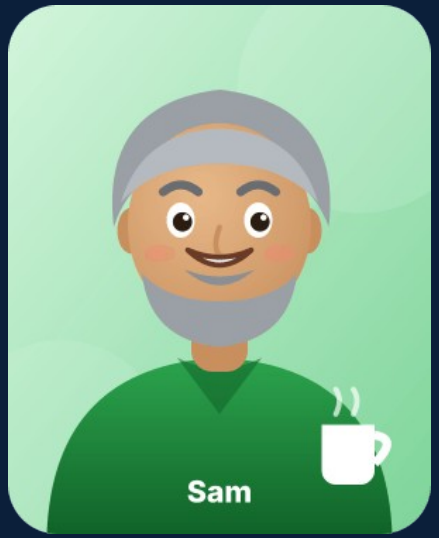
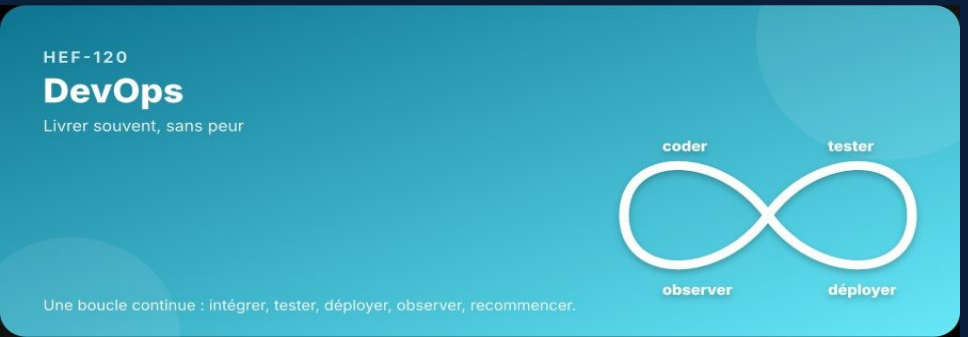
HEF-120 · MISSION 03 / 10

Build reproductible

« Ça marche chez moi » est un symptôme : la machine du développeur et celle de production ne construisent pas la même chose.





Badge : Constructeur déterministe

95 min · intermédiaire

L'histoire

Un jeudi, HashBox reconstruit une ancienne version pour corriger un bug urgent. Le résultat n'a rien à voir avec celui d'il y a six mois : une dépendance a été mise à jour automatiquement, et le programme plante au démarrage. Personne ne sait quelle version exacte de chaque bibliothèque avait été utilisée.

> *« Un build doit être comme une recette précise, dit Sam : mêmes ingrédients, mêmes quantités, même four. Si le résultat change d'une fois sur l'autre sans que la recette change, c'est que la recette est incomplète. »*



Tu apprends à rendre un build déterministe : versions figées, empreintes vérifiées, artefacts identiques bit à bit, caches fiables.

Ta mission

À la fin, je serai capable de…

1. expliquer pourquoi un build doit être reproductible
2. figer les dépendances avec un fichier de verrouillage et des empreintes
3. produire deux fois un artefact identique (déterminisme)
4. construire un cache fondé sur le contenu et un build incrémental
5. produire un inventaire des composants (SBOM) d'un artefact

Défi d'échauffement

Défi d'échauffement

Une pâtisserie vend des gâteaux. La recette dit « un peu de farine, du beurre, cuire jusqu'à ce que ce soit prêt ». Chaque cuisinier fait un gâteau différent. Écris ce qu'il faudrait préciser pour que deux cuisiniers obtiennent le **même** gâteau à chaque fois. Qu'est-ce qui, dans un logiciel, joue le rôle de la « farine » et du « four » ?

Le concept

Qu'est-ce qu'un build reproductible ?

Un build est **reproductible** si, à partir des **mêmes sources** et de la **même configuration**, on obtient le **même artefact**, quel que soit le jour, la machine ou la personne.

Avantage	Pourquoi
Fiabilité	ce qu'on teste est ce qu'on déploie
Diagnostic	« chez moi ça marche » disparaît
Sécurité	on peut vérifier qu'un binaire correspond bien aux sources (chaîne d'approvisionnement)
Retour arrière	on peut reconstruire exactement une ancienne version
Cache	même entrée → même sortie → on peut réutiliser

Le concept

Les sources de non-déterminisme

Source	Exemple	Remède
Dépendances flottantes	`requests>=2`	versions exactes + verrouillage
Horodatages	date de build dans l'archive	date fixe (`SOURCE_DATE_EPOCH`)
Ordre des fichiers	ordre du système de fichiers	trier
Chemins absolus	`/home/kodi/...` incorporé	chemins relatifs, répertoires normalisés
Environnement	variables, locale, fuseau	environnement explicite
Aléatoire	identifiants générés	graine fixée, pas d'aléa dans le build
Réseau	téléchargement au moment du build	dépendances préchargées et vérifiées
Outils	compilateur différent	version de l'outil figée

Le concept

Verrouillage et empreintes

- **Fichier de dépendances** : ce que tu demandes (`requests>=2,<3`).
- **Fichier de verrouillage** (*lock file*) : **ce qui a été résolu**, versions exactes **et empreintes** (hachés) de chaque paquet, y compris les dépendances transitives.

```
requests==2.32.3 --hash=sha256:...
urllib3==2.2.2   --hash=sha256:...
```

À l'installation, l'outil **vérifie** l'empreinte : un paquet altéré (ou remplacé par un homonyme malveillant) est refusé (`pip install --require-hashes`). Le verrouillage se **commit** dans le dépôt et se met à jour par un changement **explicite et relu**.

Le concept

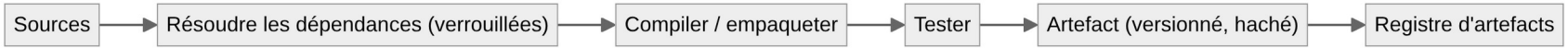
Le contenu comme clé : cache et build incrémental

Un **cache adressé par le contenu** : clé = haché de toutes les entrées de l'étape (sources, versions, options). Si la clé existe, on réutilise le résultat ; sinon on construit et on stocke. Un changement dans une entrée change la clé, donc **invalide** exactement ce qu'il faut.

```
clé(étape) = H( H(sources) + H(dépendances) + H(commande) + H(clés des étapes précédentes) )
```

C'est le principe des systèmes de build modernes (Bazel, Nix, Docker avec ses couches).

Les étapes d'un build



Le concept

Les étapes d'un build (suite)

Description du schéma : cinq étapes en chaîne : les sources, la résolution des dépendances verrouillées, la compilation ou l'empaquetage, les tests, et la publication d'un artefact versionné et haché dans un registre.

Principe clé : construire une fois, déployer partout. L'artefact testé est **exactement** celui qui part en production (jamais reconstruit pour chaque environnement).

SBOM : inventaire des composants

Un **SBOM** (*Software Bill of Materials*) liste chaque composant d'un artefact (nom, version, empreinte, licence). Il sert à repérer rapidement les composants vulnérables (HEF-100, mission 8) quand une faille est annoncée.

Le concept

Versionner les artefacts

Chaque artefact porte une version immuable (par exemple `1.4.2+build.57` ou le SHA du commit) et son **haché**. On ne remplace jamais un artefact publié.

🧠 ***Astuce*** *: fais deux builds consécutifs et compare les empreintes. Si elles diffèrent, cherche la différence avec diff : tu trouveras presque toujours un horodatage ou un ordre de fichiers.*

Exemple pas à pas

Problème : montrer qu'un build qui incorpore la date n'est pas reproductible, le rendre déterministe, vérifier un fichier de verrouillage par empreintes, et produire un SBOM.

```
import hashlib, io, json, tempfile, time, zipfile
from pathlib import Path

def sha256(octets):
    return hashlib.sha256(octets).hexdigest()

def creer_projet(dossier):
    Path(dossier, "app").mkdir()
    Path(dossier, "app", "main.py").write_text("print('HashBox')\n")
    Path(dossier, "app", "outils.py").write_text("def double(x):\n    return 2 * x
   \n")
    Path(dossier, "LISEZMOI.txt").write_text("HashBox 1.0\n")

# ---------------- build NAÏF : horodatage et ordre du système de fichiers -------
   ---------
```

Exemple pas à pas

Exemple pas à pas (suite)

```
def build_naif(dossier):
    tampon = io.BytesIO()
    with zipfile.ZipFile(tampon, "w") as z:
        for chemin in Path(dossier).rglob("*"):                          # ordre no
   n garanti
            if chemin.is_file():
                z.write(chemin, chemin.relative_to(dossier))             # incorpor
   e la date de modification
        z.writestr("BUILD_INFO", f"construit à {time.time()}")          # horodatag
   e dans l'artefact
    return tampon.getvalue()

# ---------------- build REPRODUCTIBLE ----------------
DATE_FIXE = (2025, 1, 1, 0, 0, 0)

def build_reproductible(dossier, version):
    tampon = io.BytesIO()
    with zipfile.ZipFile(tampon, "w", zipfile.ZIP_DEFLATED) as z:
        for chemin in sorted(p for p in Path(dossier).rglob("*") if p.is_file()):
```

Exemple pas à pas

Exemple pas à pas (suite)

```
            # ordre trié
              info = zipfile.ZipInfo(str(chemin.relative_to(dossier)), date_time=DAT
   E_FIXE)      # date fixe
              info.external_attr = 0o644 << 16
                 # permissions fixes
              info.compress_type = zipfile.ZIP_DEFLATED
              z.writestr(info, chemin.read_bytes())
        info = zipfile.ZipInfo("VERSION", date_time=DATE_FIXE); info.external_attr
    = 0o644 << 16
        z.writestr(info, version)
    return tampon.getvalue()

with tempfile.TemporaryDirectory() as d:
    creer_projet(d)
    a = sha256(build_naif(d)); time.sleep(1.1); b = sha256(build_naif(d))
    print("build naïf          :", a[:12], b[:12], "identiques ?", a == b)
    assert a != b
    a = sha256(build_reproductible(d, "1.0.0")); time.sleep(1.1); b = sha256(build
   _reproductible(d, "1.0.0"))
```

Exemple pas à pas (suite)

```
    print("build reproductible :", a[:12], b[:12], "identiques ?", a == b)
    assert a == b

# ---------------- le même contenu construit dans un AUTRE dossier donne le même a
   rtefact ----------------
with tempfile.TemporaryDirectory() as d1, tempfile.TemporaryDirectory() as d2:
    creer_projet(d1); creer_projet(d2)
    assert sha256(build_reproductible(d1, "1.0.0")) == sha256(build_reproductible(
   d2, "1.0.0"))
    print("chemins différents, même artefact : OK")

# ---------------- fichier de verrouillage avec empreintes ----------------
PAQUETS = {"outil-a-1.2.0.whl": b"contenu de outil-a 1.2.0", "outil-b-0.9.1.whl":
   b"contenu de outil-b 0.9.1"}
LOCK = [{"nom": "outil-a", "version": "1.2.0", "fichier": "outil-a-1.2.0.whl", "sh
   a256": sha256(PAQUETS["outil-a-1.2.0.whl"])},
        {"nom": "outil-b", "version": "0.9.1", "fichier": "outil-b-0.9.1.whl", "sh
   a256": sha256(PAQUETS["outil-b-0.9.1.whl"])}]
```

Exemple pas à pas (suite)

```
def installer(lock, telecharger):
    installes = []
    for entree in lock:
        contenu = telecharger(entree["fichier"])
        if sha256(contenu) != entree["sha256"]:
            raise ValueError(f"empreinte invalide pour {entree['nom']} : paquet al
   téré ou remplacé")
        installes.append(f"{entree['nom']}=={entree['version']}")
    return installes

assert installer(LOCK, PAQUETS.__getitem__) == ["outil-a==1.2.0", "outil-b==0.9.1"
   ]
pirate = dict(PAQUETS, **{"outil-b-0.9.1.whl": b"contenu de outil-b avec une porte
    derobee"})
try:
    installer(LOCK, pirate.__getitem__)
    raise AssertionError("l'altération aurait dû être refusée")
except ValueError as e:
    print("refusé :", e)
```

Exemple pas à pas (suite)

```
# ---------------- SBOM minimal ----------------
def sbom(lock, artefact):
    return {"artefact_sha256": sha256(artefact),
            "composants": [{"nom": e["nom"], "version": e["version"], "sha256": e[
   "sha256"]} for e in sorted(lock, key=lambda e: e["nom"])]}

with tempfile.TemporaryDirectory() as d:
    creer_projet(d)
    document = sbom(LOCK, build_reproductible(d, "1.0.0"))
print(json.dumps(document, indent=1)[:200], "...")
assert len(document["composants"]) == 2 and len(document["artefact_sha256"]) == 64
```

Lecture :

1. Le **build naïf** change à chaque exécution (horodatage, date de modification des fichiers) : impossible de comparer ou de vérifier.

Exemple pas à pas (suite)

2. Le **build reproductible** fixe tout ce qui varie : ordre trié, dates et permissions figées, aucune donnée de l'heure. Deux builds, même dans des **dossiers différents**, donnent le même artefact bit à bit.
3. Le **verrouillage avec empreintes** refuse un paquet altéré — la défense contre un remplacement malveillant dans la chaîne d'approvisionnement (voir HEF-100).
4. Le **SBOM** relie l'artefact (son haché) à la liste exacte de ses composants.

À toi, avec aide

Écris `verifier_lock(requirements, lock)` qui contrôle la cohérence entre un fichier de dépendances et son verrouillage :

- chaque dépendance déclarée (`nom` + contrainte `==`, `>=` ou `<`) est présente dans le lock ;
- la version verrouillée respecte la contrainte ;
- chaque entrée du lock a une empreinte (non vide) ;
- aucun paquet n'est verrouillé sans être déclaré **ou** dépendance transitive connue (liste fournie).

Renvoie une liste de problèmes. Teste avec un cas bon et trois mauvais.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Représente les versions par des tuples d'entiers pour comparer.

Indice 2

Une contrainte `>=1.2` est respectée si `(1,2,…) >= (1,2)`.

Indice 3

Les paquets transitifs sont fournis dans un ensemble connus.

À toi, avec aide · réponse

Réponse

```
import re

def version(v):
    return tuple(int(x) for x in v.split("."))

def respecte(v, contrainte):
    m = re.fullmatch(r"(==|>=|<)(.+)", contrainte)
    op, ref = m.group(1), version(m.group(2))
    return {"==": version(v) == ref, ">=": version(v) >= ref, "<": version(v) < re
   f}[op]

def verifier_lock(requirements, lock, connus=frozenset()):
    problemes = []
    par_nom = {e["nom"]: e for e in lock}
    for nom, contrainte in requirements.items():
        if nom not in par_nom:
            problemes.append(f"{nom} déclaré mais absent du lock"); continue
        if not respecte(par_nom[nom]["version"], contrainte):
            problemes.append(f"{nom} {par_nom[nom]['version']} ne respecte pas {co
```

À toi, avec aide · réponse

Réponse (suite)

```
    ntrainte}")
     for e in lock:
         if not e.get("sha256"):
             problemes.append(f"{e['nom']} sans empreinte")
         if e["nom"] not in requirements and e["nom"] not in connus:
             problemes.append(f"{e['nom']} verrouillé mais ni déclaré ni transitif
    connu")
     return problemes

req = {"flask": ">=2.0", "outil": "==1.2.0"}
bon = [{"nom": "flask", "version": "2.3.1", "sha256": "ab"}, {"nom": "outil", "ver
    sion": "1.2.0", "sha256": "cd"}, {"nom": "click", "version": "8.1.0", "sha256":
     "ef"}]
assert verifier_lock(req, bon, connus={"click"}) == []
assert verifier_lock(req, bon[:1]) == ["outil déclaré mais absent du lock"]
assert verifier_lock(req, [dict(bon[0], version="1.9.0"), bon[1]]) == ["flask 1.9.
    0 ne respecte pas >=2.0"]
assert verifier_lock(req, [bon[0], dict(bon[1], sha256="")]) == ["outil sans empre
    inte"]
```

À toi, avec aide · réponse

Réponse (suite)

```
assert verifier_lock(req, bon) == ["click verrouillé mais ni déclaré ni transitif
   connu"]
print("verrouillage vérifié")
```

Ce contrôle en CI garantit que le verrouillage suit les déclarations : personne ne contourne l'empreinte « pour que ça passe ».

À toi, en solo

Un petit système de build incrémental. Écris `Construction` où chaque cible a : des **sources** (contenu en mémoire), des **dépendances** (autres cibles) et une **commande** (fonction pure qui produit un contenu). Le système :

1. calcule la **clé** de chaque cible = haché(contenu des sources + clés des dépendances + nom de la commande) ;
2. n'exécute une cible que si sa clé n'est pas déjà en cache ;
3. compte les exécutions réelles ;
4. teste un graphe `lib → app → paquet` et `lib → tests` : une première construction (tout est exécuté), une seconde identique (rien n'est exécuté), la modification d'un fichier de `app` (seuls `app` et `paquet` sont reconstruits, pas `lib` ni `tests`), la modification de `lib` (tout est reconstruit).

À toi, en solo · réponse

Pistes de correction

```
import hashlib

class Construction:
    def __init__(self):
        self.cibles, self.cache, self.executions = {}, {}, []
    def cible(self, nom, sources, deps, commande):
        self.cibles[nom] = {"sources": sources, "deps": deps, "commande": commande
   }
    def cle(self, nom, memo=None):
        memo = {} if memo is None else memo
        if nom in memo:
            return memo[nom]
        c = self.cibles[nom]
        h = hashlib.sha256()
        for chemin in sorted(c["sources"]):
            h.update(chemin.encode()); h.update(c["sources"][chemin].encode())
        for d in sorted(c["deps"]):
            h.update(self.cle(d, memo).encode())
        h.update(c["commande"].__name__.encode())
```

À toi, en solo · réponse

Pistes de correction (suite)

```
        memo[nom] = h.hexdigest()
        return memo[nom]
    def construire(self, nom, memo=None):
        memo = {} if memo is None else memo
        for d in self.cibles[nom]["deps"]:
            self.construire(d, memo)
        cle = self.cle(nom, memo)
        if cle not in self.cache:
            c = self.cibles[nom]
            entrees = {d: self.cache[self.cle(d, memo)] for d in c["deps"]}
            self.cache[cle] = c["commande"](c["sources"], entrees)
            self.executions.append(nom)
        return self.cache[cle]

def compiler(sources, entrees):  return "compilé(" + "+".join(sorted(sources)) + "
   )"
def assembler(sources, entrees): return "app(" + ",".join(sorted(entrees.values())
   ) + ")"
def empaqueter(sources, entrees): return "zip(" + ",".join(sorted(entrees.values()
```

À toi, en solo · réponse

Pistes de correction (suite)

```
    )) + ")"
def tester(sources, entrees):     return "ok(" + ",".join(sorted(entrees.values())
    ) + ")"

def monter(app_src, lib_src):
    b = Construction()
    b.cible("lib", {"lib.py": lib_src}, [], compiler)
    b.cible("app", {"app.py": app_src}, ["lib"], assembler)
    b.cible("paquet", {}, ["app"], empaqueter)
    b.cible("tests", {"t.py": "x"}, ["lib"], tester)
    return b

b = monter("v1", "L1")
for t in ("paquet", "tests"): b.construire(t)
assert sorted(b.executions) == ["app", "lib", "paquet", "tests"]
b.executions.clear()
for t in ("paquet", "tests"): b.construire(t)
assert b.executions == []                                    # rien n'a chan
    gé : tout vient du cache
```

À toi, en solo · réponse

Pistes de correction (suite)

```
b.cibles["app"]["sources"]["app.py"] = "v2"
for t in ("paquet", "tests"): b.construire(t)
assert sorted(b.executions) == ["app", "paquet"]                    # seulement ce
   qui dépend de app
b.executions.clear()
b.cibles["lib"]["sources"]["lib.py"] = "L2"
for t in ("paquet", "tests"): b.construire(t)
assert sorted(b.executions) == ["app", "lib", "paquet", "tests"]
print("construction incrémentale correcte")
```

Le cache adressé par le contenu reconstruit **exactement** ce qui dépend d'un changement — jamais plus, jamais moins. C'est le cœur de Bazel, Nix et des couches Docker.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Dépendances flottantes (`>=`) en production	Le build d'hier ≠ celui d'aujourd'hui	Verrouillage versions + empreintes, mise à jour explicite
Horodatages et ordre aléatoire dans l'artefact	Impossible à comparer ou vérifier	Dates fixes, tri, `SOURCE_DATE_EPOCH`
Reconstruire par environnement	Ce qu'on teste n'est pas ce qu'on déploie	Construire une fois, promouvoir le même artefact
Cache indexé par le nom (pas le contenu)	Résultats périmés	Clé = haché de toutes les entrées

Mini-boss

Mini-boss

Chaîne de build vérifiable. Construis `chaine_build.py` pour un petit projet Python :

1. un verrouillage avec empreintes, vérifié avant toute installation ;
2. un build reproductible (archive à date fixe, ordre trié, permissions fixes) ;
3. un cache par contenu avec compteur d'exécutions réelles ;
4. un SBOM JSON (artefact + composants + empreintes) ;
5. une **vérification indépendante** : un second script reconstruit l'artefact depuis les sources dans un autre dossier et compare l'empreinte avec celle publiée ;
6. tests : modification d'une source, modification d'une dépendance, paquet altéré, horodatage glissé volontairement dans le build (le test de reproductibilité doit échouer).

Mini-boss · réponse

Pistes de correction

Un bon travail : fonctions `verifier_lock`, `build_reproductible`, `Construction`, `sbom` assemblées ; le second script (`verifier_build.py`) recalcule l'empreinte et compare avec le SBOM ; le test de « mauvais build » ajoute `time.time()` et vérifie que deux builds consécutifs diffèrent, ce qui déclenche l'échec de la porte de reproductibilité. Point de réflexion attendu : un build reproductible permet à un tiers de **vérifier** que l'artefact distribué correspond aux sources publiées.

Quiz éclair

1. Un build reproductible produit :

A. un artefact différent à chaque fois
B. le même artefact à partir des mêmes sources et configuration
C. uniquement des logs

2. Un fichier de verrouillage avec empreintes sert à :

A. augmenter la vitesse du build
B. remplacer les tests
C. figer les versions exactes et refuser un paquet altéré

3. Une clé de cache adressée par le contenu change quand :

A. une des entrées de l'étape change

Quiz éclair

Quiz éclair (suite)

B. l'heure change
C. le nom de la machine change

4. 🔁 *Révision mission 02* : Une branche Git est :

A. une copie complète du projet
B. un pointeur mobile vers un commit
C. un fichier de configuration

5. 🔁 *Révision mission 02* : Pour annuler un changement déjà partagé, on utilise :

A. git rebase sur la branche publique
B. supprimer le dépôt
C. git revert, qui crée un commit d'annulation

Quiz éclair · réponse

Réponses

1. **B**.
2. **C**.
3. **A**.
4. **B**.
5. **C**.

Je réfléchis

1. Peux-tu reconstruire aujourd'hui, à l'identique, la version de ton projet d'il y a six mois ? Qu'est-ce qui manquerait ?
2. Quelle entrée non verrouillée (version, outil, variable) est la plus risquée dans ton build ?

À retenir

1. Reproductible : mêmes entrées, même artefact.
2. Sources de variation : dépendances flottantes, horodatages, ordre, chemins, environnement.
3. Verrouillage avec empreintes ; paquets altérés refusés.
4. Cache adressé par le contenu ; construire une fois, déployer partout.
5. SBOM : inventaire des composants relié à l'empreinte de l'artefact.

MISSION ACCOMPLIE

+100 XP

Badge débloqué : Constructeur déterministe

Prochaine mission : Conteneurs

