: [...], colonnes: [...] }` et que le paramètre `element` vaille `Sr`.

Écrit dans l'argument	Reçu par la tâche	Remarque
<code>\${e1.nombre}</code>	238	Référence seule : la valeur passe avec son type.
<code>\${e1.lignes}</code>	[...]	Le tableau lui-même, et non sa forme textuelle.
<code>\${e1}</code>	{ nombre, lignes, colonnes }	La sortie entière. C'est ce qu'attendent <code>core.pour_chaque</code> et <code>core.exporter_fichier</code> .
<code>\${e1.nombre} lignes</code>	"238 lignes"	Référence dans un texte : le résultat est du texte.
<code>\${e1.absent}</code>	undefined	Aucune erreur. C'est la tâche qui s'en plaindra, ou personne.
<code>\${e1.absent} lignes</code>	" lignes"	Une valeur absente laisse une chaîne vide, sans le signaler.
<code>\${param.element}</code>	"Sr"	Les paramètres sont du texte, même quand ils portent un nombre.
<code>\${e3.sorties.total}</code>	la valeur déclarée	Le chemin traverse les objets sans limite de profondeur.
<code>e1.nombre</code>	"e1.nombre"	Sans les accolades, rien n'est substitué. Sauf dans le champ <code>si</code> , qui attend un chemin nu.
<code>\${e1.lignes[0]}</code>	undefined	L'indexation par crochets n'existe pas. Le chemin se découpe aux points, donc <code>\${e1.lignes.0}</code> désigne le premier élément.

La substitution ne protège pas ce qu'elle insère. Un paramètre inséré dans une requête SQL y entre tel quel : une apostrophe dans la valeur casse la requête, et le message d'erreur viendra de la base. Pour un critère qui vient d'un paramètre, mieux vaut une tâche qui prend ce critère en argument, comme `core.analyses` ou `core.echantillons`, qu'une requête écrite à la main.

Pourquoi les paramètres ont leur propre espace de noms. Une étape peut s'appeler `element`. Sans le préfixe `param`, `${element}` désignerait tantôt ses sorties, tantôt un paramètre, selon l'ordre dans lequel les deux ont été écrits, et une ambiguïté qui se résout par l'ordre finit toujours par surprendre quelqu'un.

Une référence qui ne désigne rien ne fait pas échouer l'étape. Elle produit une valeur absente, et c'est la tâche qui s'en plaint, parfois plusieurs étapes plus loin. C'est la panne la plus fréquente, et c'est ce que le débogueur sert d'abord à diagnostiquer.

6. Contrat et implémentation

Rien ne recompile ici. Un workflow enregistré est du texte qui cite des noms, et ces noms sont résolus au lancement, parfois des mois plus tard. La frontière entre ce qui est promis et ce qui est libre de bouger mérite donc d'être explicite.

Dans chaque fichier de tâches, la déclaration est le contrat et la fonction `executer` est l'implémentation. C'est aussi pourquoi le catalogue de la section 15 est produit à partir des déclarations : ce qui s'y lit est exactement ce sur quoi un flux peut s'appuyer.

Objet	Contrat	Implémentation
Tâche	Son identifiant, ses drapeaux <code>ecrit</code> et <code>outil</code> , sa capacité, le nom et le type de ses entrées, le nom de ses sorties et le sens des valeurs qu'elles portent.	La fonction <code>executer</code> , les routes appelées, la forme des requêtes, la rédaction des messages d'erreur, le fichier où la tâche est déclarée.
Flux	Son identifiant, son nom (parce qu'un appel peut le désigner ainsi), ses paramètres et leurs défauts, ses sorties déclarées.	Ses étapes, leur ordre, leurs identifiants, ses conditions et ses replis, tant que les sorties déclarées gardent leur sens.
Étape	Son identifiant, cité par les étapes suivantes et par les points d'arrêt.	La tâche employée et ses arguments.
Moteur	La substitution littérale, les noms d'états, l'ordre linéaire, l'agent transmis à chaque tâche, le fait qu'une écriture soit sautée en simulation.	La file, la pile, la porte de débogage, les minuteurs, la façon dont la trace est allégée.
Trace	Les champs conservés d'un passage : étapes, états, messages, durées, aperçu des sorties, paramètres.	Son encodage, sa taille, la clef de stockage.

Ce qui casse des flux déjà enregistrés

- **Renommer une sortie de tâche.** Tous les flux qui lisaient `#{e1.ancien_nom}` reçoivent une valeur absente, sans erreur au chargement, parce que rien ne vérifie ces chaînes avant l'exécution.
- **Rendre requise une entrée qui ne l'était pas.** Les flux existants ne la fournissent pas. Ajouter une entrée facultative avec un défaut, en revanche, ne casse rien.
- **Renommer un flux.** Les appels par identifiant continuent de fonctionner ; les appels par nom, non.
- **Renommer une étape** quand le flux ne déclare pas ses sorties : l'appelant lisait l'identifiant, il ne trouve plus rien. Les points d'arrêt posés sur cette étape sont également perdus.
- **Changer la capacité d'une tâche.** Un flux importé qui avait reçu l'ancienne voit ses étapes passer à l'état *refusée* tant que les accès n'ont pas été révisés.

- Passer **ecrit** de faux à vrai. La tâche sort du mode conversation et devient sautée en simulation, ce qui change le comportement des flux qui l'emploient sans qu'on les ait touchés.

Les bornes sont des garde-fous, pas des promesses

Les plafonds décrits en section 19 protègent la machine et la base. Ils peuvent changer, et un flux ne devrait pas s'appuyer sur leur valeur exacte. Un flux qui compte sur cinq cents tours pour traiter une liste de quatre cent quatre-vingts éléments est un flux qui tombera le jour où la liste grandit, avec **tronque** pour seul avertissement.

La frontière avec le serveur suit la même logique. Les routes sont l'implémentation des tâches, et les vérifications qu'elles font, sur les racines autorisées, la visibilité et les plafonds d'écriture, sont le contrat de sécurité. Rien n'est redoublé dans la page : une seconde série de contrôles finirait par diverger, et l'une des deux deviendrait la porte que personne ne surveille.

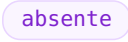
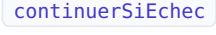
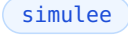
Exécuter

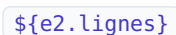
Ce qui se passe au lancement : les états, les trois modes, le pas à pas, les déclenchements et la composition.

7. Les états

Six états d'étape et quatre états de passage. Les garder distincts est ce qui permet de relire une trace sans avoir à deviner ce qui s'est passé.

Étape

État	Ce qui s'est passé	Arrête le passage	Ce que lisent les étapes suivantes
	Exécutée sans erreur.	non	Ses sorties, sous son identifiant.
	Condition  fausse, ou étape passée depuis le débogueur. Le message précise alors qui l'a décidé, car un pas manquant sans auteur est une énigme pour qui relit la trace.	non	Rien.
	Tâche inconnue sur cette installation, généralement un flux écrit pour un autre poste ou pour un plugin qui n'est pas là.	oui, sauf 	Rien.
	Capacité non accordée à un flux importé. Le refus se lit dans la trace à côté de « sautée » et de « absente », plutôt que comme une panne.	oui, sauf 	Rien.
	Écriture décrite et non exécutée. La trace garde les arguments qu'elle aurait reçus, et le panneau les montre sous ce titre plutôt que comme un résultat.	non	Rien : les arguments décrits vivent dans la trace, pas dans le contexte.
	Erreur remontée par la tâche, avec son message réel.	oui, sauf 	Rien, sauf les sorties du repli sous  .

Quatre de ces six états ne laissent rien dans le contexte. Une étape suivante qui lit  reçoit alors une valeur absente, sans erreur, ce qui explique la plupart des pannes décrites en section 18.

Passage

État	Ce qu'il signifie
encours	Le passage tourne, ou attend un ordre du débogueur.
ok	Toutes les étapes ont abouti, ou celles qui ont cédé portaient <code>continuerSiEchec</code> .
echec	Une étape a cédé et rien ne demandait de poursuivre. Le message du passage est celui de l'étape.
annulé	Arrêté à la main, ou depuis le débogueur. Les écritures déjà parties sont allées au bout.

Un flux appelé calcule son résultat dans les deux cas, succès comme échec. Avec `continuerSiEchec`, les étapes qui ont abouti ont produit quelque chose, et l'appelant doit pouvoir le lire pour décider de la suite.

Le passage connaît aussi son nombre d'étapes prévues avant de partir, puisque le flux existe déjà. C'est ce qui permet à un observateur extérieur, un badge de progression ou le panneau de suivi, d'afficher un avancement pendant l'exécution plutôt qu'après.

Annuler lève un drapeau, sans interrompre. Une tâche en vol qui écrit dans la base finit son écriture. La couper au milieu laisserait la base dans un état que le registre ne saurait pas décrire. Un passage en pause dans le débogueur ne verrait jamais ce drapeau, donc annuler ferme aussi la session de débogage.

8. Lancer, simuler, déboguer

Trois façons de lancer le même workflow. Le mode est un argument de lancement et non un réglage du flux, ce qui permet de lancer, simuler ou déboguer sans rien modifier.

Appel	Lectures	Écritures	Usage
executer	exécutées	exécutées	Lancement normal, manuel ou déclenché.
simuler	exécutées	décrites, non faites	Voir tourner un flux avant de l'armer.
deboguer	exécutées	exécutées	Pas à pas, avec arrêt avant chaque étape retenue.

La simulation ne fabrique pas de sorties

Une tâche qui écrit rend un objet vide, donc les étapes suivantes voient des variables absentes. C'est volontaire : en fabriquant des sorties plausibles, on obtiendrait une simulation qui réussit là où l'exécution réelle échouerait, et le premier lancement réel serait de nouveau un saut dans le vide.

Un workflow appelé depuis une simulation est simulé lui aussi. Sans cette transmission, simuler un flux parent exécuterait pour de bon les écritures de ses enfants.

Une conséquence à connaître : un flux dont une lecture dépend d'une écriture précédente échouera en simulation, et c'est le comportement attendu plutôt qu'un défaut de la simulation.

Armer sans avoir simulé. Sur un logiciel dont la provenance est signée, mieux vaut que la première exécution réelle ne soit pas la première exécution tout court. C'est aussi le moment où l'on s'aperçoit qu'un critère désigne quatre mille lignes au lieu de douze.

Un déclencheur peut lui-même être réglé en simulation. Un flux armé tourne alors à chaque évènement sans rien écrire, ce qui sert à observer pendant quelques nuits ce qu'il ferait, alertes comprises, puisque `core.alerter` reste active en simulation et signale l'origine de ses messages.

La file plutôt que le refus

Un seul passage racine tourne à la fois. Refuser les autres se tenait tant qu'un humain cliquait, puisqu'il voyait le message et recommençait. Avec des déclencheurs, deux acquisitions à quinze secondes d'écart feraient perdre la seconde, en silence, la nuit. Les passages en trop attendent donc leur tour, dans une file bornée à vingt : au delà, un déclencheur trop bavard deviendrait une fuite de mémoire.

Un passage imbriqué ne passe jamais par la file et s'exécute dans le tour de son parent. Mis en file, il attendrait derrière ce parent, qui attend justement qu'il finisse.

9. Le débogueur

Un enchainement linéaire échoue presque toujours de la même façon : un argument arrive vide parce que `${e2.lignes}` ne désignait pas ce qu'on croyait. La trace le dit après coup, quand le contexte a disparu.

Le pas à pas s'obtient en intercalant une porte au début de chaque maillon de la chaîne. Hors session, cette porte rend `undefined`, la chaîne ne fait pas même un tour d'évènements de plus, et le débogueur ne coute rien quand il dort.

Une session démarre en pause, avant la première étape. Un bouton « déboguer » qui se comporterait comme « lancer » tant qu'aucun point d'arrêt n'est posé obligerait à préparer la séance avant de pouvoir la conduire, alors que ce qu'on veut voir d'abord, c'est partir l'étape 1.

Les ordres

Ordre	Effet	Pas à pas ensuite
<code>pas</code>	Exécute l'étape et s'arrête à la suivante.	armé
<code>sauter</code>	Passe l'étape sans l'exécuter. Elle est marquée <i>sautée</i> , avec un message qui attribue la décision au débogueur.	armé
<code>continuer</code>	Laisse courir jusqu'au prochain point d'arrêt.	désarmé
<code>arreter</code>	Annule le passage.	session close

Ce que montre un arrêt

Le rang et le titre de l'étape, ses arguments, sa condition, le contexte réel du passage, et deux informations calculées avant qu'elle ne parte : si elle écrit, et si elle serait refusée. Les afficher à cet instant laisse encore le choix de ne pas la laisser partir. Voir « accès non accordé » avant de faire le pas évite aussi de chercher la cause dans le workflow alors qu'elle se trouve dans le consentement donné à l'import.

Le champ d'évaluation ne fait que substituer

Il applique la substitution littérale du moteur au contexte de l'arrêt. Ce qu'il affiche correspond donc exactement à ce que l'étape recevrait, et rien de ce qu'on y tape n'est exécuté. La règle qui interdit l'évaluation dans un workflow vaut aussi pour son débogueur.

Il accepte les deux formes : un chemin nu comme `e2.lignes`, et un texte qui contient des références, comme `l'étape ${param.etape} a cédé`. Une référence au milieu d'une phrase est un cas courant dans les messages d'alerte, et c'est justement là qu'on veut vérifier ce qui sortira.

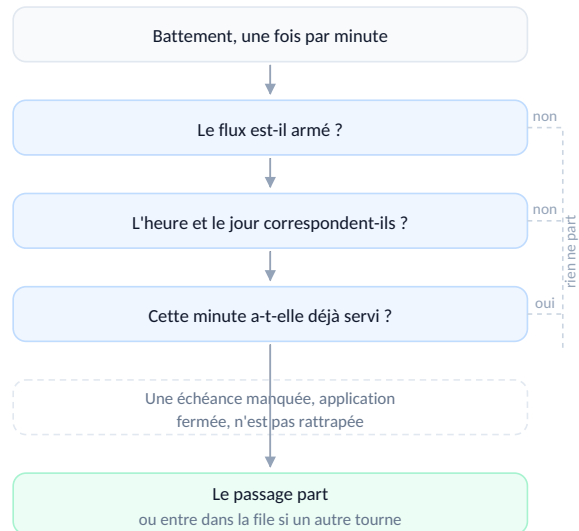
- **Les points d'arrêt vivent hors des workflows**, dans le stockage local sous `isofind_workflow_points`. Rangés dans la définition, ils partiraient avec un workflow exporté et arriveraient chez un collègue qui n'a rien demandé.
- **Le repère** est l'identifiant de l'étape quand il existe, son rang sinon. Un point posé sur une étape sans identifiant suit donc le rang, et se déplace à la première insertion.
- **Le débogage descend avec l'appel**. Un flux appelé depuis une session s'arrête à sa première étape pendant que son parent attend son tour de chaîne, ce qui donne le « pas dans » d'un environnement de développement sans rien ajouter. Un flux appelé hérite de l'état du pas à pas, mais pas de la pause initiale, sans quoi une boucle rouvrirait une pause à chaque tour.
- **Un dernier arrêt** a lieu après la dernière étape, sinon les sorties de l'étape finale ne seraient jamais visibles.
- **Le contexte n'est ni copié ni persisté**. Il porte les sorties véritables, tableaux de mesures compris, et il est détaché du passage à la fin de la session.
- **Quinze minutes sans geste** ferment la session. Un passage en pause occupe le moteur, donc la file, et un débogage laissé ouvert avant de partir déjeuner ferait refuser les déclenchements de l'après-midi sans qu'on fasse le lien.

10. Déclencheurs et armement

Une acquisition tombe à trois heures du matin. Si la correction de blanc attend qu'un humain clique, elle attendra le matin, et l'automatisation n'aura servi qu'à économiser un formulaire.

Déclencheur	Quand	Charge lisible
(aucun)	Lancement manuel, commande, raccourci.	-
<code>lims.pull.done</code>	Après une acquisition LIMS.	<code>\${evenement.resultat...}</code>
<code>lims.pull.error</code>	Après un échec d'acquisition.	<code>\${evenement...}</code>
<code>workflow.finished</code>	Après un autre workflow.	<code>\${evenement.passage...}</code>

La condition `si` porte sur la charge de l'évènement. Un tirage LIMS qui n'a rien ingéré émet quand même : sans condition, un workflow armé sur `lims.pull.done` repartirait toutes les cinq secondes sur rien, et la trace se remplirait de passages vides. Pour l'horaire, aucun jour coché vaut tous les jours, ce qui est le cas le plus fréquent et ne demande donc aucune saisie.

Sur évènement**À l'heure dite**

À gauche, la troisième question se lit à l'envers des autres : un évènement émis par un passage n'a pas le droit de relancer un flux déclenché, et c'est « oui » qui arrête là. À droite, la minute courante sert de clef pour qu'un créneau ne parte pas deux fois.

Trois verrous

L'armement est explicite, workflow par workflow, et ne s'importe jamais. Déclarer un déclencheur ne l'arme pas : il faut le dire une seconde fois, et cette seconde fois porte le mot « écrira » quand le flux contient une écriture. Changer le déclencheur pour « aucun » désarme le flux du même geste.

Un workflow déclenché n'est pas re-déclenchable par les évènements que son propre passage émet. Sans cela, un flux à l'écoute de `workflow.finished` se rappellerait indéfiniment, la nuit, en écrivant dans une base signée.

Un repos de trois secondes sépare deux déclenchements du même flux. Le LIMS ingère fichier par fichier et chacun émet, donc dix fichiers déposés ensemble lanceraient sinon dix passages concurrents sur les mêmes données.

Les échéances manquées ne sont pas rattrapées. L'application est fermée le soir ; au démarrage du lundi, un moteur qui rattraperait lancerait d'un coup les sept passages de la semaine écoulée, tous écrivant dans la base. Le battement est d'une minute, et la minute courante sert de clef pour qu'un même créneau ne parte pas deux fois.

L'alerte, dernière étape d'un workflow sans témoin

Une étape qui échoue écrit dans la console et dans le journal, que personne ne lit au matin. `core.alerter` dépose à la place un message qui survit au rechargement et se représente dans le panneau au démarrage suivant. L'alerte porte le nom du workflow, son déclencheur, et le fait qu'elle vienne ou non d'une simulation, puisque « 3 fichiers rejetés » posé par une simulation ne décrit pas l'état de la base.

Trois niveaux existent, `info`, `attention` et `erreur`. Les deux derniers déclenchent en plus une notification à l'écran quand quelqu'un est là pour la voir, sans remplacer l'alerte persistante, qui est la seule à survivre à la nuit.

11. Composition et itération

La composition évite qu'un workflow de vingt étapes soit recopié en quatre exemplaires parce qu'il diffère d'une seule. Elle rouvre la boucle, que la pile et la profondeur referment.

`core.workflow`

Lance un autre workflow et attend qu'il finisse. Ce qu'il a produit revient sous la clef `sorties` plutôt que fondu dans l'objet de retour : un flux qui déclarerait une sortie nommée `etat` écraserait sinon l'état du passage, et la panne se lirait à l'étape suivante, loin de sa cause. La cible s'écrit indifféremment par identifiant ou par nom, un identifiant comme `wf-lx8k2` n'étant pas ce qu'un utilisateur retient.

`core.pour_chaque`

Répète un workflow sur chaque élément d'une liste. La liste vient d'une étape précédente et se donne entière : la tâche sait retrouver ses lignes, qu'elles s'appellent `lignes`, `rows`, `echantillons`, `trouves` ou `entrees`. Une sortie qui range ses éléments sous un autre nom se donne directement, par exemple `${e1.ids}`, puisqu'un tableau est accepté tel quel.

- Chaque tour lance le workflow avec l'élément en paramètre, sous le nom donné par `variable`, et lui passe aussi `index` et `total`.
- Quand les colonnes sont connues, la ligne arrive nommée. Le corps lit `${param.element.name}` au lieu d'une position, qui se décalerait à la première colonne ajoutée à la requête.
- Des paramètres communs à tous les tours peuvent être passés à part, ce qui évite de les répéter dans le corps.
- Les tours s'enchainent en série. En parallèle, deux tours qui écrivent en même temps dans la même base se marcheraient dessus.
- Le corps est un workflow et non un bloc d'étapes imbriquées. Un bloc aurait demandé un éditeur et une trace à deux niveaux, là où un workflow se relit seul, se lance seul pour l'essayer sur un élément, et se réutilise ailleurs.
- Un échec arrête la boucle, sauf si `continuerSiEchec`. Le message dit quel tour a cédé et combien avaient abouti avant lui, parce qu'une boucle qui n'est jamais partie et une boucle tombée au quarantième élément ne se reprennent pas de la même façon.

Deux plafonds, qui ne disent pas la même chose. Celui que l'auteur règle (`max` , cinquante par défaut) borne son intention. Celui du moteur, cinq cents, borne les dégâts, y compris quand la liste vient d'un `core.sql` dont personne n'a relu le critère. Au delà, les éléments restants sont laissés et `tronque` le signale.

Ce qui referme la boucle

Le moteur tient une pile des flux en vol, racine comprise, et refuse l'appel circulaire, qu'il soit direct ou qu'il passe par un troisième flux. Tant que la pile n'était poussée que par l'appelant imbriqué, un flux s'appelant lui-même passait au premier tour et n'était refusé qu'au second, une fois deux écritures faites.

L'imbrication s'arrête à trois niveaux. Un corps de boucle qui appelle lui-même un flux consomme donc deux de ces trois niveaux.

Gouverner

Ce qu'un flux a le droit de toucher, comment on l'accorde, et ce que le journal en retient.

12. Capacités et consentement

Un document signé dit qui a écrit le workflow. Il ne dit pas ce que ce workflow touche, et c'est l'autre question.

Tant qu'un flux savait lire une base et écrire un CSV, la liste des étapes affichée à l'import tenait lieu de réponse. Maintenant qu'il compose, qu'il itère et qu'il se replie sur un autre flux, cette liste ne se lit plus d'un coup d'oeil, et une étape `core.pour_chaque` ne montre rien de ce que fait son corps de boucle. La question est donc ramenée à une poignée d'accès nommés, calculés en descendant dans les flux appelés, et consentis une fois pour toutes.

Le vocabulaire est celui des plugins. `samples:read`, `files:write` et `ui:notify` sont les mots qu'un plugin déclare déjà dans son manifeste, et que l'utilisateur a déjà vus dans l'écran de consentement des plugins. Une seconde série de noms obligerait à apprendre deux fois la même chose, et à se demander devant `db:write` s'il s'agit de `samples:write`.

Le vocabulaire complet

Capacité	Ce qu'elle accorde	Trace
<code>samples:read</code>	Lire la base : échantillons, analyses, mesures	lecture
<code>samples:write</code>	Écrire dans la base, sous votre nom, dans le registre signé	écrit
<code>files:read</code>	Lire des fichiers, dans les dossiers déclarés	lecture
<code>files:write</code>	Déposer des fichiers sur le disque	écrit
<code>lims:read</code>	Voir les sources d'acquisition et leur journal	lecture
<code>lims:write</code>	Ingérer une acquisition dans la base	écrit
<code>methods:write</code>	Ranger des méthodes dans la bibliothèque	écrit
<code>campaigns:read</code>	Voir les campagnes déclarées et leurs chaines d'aliquotes	lecture
<code>campaigns:write</code>	Écrire dans le descripteur d'une campagne, qui fait foi pour sa provenance	écrit
<code>sites:read</code>	Lire les sites de prospection, leurs forages et leurs interprétations	lecture
<code>sites:solve</code>	Faire tourner les solveurs d'un site : écoulement, gradient, profil	lecture
<code>nexus:run</code>	Interroger le moteur Nexus et ses bases de référence	lecture
<code>notebook:run</code>	Exécuter un carnet enregistré : le script fait ce que son noyau permet	écrit

<code>report:write</code>	Produire un rapport ou un fichier ISOF	écrit
<code>ui:notify</code>	Déposer une alerte que vous verrez au prochain démarrage	lecture
<code>ui:command</code>	Ouvrir n'importe quelle fenêtre d'IsoFind	écrit
<code>workflow:call</code>	Lancer d'autres workflows, et répéter l'un d'eux sur une liste	écrit

Le drapeau `écrit` ne signifie pas que la capacité touche la base, mais qu'elle laisse une trace en dehors du passage. C'est lui qui décide de la case pré-cochée ou non, et il correspond à la seule distinction qu'on puisse demander à quelqu'un de faire en trois secondes. `sites:solve` ne modifie rien mais coute du temps machine par dizaines de secondes, et un consentement qui ne distinguerait pas la lecture d'un site du lancement d'un couplage ne dirait pas grand chose de ce qu'on accorde.

Trois tâches Nexus font exception à l'apparence de leur nom : celles qui cherchent dans les échantillons relèvent de `samples:read` et non de `nexus:run`, sans quoi un flux à qui l'on n'a pas accordé la lecture de la base y accéderait par ce chemin.

De la tâche à la capacité

1. La tâche déclare son champ `capacite`, qui l'emporte. C'est ainsi qu'un plugin range la sienne dans le vocabulaire commun.
2. Sinon, la table du noyau, rassemblée dans un seul fichier pour qu'on puisse répondre à « que peut toucher un workflow » sans en ouvrir quinze. Les tâches qui ne demandent rien y figurent avec une capacité nulle, pour qu'on lise que l'omission est un choix.
3. Sinon, une capacité construite sur la source de la tâche, `<source>:read` ou `<source>:write` selon son drapeau. Elle apparaîtra telle quelle dans l'écran, et une ligne obscure vaut mieux qu'un accès accordé sans le dire.

Une capacité hors vocabulaire est traitée comme écrivante : dans le doute, une case de trop à cocher coute moins cher qu'un accès accordé en silence.

Ce que le calcul ne peut pas savoir

Le parcours descend dans les flux appelés et dans les replis, jusqu'à quatre niveaux. Un repli est un workflow lui aussi, et ce qu'il touche est accordé au même titre, sinon il suffirait de faire échouer une étape pour ouvrir une porte que personne n'a regardée. Deux angles morts subsistent, et l'écran les annonce :

- **Flux inconnus.** Le workflow en appelle d'autres qui n'existent pas ici. Ces étapes échoueront, et leurs accès n'ont pas pu être calculés.
- **Cible indécidable.** Un appel désigne sa cible par une substitution, connue seulement à l'exécution. Le moteur refusera alors toute étape sortant des accès accordés.

Le faire savoir importe autant que le calcul, car un écran de permissions muet sur ses angles morts donne la certitude d'avoir tout vu.

Qui est contraint, et jusqu'où

Un workflow écrit ici n'a rien à demander, puisque celui qui le lance est celui qui l'a écrit. Un workflow importé porte une origine, et le moteur ne le laisse alors exécuter que les tâches dont la capacité a été accordée.

La contrainte se transmet à tout l'arbre d'appel, sans quoi un flux importé s'accorderait par la composition ce qu'on ne lui a pas donné. Un flux local appelé par un flux importé hérite donc de la même liste.

- À l'import, les lectures sont pré-cochées et les écritures ne le sont pas. Comme pour l'armement, ce qui engage se dit deux fois.
- L'écran de consentement n'est pas une fenêtre à lui seul : c'est un bloc posé dans celle de l'import, car empiler une seconde fenêtre ferait perdre de vue ce à quoi on consent.
- Les accès se révisent après coup, sans réimporter. Retirer trois semaines plus tard un accès accordé devant un écran qu'on lisait pour la première fois est un geste normal ; l'interdire pousserait à supprimer le workflow et à le réimporter, ce qui lui donnerait un identifiant neuf et perdrait sa trace.
- La capacité est vérifiée avant la substitution des arguments, donc avant qu'aucune préparation n'ait lieu.
- La liste des accès sous lesquels un passage a tourné est conservée avec lui dans la trace. C'est la première chose qu'on regarde devant une modification faite par une automatisation reçue d'ailleurs.
- Quand le registre des capacités n'est pas chargé sur la page, les lectures passent et les écritures sont retenues. Un module de sécurité absent ne doit pas se traduire par une autorisation, ni bloquer une page qui se contente de consulter.

13. Journal des accès

Qui accède aux échantillons, et sous quel titre. Le journal est écrit par le serveur ; la page lui fournit le nom.

Chaque fichier de tâches appelle le réseau à sa façon. Poser l'en-tête dans les quinze demanderait de maintenir quinze endroits qui doivent rester d'accord, et le premier fichier ajouté ensuite l'oublierait sans que rien ne le signale, alors qu'un accès non déclaré ressemble à un accès qui n'a pas eu lieu. Une seule enveloppe est donc posée, et la portée qui dit à quelle tâche attribuer la requête vient du registre. Une tâche apportée par un plugin passe par la même porte, donc elle est couverte sans que son auteur ait à s'en occuper.

En-tête	Contenu
X-IsoFind-Agent-Kind	workflow, plugin, ou le genre déclaré par l'appelant
X-IsoFind-Agent-Id	Identifiant du flux, qui sert de clef de jointure
X-IsoFind-Agent-Nom	Nom lisible, qui peut changer sans rien casser
X-IsoFind-Capability	Capacité de la tâche en cours
X-IsoFind-Run	Référence du passage
X-IsoFind-Task	Identifiant de la tâche

L'identifiant voyage, le nom l'accompagne. Classé sous le nom, le journal perdrait le lien au premier workflow renommé, et ses lignes anciennes désigneraient un flux qui n'existe plus sous ce libellé.

L'attribution est déclarative, et elle signale ses doutes

Le moteur tourne dans la page, donc ces en-têtes valent comme déclaration et non comme preuve, et le serveur les traite ainsi. Quand deux passages simultanés entrelacent leurs requêtes, l'enveloppe pose un marqueur d'ambiguïté au lieu d'attribuer au hasard, et l'entrée du journal porte cette réserve.

- **Rien n'est refusé ici.** Le refus des accès révoqués revient au serveur, seul à savoir ce qui a été révoqué, et le consentement des capacités reste l'affaire du moteur.
- **Les requêtes sortantes ne reçoivent rien.** Ces en-têtes déclencheraient un contrôle préalable CORS sur les tuiles de carte et le géocodage, qui le refuseraient, et la déclaration casserait ce qu'elle observe.
- **La portée tient jusqu'à ce que la promesse retombe,** pas seulement pendant la partie synchrone. Une tâche qui enchaîne deux requêtes perdrait sinon son nom dès la seconde.
- **Les consommateurs hors workflow se nomment aussi,** par une portée ouverte explicitement : le pont des plugins, qui est le seul à savoir quel plugin il sert, et le carnet.
- **Un estampillage impossible n'empêche pas la requête.** L'appel part sans nom, et le serveur le consigne comme non déclaré.

14. Le panneau et les commandes

Le panneau Workflows liste, lance, et donne à voir ce qui s'est passé. Il occupe le dock de gauche et se compose de deux onglets, les workflows et les exécutions.

Chercher dans une liste longue

Le filtre porte sur le nom, en ignorant accents et casse, et reconnaît quelques mots d'état, parce que les deux questions qu'on pose vraiment à une liste longue ne se posent pas par un nom.

Onglet	Mots reconnus
Workflows	<code>arme</code> , <code>importe</code> , <code>ecrit</code> et leurs équivalents anglais
Exécutions	Les codes d'état tels quels : <code>ok</code> , <code>echec</code> , <code>annule</code> , et <code>simulation</code>

Le filtre se vide en changeant d'onglet, sans quoi un « echec » gardé en passant sur les workflows y masquerait tout sans qu'on comprenne pourquoi. Échap le vide sans fermer le panneau. Le compteur n'apparaît que lorsqu'un filtre porte, sous la forme « 4 sur 12 ».

Les flux armés remontent en tête de liste : ce sont les seuls qui partent sans qu'on les regarde, donc les seuls dont l'état compte quand on ouvre le panneau sans raison particulière.

Une ligne, un geste

Seul « Lancer » reste sur la ligne. Simuler, exporter, dupliquer et supprimer vivent dans le menu contextuel, qui sait marquer ce qui écrit et griser ce qui n'est pas possible. Quatre liens permanents par ligne faisaient quatre cibles à viser pour des gestes dont un seul est fréquent.

Un flux importé porte une marque, avec le nom de son signataire et l'état de sa signature, et donne accès à la révision de ses accès. L'import lui-même se fait par la flèche en tête de panneau, quand le module de documents signés est présent.

Lire un passage

- Le sommaire se replie, et la ligne entière sert de bouton.
- Chaque étape affiche son état, sa durée, et son message quand il y en a un.
- Les sorties se déplient sous un résumé qui vaut à lui seul : « 11 lignes » dit ce qu'il y a à voir, là où « sorties » oblige à ouvrir pour savoir s'il y avait quelque chose.
- Un tableau se copie au format tableur ou s'exporte en CSV, avec toutes ses lignes et non les cinquante affichées. Le nom des colonnes reste celui de la base dans le fichier, même si le tableau les affiche en clair, pour qu'un export puisse se recoller à une requête.
- Une étape simulée montre les arguments qu'elle aurait reçus, et non un résultat.
- Une trace relue après rechargement affiche l'ombre des sorties, « 238 lignes, données non conservées », plutôt qu'une étape muette.
- Les tours d'une boucle sont repliés sous leur parent, qui annonce leur nombre. Un passage dont le parent est sorti de la trace remonte à la racine en le signalant, plutôt que de disparaître.

Le bandeau d'alertes. Les alertes non lues s'affichent en tête du panneau. La croix les masque pour la séance en cours sans les marquer lues, et une alerte neuve rouvre le bandeau : l'avoir écarté à dix heures ne doit pas masquer l'échec de onze heures. « Marquer comme lues » reste le seul geste qui les efface.

Les commandes

Aucune touche n'est écoutée par le panneau. Ce sont des commandes, donc elles entrent dans la palette, dans le panneau Actions et dans la recherche universelle, et chacune peut recevoir le raccourci que l'utilisateur veut.

Commande	Effet
<code>core.workflows.open</code>	Ouvrir le panneau
<code>core.workflows.new</code>	Nouveau workflow
<code>core.workflows.run</code>	Lancer le workflow choisi dans la barre de titre
<code>core.workflows.dryrun</code>	Le simuler

<code>core.workflows.debug</code>	Le déboguer, en ouvrant le panneau de débogage
<code>core.workflows.replay</code>	Rejouer le dernier passage racine, avec ses paramètres
<code>core.workflows.stop</code>	Arrêter le passage en cours
<code>core.workflows.filter</code>	Ouvrir le panneau et viser le champ de filtre
<code>workflow.<id></code>	Une par workflow enregistré

Le rejeu reprend les paramètres du passage et non les défauts du flux, et il reste en simulation si le passage rejoué en était une. Rejouer avec d'autres valeurs reviendrait à faire autre chose en prétendant reproduire.

Référence

Le catalogue des tâches, les workflows types, le dépannage, les bornes et l'ajout d'une tâche.

15. Catalogue des tâches

Les soixante et une tâches livrées avec le noyau, groupées par ce qu'elles touchent. Cette liste tient lieu d'architecture : une tâche absente est une chose que personne ne pourra automatiser, et une tâche de trop est une porte qu'il faudra défendre.

Dans chaque ligne, un argument suivi d'une étoile est requis, son type suit en gris, et une valeur par défaut apparait en vert. **écrit** marque une tâche qui modifie quelque chose, donc absente du mode conversation et simulée à vide. **hors modèle** marque une tâche que l'assistant ne voit dans aucun mode.

Sans accès **null**

Elles ne sortent pas du passage : rien à demander, et l'omission est un choix.

Tâche	Entrées	Sorties	Drapeaux
core.calcul Appliquer une formule géochimique	formule* texte parametres* texte element texte sql sql corrigees boolean = false	resultat nombre incertitude nombre lignes liste colonnes liste	lecture
core.formules Catalogue des formules géochimiques	domaine texte formule texte	formules liste constantes liste nombre nombre	lecture
core.condition Condition	valeur* texte operateur* texte = > reference* texte	vrai boolean	lecture hors modèle
core.carnet_noyaux Noyaux de carnet disponibles	aucun	noyaux liste nombre nombre	lecture
core.journal Écrire au journal	message* texte	aucune	lecture hors modèle

Lire la base **samples : read**

Tâche	Entrées	Sorties	Drapeaux
core.analyses Analyses par élément	element* texte limite nombre = 500 corrigees boolean = false	lignes liste colonnes liste nombre nombre	lecture
core.methodes Bibliothèque de méthodes	type texte element texte	methodes liste nombre nombre	lecture
core.carte Carte d'un échantillon	echantillon* texte	carte objet	lecture
core.correspondance	echantillon* texte	lignes liste	lecture

Correspondance géographique	seuil nombre	colonnes liste nombre nombre	
core.echantillon_detail Détail d'un échantillon	nom* texte corrigees boolean = false	echantillon objet analyses liste	lecture
core.inventaire Inventaire de la base	par texte = element	lignes liste colonnes liste graphique objet nombre nombre	lecture
core.rapport_modeles Modèles de rapport	aucun	lignes liste colonnes liste nombre nombre	lecture
core.nexus_authenticite Nexus : contrôle d'authenticité	element* texte = Sb delta* nombre incertitude nombre = 0.05 origine_declaree texte categorie texte seuil_confiance nombre = 0.8	score nombre verdict texte references liste	lecture
core.nexus_correspondance Nexus : correspondance rapide	element* texte = Sb delta* nombre incertitude nombre = 0.05 gabarit texte materiau texte seuil_sigma nombre = 3 conditions objet molecule texte	matches liste transformed_signature objet steps liste	lecture
core.nexus_tracabilite Nexus : remonter un procédé	element* texte = Sb delta* nombre incertitude nombre = 0.05 procede texte materiau texte seuil_sigma nombre = 3 molecule texte	matches liste inversion_step objet method texte	lecture
core.sql Requete SQL	base* texte = isofind sql* sql	lignes liste colonnes liste nombre nombre	lecture
core.schema Schéma de la base	base texte = isofind table texte colonne texte	bases liste tables liste	lecture
core.resume Statistiques récapitulatives	element texte sql sql par texte sur texte corrigees boolean = false	lignes liste colonnes liste nombre nombre	lecture
core.echantillons Sélectionner des échantillons	filtre texte limite nombre = 100	ids liste noms liste nombre nombre	lecture
core.graphique Tracer un graphique	sql sql element texte type texte = barres x texte y texte erreur texte	graphique objet lignes liste colonnes liste nombre nombre	lecture

	couleur texte agregger texte par texte titre texte titre_x texte titre_y texte corrigees boolean = false		
core.derivees Valeurs dérivées d'un échantillon	sample_id* nombre actives boolean = false	derivees liste nombre nombre	lecture
core.verifier_chaine Vérifier la chaîne d'un échantillon	id* nombre	valide boolean entrees nombre raison texte	lecture

Campagnes et chaines d'aliquotes, lecture campaigns:read

Tâche	Entrées	Sorties	Drapeaux
core.campagnes Campagnes déclarées	nom texte projet_id texte	campagnes liste nombre nombre	lecture
core.provenance_comparer Ce qui diffère entre deux lignées	a* objet b* objet	differences liste etapes_identiques liste ecart_valeurs objet	lecture
core.chaine Chaîne d'aliquotes	path* texte	chaine objet resume objet orphelins liste integrite_resume objet	lecture
core.provenance_resoudre Retrouver un échantillon dans les campagnes	terme* texte path texte	trouves liste terme texte	lecture
core.provenance_recit Tout ce qui a conduit à ce nœud	path* texte noeud* texte	etapes liste valeur texte repris_dans liste noeud texte	lecture

Sites de prospection, lecture sites:read

Tâche	Entrées	Sorties	Drapeaux
core.sites Sites de prospection	nom texte	sites liste nombre nombre	lecture
core.site_matrice Unités et ratios d'un site	site_id* nombre ratios texte interpretation_id nombre	units liste ratios liste	lecture

Solveurs de site sites:solve

Elles ne modifient rien, mais coutent du temps machine par dizaines de secondes.

Tâche	Entrées	Sorties	Drapeaux
core.gradient_isotopique Gradient isotopique interpolé	site_id* nombre forages* liste	grid objet ratio_key texte	lecture

```
ratio* texte
interpretation_id nombre
nx nombre = 60
nz nombre = 40
anisotropie nombre = 5
puissance nombre = 2
distance_max nombre = 30
contact_permeabilite nombre
= 0
```

core.profil_isotopique Profil isotopique d'un transect	site_id* nombre forages* liste ratio texte interpretation_id nombre	boreholes liste layer_stats liste	lecture
--	--	--------------------------------------	---------

core.flux_simuler Simuler un écoulement	site_id* nombre forages* liste interpretation_id nombre nx nombre = 80 nz nombre = 60 vadose booleen = false recharge nombre = 0	head objet velocity objet grid objet	lecture
---	--	--	---------

Moteur Nexus **nexus:run**

Tâche	Entrées	Sorties	Drapeaux
core.nexus_demixage Nexus : démixage de sources	element* texte = Sb delta_melange* nombre incertitude nombre = 0.05 sources* liste methode texte = nnls	fractions liste method texte	lecture
core.nexus_gabarits Nexus : gabarits de fractionnement	aucun	templates liste	lecture
core.nexus_processus Nexus : identifier un processus	element* texte = Sb delta_initial* nombre delta_final* nombre incertitude_initiale nombre = 0.05 incertitude_finale nombre = 0.05 conditions objet	processes liste delta_change nombre	lecture
core.nexus_serie Nexus : suivre une série	element* texte = Sb mesures* liste site texte fenetre nombre = 3 sensibilite texte = medium	breakpoints liste trend objet statistics objet	lecture

Lire des fichiers **files:read**

Bornées aux racines déclarées, chemins résolus avant comparaison.

Tâche	Entrées	Sorties	Drapeaux
core.fichier_lire Lire un fichier	chemin* texte	contenu texte lignes nombre	lecture

<code>core.fichier_lister</code>	chemin texte	entrees liste nombre nombre	lecture
Lister un dossier			

Acquisition, lecture `lims:read`

Tâche	Entrées	Sorties	Drapeaux
<code>core.lims_evenements</code> Journal d'acquisition	statut texte limite nombre = 40	evenements liste nombre nombre erreurs nombre	lecture
<code>core.lims_sources</code> Sources d'acquisition	aucun	sources liste nombre nombre actives nombre	lecture
<code>core.lims_sonder</code> Tester une source	source* texte	ok boolean message texte exemples liste	lecture

Écrire dans la base `samples:write`

Tâche	Entrées	Sorties	Drapeaux
<code>core.echantillon_creer</code> Créer un échantillon	nom* texte materiau texte secteur texte lieu texte date texte description texte classification texte projet texte latitude nombre longitude nombre isotopes liste	sample_id nombre isotopes nombre	écrit hors modèle
<code>core.geochimie_enregistrer</code> Enregistrer des concentrations	sample_id* nombre donnees* liste remplacer boolean = false	enregistrees nombre ids liste	écrit hors modèle
<code>core.physico_enregistrer</code> Enregistrer un paramètre physico-chimique	sample_id* nombre parametre* texte valeur* nombre incertitude nombre methode texte mesure_le texte notes texte	id nombre parametre texte	écrit hors modèle
<code>core.rendement_enregistrer</code> Enregistrer un rendement de purification	sample_id* nombre element* texte rendement* nombre opérateur texte methode texte	element texte rendement nombre	écrit hors modèle
<code>core.enregistrer_derivee</code> Enregistrer une valeur dérivée	donnees* objet motif* texte nature texte = correction lignes_max nombre = 100 colonne texte	enregistrees nombre ignorees nombre sample_ids liste	écrit hors modèle
<code>core.echantillon_modifier</code> Modifier un échantillon	sample_id* nombre champ texte	sample_id nombre modifies liste	écrit

valeur texte
champs objet

hors modèle

Campagnes, écriture `campaigns:write`

Le descripteur d'une campagne fait foi pour sa provenance.

Tâche	Entrées	Sorties	Drapeaux
<code>core.chaine_noeud</code> Ajouter un nœud à la chaîne	<code>path*</code> texte <code>nom*</code> texte <code>type</code> texte = <code>aliquote</code> <code>quantite</code> texte <code>valeur</code> texte <code>reference</code> texte <code>fichiers</code> liste <code>note</code> texte	<code>noeud</code> objet <code>resume</code> objet	<code>écrit</code> <code>hors modèle</code>
<code>core.campagne_declarer</code> Déclarer une campagne	<code>path*</code> texte <code>nom</code> texte <code>projet_id</code> texte	<code>racine</code> texte <code>fichiers</code> nombre <code>signalements</code> liste	<code>écrit</code> <code>hors modèle</code>
<code>core.chaine_rattacher</code> Rattacher un fichier à la chaîne	<code>path*</code> texte <code>id*</code> texte <code>chemin*</code> texte <code>repere</code> texte <code>detacher</code> booleen = <code>false</code>	<code>fait</code> booleen <code>chemin</code> texte	<code>écrit</code> <code>hors modèle</code>
<code>core.chaine_arete</code> Relier deux nœuds	<code>path*</code> texte <code>de*</code> texte <code>vers*</code> texte <code>operation</code> texte = <code>aliquotage</code> <code>protocole</code> texte <code>fichiers</code> liste <code>note</code> texte	<code>arete</code> objet <code>resume</code> objet	<code>écrit</code> <code>hors modèle</code>

Bibliothèque de méthodes `methods:write`

Tâche	Entrées	Sorties	Drapeaux
<code>core.methode_enregistrer</code> Enregistrer une méthode	<code>cle*</code> texte <code>nom*</code> texte <code>type*</code> texte = <code>purification</code> <code>element</code> texte <code>protocole</code> objet <code>materiel</code> texte <code>laboratoire</code> texte <code>auteur</code> texte <code>version</code> texte = <code>1.0</code> <code>doi</code> texte <code>reference</code> texte	<code>id</code> nombre <code>cle</code> texte <code>cree</code> booleen	<code>écrit</code> <code>hors modèle</code>

Écrire des fichiers `files:write`

Deux formats, et l'extension est imposée par le format.

Tâche	Entrées	Sorties	Drapeaux
<code>core.exporter_fichier</code>	<code>donnees*</code> objet	<code>fichier</code> texte	

Exporter dans un fichier	fichier* texte format texte = csv horodater boolean = true	lignes nombre octets nombre	écrit hors modèle
--------------------------	--	--------------------------------	----------------------

Acquisition, ingestion lims:write

Tâche	Entrées	Sorties	Drapeaux
core.lims_tirer Tirer une acquisition	source texte	fichiers nombre echantillons nombre lignes nombre neuf boolean	écrit

Rapports et ISOF report:write

Tâche	Entrées	Sorties	Drapeaux
core.export_isof Exporter en ISOF	ids* liste signer boolean = true provenance texte = complet	document objet echantillons nombre	écrit
core.rapport Générer un rapport	modele* nombre echantillons texte titre texte projet texte	rapport objet	écrit

Carnets notebook:run

Tâche	Entrées	Sorties	Drapeaux
core.carnet_executer Exécuter un carnet	chemin* texte noyau texte continuerSiEchec boolean = false	cellules nombre reussies nombre sortie texte resultat objet	écrit hors modèle

Alerter ui:notify

Tâche	Entrées	Sorties	Drapeaux
core.alerter Alerter	message* texte niveau texte = info	posee boolean	lecture

Ouvrir une fenêtre ui:command

Tâche	Entrées	Sorties	Drapeaux
core.commande Exécuter une commande	id* commande	aucune	écrit

Composer et répéter workflow:call

Tâche	Entrées	Sorties	Drapeaux
core.workflow	workflow* workflow	etat texte	écrit

Lancer un workflow	parametres objet	etapes nombre duree nombre sorties objet
core.pour_chaque Pour chaque élément	liste* objet workflow* workflow variable texte = element max nombre = 50 continuerSiEchec boolean = false parametres objet	tours nombre reussis nombre echoues nombre resultats liste tronque boolean

écrit

La table des capacités mentionne aussi `core.contexte`, `core.doc`, `core.workflow_detail`, `core.workflow_liste`, `core.workflow_passages`, `core.workflow_simuler`, déclarées hors des fichiers de tâches ci-dessus. Elles n'ajoutent aucun accès, sauf `core.workflow_simuler`, rangée sous `workflow:call`.

16. Ce que voit l'assistant

Deux drapeaux découpent trois surfaces. Ils ne servent pas à documenter : ils déterminent ce que le modèle peut appeler.

Surface	Ce qu'elle reçoit
Mode conversation	Les tâches où <code>ecrit</code> vaut faux et où <code>outil</code> ne vaut pas faux.
Mode agent	Les tâches où <code>outil</code> ne vaut pas faux, écritures comprises.
Workflows seulement	Les tâches marquées <code>outil: false</code> , éditables dans un plan relu et adopté par un humain, invisibles au modèle.

Retirer un outil ne se discute pas, le demander se discute

Dire au modèle de ne rien modifier reviendrait à formuler une demande, qu'un modèle qui hallucine, ou qu'une donnée empoisonnée manipule, pourrait ignorer. Lui retirer les outils qui écrivent le met dans une situation où la question ne se pose plus.

Une tâche dont le drapeau est absent est traitée comme écrivante, faute de quoi un plugin qui oublie de le déclarer verrait sa tâche exposée au mode le moins surveillé.

`outil: false` couvre deux situations différentes. Les pièces d'enchaînement, comme `core.condition` ou `core.journal`, ne répondent à aucune question qu'un utilisateur pose, et un modèle qui les verrait les essaierait, puisqu'il essaie ce qu'on lui montre. Les écritures du domaine, comme `core.echantillon_creer`, `core.enregistrer_derivee` ou `core.exporter_fichier`, sont retirées pour une autre raison : écrire dans une base signée parce qu'un humain a relu un plan et écrire parce qu'un modèle l'a décidé au milieu d'une phrase sont deux choses différentes, et c'est ce que le mode plan sert à séparer.

Deux tâches sont marquées comme écrivantes sans écrire elles-mêmes : `core.workflow` et `core.pour_chaque`. Le flux qu'elles lancent peut écrire, et elles ne savent pas ce qu'il fait. Les déclarer en lecture ouvrirait, en mode conversation, un chemin vers toutes les écritures que le modèle ne peut pas appeler directement, multiplié par la longueur d'une liste dans le second cas.

La notice

Chaque tâche exposée au modèle porte une notice bilingue qui dit quand l'appeler, quand s'en abstenir, et comment lire une réponse vide. Quatre consignes y reviennent :

- **Ne rien trouver est une réponse.** Le dire évite que le modèle aille chercher ailleurs une explication que rien n'appuie.
- **Les identifiants ne s'inventent pas.** La notice nomme la tâche qui les rend et demande de les reprendre tels quels : un identifiant de noeud comme `n17` n'a aucun sens hors de sa campagne.
- **Le calcul revient au code.** Le modèle choisit la formule, le code l'applique, propage l'incertitude et rend la référence. Une formule appliquée de tête est une formule dont personne ne connaîtra la version, et « Rayleigh » en désigne quatre.
- **Un échec doit s'apprendre.** Les tâches font remonter le message réel du serveur plutôt qu'un code : avec « no such column: element », le modèle corrige tout seul ; avec « HTTP 500 », il redemande la même chose ou abandonne. La tâche SQL va plus loin et rend la liste des tables réelles quand une requête échoue, ce qui redresse la faute la plus fréquente, celle d'inventer un nom.

17. Quatre workflows types

Quatre formes qui reviennent, écrites ici dans la forme enregistrée plutôt que décrites. Les noms de colonnes et les valeurs numériques sont à remplacer par ceux de votre base.

Acquisition de nuit, avec compte rendu au matin

La sonde en tête est ce qui distingue un test utile d'un voyant vert : ingérer depuis une source injoignable ne produit rien, mais ingérer depuis un dossier à moitié synchronisé produit des données fausses.

```
{
  nom: "Acquisition de nuit",
  declencheur: { type: "horaire", heure: "03:00", jours: [] },
  arme: true,
  etapes: [
    { id: "sonde", tache: "core.lims_sonder",
      args: { source: "neptune" } },
    { id: "tirage", tache: "core.lims_tirer",
      args: { source: "neptune" }, si: "sonde.ok" },
    { id: "rejets", tache: "core.lims_evenements",
      args: { statut: "erreur" }, continuerSiEchec: true },
    { id: "alerte", tache: "core.alerter",
      args: { message: "${rejets.erreurs} fichier(s) rejeté(s), "
```

```

        + "${tirage.echantillons} échantillon(s) entrés",
        niveau: "attention" },
    si: "rejets.erreurs" }
  ]
}

```

Le tirage est idempotent : relancer n'ingère rien deux fois. Zéro partout signifie qu'il n'y avait rien de neuf, et non que l'acquisition a échoué, ce qui est précisément la raison de l'étape **rejets**.

Correction paramétrée, avec écriture dérivée

Le calcul se fait en code, avec propagation d'incertitude, et la correction s'ajoute à la mesure au lieu de la remplacer. Le motif est obligatoire, et il part avec la valeur dans le registre.

```

{
  nom: "Correction de blanc",
  parametres: [
    { nom: "element",      default: "Sr" },
    { nom: "delta_blanc",  default: "-0.8" },
    { nom: "q_blanc",      default: "0.12" }
  ],
  etapes: [
    { id: "calcul", tache: "core.calcul",
      args: {
        formule: "correction_blanc",          // identifiant rendu par core.formules
        element: "${param.element}",          // la source, donc les colonnes @
        parametres: "delta_mesure=@ratio_value; q_mesure=@amount; "
                    + "delta_blanc=${param.delta_blanc}; q_blanc=${param.q_blanc}"
      } },
    { id: "pose", tache: "core.enregistrer_derivee",
      args: { donnees: "${calcul}",            // la sortie entiere, pas une colonne
        motif: "Blanc de session, ${param.element}",
        nature: "blanc", lignes_max: 200 } }
  ],
  sorties: [ { nom: "corrigees", valeur: "${pose.enregistrees}" } ]
}

```

Les noms précédés de **@** désignent des colonnes de la source et doivent y exister. La sortie du calcul porte l'identifiant de mesure, la valeur corrigée et son incertitude, ce qui permet à l'étape suivante de rattacher chaque valeur à la bonne mesure sans que le flux ait à nommer une colonne.

Répéter sur une sélection

Le corps est un workflow ordinaire, qu'on peut lancer seul sur un échantillon pour l'essayer avant de le mettre en boucle.

```
// Le flux appelant
{
  nom: "Contrôler une sélection",
  parametres: [ { nom: "filtre", default: "sediment" } ],
  etapes: [
    { id: "liste", tache: "core.echantillons",
      args: { filtre: "${param.filtre}", limite: 200 } },

    { id: "boucle", tache: "core.pour_chaque",
      args: { liste: "${liste.ids}", // un tableau, accepte tel quel
        workflow: "Contrôler un échantillon",
        variable: "id", max: 200, continuerSiEchec: true } },

    { id: "bilan", tache: "core.alerter",
      args: { message: "${boucle.echoues} contrôle(s) en échec sur ${boucle.tours}" },
      si: "boucle.echoues" }
  ]
}

// Le corps de boucle
{
  nom: "Contrôler un échantillon",
  etapes: [
    { id: "chaine", tache: "core.verifier_chaine",
      args: { id: "${param.id}" } }
  ],
  sorties: [
    { nom: "valide", valeur: "${chaine.valide}" },
    { nom: "raison", valeur: "${chaine.raison}" }
  ]
}
```

Un contrôle négatif n'est pas un échec. `core.verifier_chaine` rend `valide: false` sans lever d'erreur, donc `echoues` ne compte que les tours qui ont vraiment cédé. Pour compter les chaînes invalides, il faut lire les `resultats` de la boucle, qui portent les sorties déclarées de chaque tour.

Récapitulatif hebdomadaire déposé sur le disque

Un workflow qui calcule sans rien poser calcule pour personne : le tableau vit dans la mémoire d'une page qui sera fermée.

```
{
  nom: "Récapitulatif hebdomadaire",
  declencheur: { type: "horaire", heure: "07:00", jours: [1] },
  arme: true,
  etapes: [
    { id: "resume", tache: "core.resume",
      args: { element: "Sr", par: "material_type" } },

    { id: "fichier", tache: "core.exporter_fichier",
```

```

    args: { donnees: "${resume}", fichier: "rapports/recap_sr",
            format: "csv", horodater: true } },

    { id: "dire", tache: "core.alerter",
      args: { message: "Récapitulatif écrit : ${fichier.fichier}" } }
  ]
}

```

L'extension est imposée par le format : un nom terminé par autre chose est reconstruit, le dossier restant intact. L'horodatage évite qu'un passage écrase celui de la veille, et se coupe explicitement quand un tiers relit toujours le même fichier au même endroit. Un critère qui ne trouve rien produit un fichier avec ses seules entêtes, parce qu'un fichier absent ne se distingue pas d'une panne au matin.

18. Dépannage

Les pannes courantes, et par où commencer. La plupart se voient en une minute avec le débogueur, en s'arrêtant sur l'étape qui précède celle qui échoue.

Symptôme	Cause probable	Où regarder
Un argument arrive vide, ou la tâche se plaint d'une donnée absente	La référence désigne une étape sans identifiant, ou une sortie qui ne porte pas ce nom	Le champ d'évaluation du débogueur, à l'arrêt sur cette étape. Le catalogue donne le nom exact des sorties.
Étape <i>absente</i>	Tâche non déclarée sur ce poste : plugin absent, ou module d'édition non monté	La liste des tâches dans l'éditeur. Les tâches de campagne, par exemple, ne se déclarent que si l'édition monte le module.
Étape <i>refusée</i>	Capacité non accordée à un flux importé, ou capacité d'une tâche modifiée depuis l'import	La révision des accès, dans le menu contextuel du flux.
La boucle laisse des éléments de côté	Plafond atteint, réglé ou celui du moteur	La sortie tronque de l'étape.
Le workflow déclenché ne part pas	Non armé, condition fausse, repos de trois secondes, passage déjà déclenché en cours, ou session de débogage ouverte qui occupe le moteur	L'onglet Exécutions, et la case d'armement dans l'éditeur.
Un horaire n'a pas été honoré	Application fermée à l'heure dite, et les échéances manquées ne sont pas rattrapées	Rien à corriger : c'est le comportement voulu.
Les sorties ont disparu après un rechargement	Elles ne sont pas persistées, seul leur contour l'est	La ligne « données non conservées » sous l'étape. Relancer les rend.
La simulation échoue là où l'exécution réelle passerait	Une lecture dépend d'une écriture précédente, qui a été sautée et n'a rien rendu	Attendu. Vérifier les étapes de lecture séparément.
« Appel circulaire » ou « imbrication trop	Un flux s'appelle lui-même, directement ou par un troisième, ou la chaîne dépasse trois niveaux	Le message nomme le flux fautif.

profonde »

Le journal des accès porte une réserve d'ambiguïté	Deux passages ont entrelacé leurs requêtes	Normal et signalé comme tel ; l'attribution n'est pas fiable pour ces lignes.
Les alertes reviennent après avoir été écartées	La croix masque pour la séance, elle ne marque pas lu	« Marquer comme lues » dans le bandeau.

Le réflexe utile. Poser un point d'arrêt sur l'étape suivante plutôt que sur l'étape qui échoue : on voit alors ce que la première a réellement produit, et non ce qu'on croyait qu'elle produirait.

19. Bornes et persistance

Les constantes du moteur

Borne	Valeur	Raison
Passages conservés	40	Trace persistée, allégée. Dix seulement si le quota est plein.
File d'attente	20	Une file sans borne serait une fuite de mémoire.
Imbrication	3	Profondeur d'appel entre workflows.
Descente du calcul de capacités	4	Parcours des flux appelés et des replis.
Tours d'une boucle	50 / 500	Plafond réglé par l'auteur, plafond du moteur.
Repos entre deux déclenchements	3 s	Dix fichiers déposés ensemble ne font pas dix passages.
Battement horaire	60 s	La minute courante sert de clef.
Inactivité en débogage	15 min	Un passage en pause occupe la file.
Alertes conservées	50	Elles restent jusqu'à lecture.

Ce qui survit à la page

Clef	Contenu
<code>isofind_workflows</code>	Les définitions
<code>isofind_workflow_passages</code>	La trace, allégée
<code>isofind_workflow_points</code>	Les points d'arrêt, hors des flux

La trace garde le contour des sorties, pas les sorties

Une sortie de `core.sql` peut porter dix mille valeurs isotopiques. Les écrire dans le stockage local saturerait le quota et déposerait des mesures dans un endroit que rien ne compartimente. Ce qui reste est ce dont un diagnostic a besoin : quelle étape, quel état, quel message, combien de temps.

Le nombre de lignes et le nom des colonnes sont conservés, puisque ce sont des noms et non des mesures. Sans eux, un passage relu affiche une étape muette et l'on croit que le workflow n'a rien produit ; avec eux, on lit « 238 lignes, non conservées », et l'on sait qu'il n'y a qu'à relancer.

Les paramètres et la liste des accès accordés survivent également : ils pèsent quelques valeurs, et ce sont eux qui permettent de rejouer un passage à l'identique et de savoir sous quels droits il a tourné.

Une trace relue porte une marque : un passage frais et un passage relu se ressemblent trait pour trait, et l'un a ses données quand l'autre ne les a plus. Sans cette marque, le panneau afficherait « ce workflow n'a rien produit » là où la vérité est « ses sorties ne sont plus en mémoire », ce qui accuse le workflow d'un défaut qui n'est pas le sien.

Évènements émis

Évènement	Quand
<code>workflow.started</code>	Le passage démarre
<code>workflow.step</code>	Une étape change d'état
<code>workflow.finished</code>	Le passage se termine, quel que soit son état
<code>workflow.queued</code>	Un passage entre dans la file
<code>workflow.changed</code>	Un flux est enregistré ou supprimé
<code>workflow.tasks.changed</code>	Une tâche est déclarée ou retirée
<code>workflow.debug.arret</code>	Le passage s'arrête sur une étape
<code>workflow.debug.repris</code>	Un ordre a été donné
<code>workflow.debug.fin</code>	La session se ferme
<code>workflow.debug.points</code>	Les points d'arrêt ont changé
<code>workflow.alerte</code>	Une alerte est posée ou marquée lue

Le panneau écoute les cinq premiers et repeint au fur et à mesure. Un panneau qui n'afficherait le résultat qu'à la fin laisserait croire que rien ne se passe, et l'utilisateur relancerait.

Exporter un passage

L'export d'un passage n'est pas un ISOF. Un ISOF est signé, porte sa provenance et se vérifie. L'export d'un passage est un JSON ordinaire, bon à poser un résultat dans un tableur ou dans un courriel, et sans valeur de preuve. Le fichier emporte donc son avertissement, en distinguant trois cas : une simulation, un passage relu dont les sorties ont été écartées, et un passage complet mais non signé.

Les CSV exportés depuis le panneau portent une marque d'encodage en tête, sans quoi un tableur lit l'UTF-8 comme du latin-1 et « échantillon » devient illisible, ce que l'utilisateur attribue à l'export plutôt qu'au tableur.

20. Déclarer une tâche

Le registre est ouvert. Un plugin déclare ses tâches, et elles apparaissent dans l'éditeur, dans la liste de choix, dans le calcul des capacités et dans le journal des accès, sans qu'aucun fichier du noyau ne change.

```
K.workflows.declarerTache({
  id: "monplugin.ranger",
  source: "monplugin",          // sert de groupe dans l'editeur

  ecrit: true,                  // absent vaut « écrit » : le default prudent
  outil: false,                 // invisible au modele, workflows seulement
  capacite: "samples:write",    // sinon <source>:read | <source>:write

  title: { fr: "Ranger un lot", en: "File a batch" },
  notice: { fr: "...", en: "..." }, // lue par le modele, si outil

  entrees: [
    { nom: "lot", type: "texte", requis: true, default: null,
      libelle: { fr: "Identifiant du lot", en: "Batch id" } }
  ],
  sorties: [ { nom: "ranges", type: "nombre" } ],

  executer: function (args, ctx) {
    // ctx.passage, ctx.etape, ctx.contexte, ctx.agent, ctx.simulation
    return Promise.resolve({ ranges: 3 });
  }
});
```

La déclaration rend un jeton qui sait se retirer, ce dont un plugin a besoin quand il se décharge : sa tâche disparaît alors du registre, et les flux qui l'employaient tombent proprement sur l'état *absente* plutôt que sur une erreur.

Six habitudes qui tiennent le reste

1. **Rendre une promesse d'objet.** Les clefs de cet objet sont ce que les étapes suivantes liront sous ``${id}.clef``. Une tâche qui ne rendrait qu'un booléen laisserait le workflow aveugle sur ce qui vient de se passer.

2. **Faire remonter le message réel.** Lire le corps de la réponse avant d'essayer de l'analyser : sur une erreur, l'analyse échoue avant d'avoir pu lire la cause, et l'appelant reçoit une plainte sur du JSON invalide au lieu du motif du refus.
3. **Accepter la chaîne autant que l'objet.** Le moteur substitue des chaînes : un booléen arrive en texte aussi souvent qu'en booléen, et un objet écrit dans l'éditeur arrive en texte JSON. Ne tester que la forme attendue fait échouer une étape correcte, ou retomber en silence sur une valeur par défaut.
4. **Ne pas réimplémenter les bornes.** Les routes bornent déjà l'accès aux racines déclarées, la visibilité et les plafonds. Une seconde vérification finirait par diverger de la première, et l'une des deux deviendrait la porte que personne ne surveille.
5. **Passer par le registre plutôt que par la route.** Quand une opération a déjà son registre dans la page, comme l'acquisition, l'appeler directement contournerait les verrous qu'il pose, et un workflow deviendrait le chemin par lequel une source non autorisée sort quand même.
6. **Ne déclarer une tâche que si son module est là.** Déclarée sans sa route, elle apparaît dans la liste et échoue à l'appel : le modèle reçoit une erreur d'introuvable là où il attend une réponse, et il en conclut que la donnée n'existe pas.