Hamac

SPÉCIFICATION

Spécification des manifestes

Le vocabulaire LOAF de Hamac (module `hamac` 0.2.0) et les vérifications effectuées au chargement des manifestes.

Version 0.2 · 2026-10-30 · BROUILLON

Hyper Open X — jalon J2

OCamlPro SAS — `forge.ocamlpro.com/OSS/hamac` — AGPL-3.0

Sommaire

1 Objet et principes . . . 3
1.1 Sortes de manifestes . . . 3
1.2 Principes . . . 3
2 Exemple . . . 4
2.1 Le patron de l’organisation . . . 4
2.2 Le provider de base de données . . . 4
2.3 Les services . . . 5
2.4 La stack . . . 6
2.5 Ce que Hamac en déduit . . . 6
3 Manifestes LOAF . . . 7
3.1 Fichier et vocabulaire . . . 7
3.2 Ensemble de fichiers . . . 7
3.3 Modules et bibliothèques . . . 7
3.4 Décrire, traduire, agir . . . 8
3.5 Outils . . . 8
3.6 Stabilité . . . 8
4 Conventions du vocabulaire . . . 9
4.1 Lecture des tableaux de champs . . . 9
4.2 Noms . . . 9
4.3 Quantités et durées . . . 9
4.4 Métadonnées . . . 10
4.5 Termes rendus par Hamac . . . 10
5 Service . . . 11
5.1 Exécution . . . 11
5.2 Ressources et mise à l’échelle . . . 11
5.3 Sondes, métriques et journaux . . . 12
5.4 Exemple . . . 12
6 Capacités, providers et consommations . . . 13
6.1 Provider . . . 13
6.2 Contrats des capacités . . . 13
6.3 Consommations . . . 14
7 Données : stockage, sauvegarde, secrets . . . 16
7.1 Volumes et sauvegarde . . . 16
7.2 Secrets . . . 16
7.3 Exemple . . . 17
8 Patrons . . . 18
8.1 Un patron est un module . . . 18
8.2 Zones et modèles de sécurité . . . 18
8.3 Enclaves . . . 20
8.4 Axes d’assurance . . . 20
8.5 Exigences de placement . . . 21
9 Sécurité, placement et exposition d’un service . . . 22
9.1 Zone et flux . . . 22
9.2 Routes . . . 22
9.3 Placement . . . 23
9.4 Exposition . . . 23
10 Bundle . . . 25
11 Profil de provisioning . . . 26
12 Infrastructure . . . 28
13 Stack . . . 30
14 Validation . . . 31
14.1 Trois étages . . . 31

14.2 Refus du langage 31
14.3 Catalogue 31

SECTION 1

Objet et principes

Ce document définit les manifestes de Hamac : ce qu'ils décrivent, les champs de chaque sorte de manifeste, les valeurs par défaut que Hamac applique et les diagnostics qu'il émet. Un manifeste conforme peut être écrit à la main ou produit par un script, un langage de plus haut niveau ou tout autre générateur.

Les manifestes sont écrits en **LOAF**, un langage de manifestes défini par sa propre spécification (document « LOAF », `spec/loaf.typ` du dépôt `loaf`) : syntaxe, blocs, champs et marques de cardinalité, tables et listes, gabarits `fmt`, modules et versions, codes de refus `L001` à `L014`. Le présent document ne reprend pas cette définition. Il décrit le **vocabulaire** de Hamac, c'est-à-dire les blocs, les champs et les règles que Hamac fournit au langage, puis les contrôles que Hamac ajoute.

L'implémentation de référence est la bibliothèque `hamac-manifest` (`pkgs/hamac-manifest/`), qui lit et valide des manifestes sans dépendre du reste de Hamac. Le vocabulaire est le fichier `pkgs/hamac-manifest/vocabulary/hamac.loaf` (module `hamac`, version 0.2.0) ; pour la forme d'un champ, ce fichier fait foi.

1.1 Sortes de manifestes

Hamac connaît six sortes de manifestes. Chacune est un bloc du vocabulaire, et chacune a son auteur :

Bloc	Auteur	Contenu
`service`	l'équipe de développement	une unité déployable, ses ressources, ses dépendances et sa sensibilité (§5) ; un service qui fournit une capacité générique (base de données, cache) à d'autres services est un **provider** (§6)
`pattern`	la personne responsable de la sécurité	des zones, le modèle de sécurité qui les ordonne, des exigences d'assurance et des contraintes sur la topologie (§8)
`stack`	l'équipe d'exploitation	les services à déployer ensemble et les choix qui les concernent (§13)
`infrastructure`	le planificateur de Hamac, relu par l'exploitation	une topologie : nœuds, segments réseau, filtrage (§12)
`provisioning_profile`	l'administration du parc	le système installé sur une machine (§11)
`bundle`	l'administration du parc	une brique réutilisable d'un profil (§10)

1.2 Principes

1. **Infrastructure inférée.** Un service déclare ses besoins : ressources, dépendances, sensibilité, stockage. Hamac calcule une topologie qui les satisfait. Un manifeste de service ne nomme aucune machine.
2. **Secrets par référence.** Aucun champ ne contient de secret en clair : un secret est une référence vers un coffre, ou une valeur que Hamac engendre.
3. **Validation stricte.** Un champ inconnu, une référence à un manifeste absent ou un flux que le modèle de sécurité interdit entraînent le refus des manifestes. Chaque diagnostic donne un code et l'emplacement de l'instruction fautive (§14).
4. **Vocabulaire versionné.** Le vocabulaire est un module LOAF ; sa version suit les règles de compatibilité du langage (§3.6).

Les mots **DOIT**, **NE DOIT PAS**, **DEVRAIT** et **PEUT** ont le sens des RFC 2119 et 8174.

SECTION 2

Exemple

L'exemple décrit une petite application de ressources humaines : un portail public, une API qui expose les données du personnel en s'appuyant sur une base PostgreSQL, et un traitement qui consolide la paie à partir de l'API. Il tient en quatre parties : le patron de l'organisation, le provider de base de données, les services et la stack qui les réunit. Les fichiers complets sont dans `examples/hr-stack/` et `providers/` ; les extraits omettent les lignes `loaf 0.1.` et `use hamac 0.2.` qui ouvrent chaque fichier.

2.1 Le patron de l'organisation

```
module exemple_rh 1.0.0.
use hamac_blp 0.2.       % le modèle de sécurité Bell-LaPadula
use hamac_rbac 0.2.      % les rôles

pattern exemple_rh {
  % Les niveaux de sensibilité, du moins au plus sensible.
  blp::levels [public, internal, confidential].
  % Les catégories de besoin d'en connaître.
  blp::category rh; finance.
  % Les rôles que l'organisation attribue à ses utilisateurs.
  rbac::role rh; comptable; admin.

  % Les zones : chacune porte un niveau et des catégories.
  zone public { blp::level public. }
  zone internal { blp::level internal. }
  zone rh {
    blp::level confidential.
    blp::category rh; finance.
  }
}
```

Le patron est écrit par la personne responsable de la sécurité. Il définit trois zones, et choisit deux modèles de sécurité livrés avec Hamac : `hamac_blp`, qui étiquette chaque zone d'un niveau et de catégories et en déduit quelle zone peut lire quelle autre, et `hamac_rbac`, qui gère des rôles. Les champs d'un modèle portent son préfixe : `blp::level`, `rbac::role`. La zone `rh` accueille des données confidentielles des catégories `rh` et `finance`. Le patron est un module : la stack le charge par `use exemple_rh 1.0.` Le fichier complet ajoute des exigences de placement, avec le module `hamac_ha` (§8.5).

2.2 Le provider de base de données

```
service postgresql {
  runtime http.
  capability database.            % il fournit une base de données
  input user, gen(alpha, 12).      % engendré pour chaque consommateur
  input password, gen(alnum, 32).
  input dbname, cat(consumer(name), "_db").
  environment "POSTGRES_USER", input(user).
  environment "POSTGRES_PASSWORD", input(password).
  environment "POSTGRES_DB", input(dbname).
  port 5432 { protocol tcp. }
}

% Ce que le provider offre à ses consommateurs : le contrat database.
database postgresql {
  host instance(name).
  port 5432.
  user input(user).
  password input(password).
  dbname input(dbname).
}
```

Un provider est un service comme un autre, qui déclare une capacité (`database`). Hamac en crée une instance par service qui le consomme : chaque consommateur reçoit sa propre base et ses propres identifiants, tirés au hasard au déploiement. Le bloc `database postgresql` remplit le contrat de la capacité : les attributs que tout provider de base de données offre à ses consommateurs (§6.2).

2.3 Les services

```
service hr_api {
  runtime http.
  replicas 2.                      % deux instances
  zone rh.                         % il traite des données de la zone rh
  spread failure_domain.           % sur deux domaines de panne distincts

  % Une base de données, sans nommer le provider : Hamac le choisit, et
  % remplit le gabarit avec les attributs de son contrat.
  consumes db {
    capability database.
    inject "DATABASE_URL",
           fmt("postgresql://{~.database.user}:{~.database.password}@{~.database.host}:
{~.database.port}/{~.database.dbname}").
  }

  route "/health" { public. }
  route "/employees/salaries" {
    blp::classification confidential.
    blp::category rh; finance.
    rbac::role comptable; admin.
  }

  port 8080 {}
}

service payroll_report {
  runtime batch.
  zone rh.

  % L'API RH, nommée : un flux réseau de payroll_report vers hr_api.
  consumes api {
    service hr_api.
    inject "HR_API_URL", fmt("http://{~.host}:{~.port}").
  }

  % Un cache sur le disque du nœud, qu'il suffit de recalculer.
  storage cache {
    mount "/var/cache/report".
    class local.
    size 5, gi.
    rebuild none.
  }
}

service portal {
  runtime http.
  replicas 2.
  zone public.
  port 8080 {}
}
```

Le bloc `service hr_api` décrit le service sans répéter son nom : chaque instruction s'applique à `hr_api`. Le bloc `consumes db` déclare un besoin, une base de données ; dans ce bloc, `~` désigne la consommation, et `~.database.user` l'attribut `user` du contrat du provider que Hamac lui associera. Une route fixe les conditions d'accès à un chemin HTTP, dans les termes des modèles du patron : seul un utilisateur habilité au niveau `confidential`, détenteur des catégories `rh` et `finance` (`hamac_blp`) et du rôle `comptable` ou `admin` (`hamac_rbac`), lit les salaires. Le fichier d'un service qui emploie ces champs charge leurs modèles (`use hamac_blp 0.2.`).

2.4 La stack

```
use exemple_rh 1.0.

stack hr {
  description "Application RH : portail public, API RH, consolidation".
  accept_experimental.

  service portal {}
  service hr_api {}
  service payroll_report {}
}
```

La stack est écrite par l'équipe d'exploitation. Elle nomme les services à déployer et charge le patron. Un service de l'ensemble qu'elle ne nomme pas, ici `postgresql`, est un provider : il n'est déployé que pour les services qui le consomment.

2.5 Ce que Hamac en déduit

Les commandes `hamac validate`, `hamac resolve` et `hamac plan`, appliquées aux dossiers `examples/hr-stack` et `providers`, établissent :

- La consommation `db` de `hr_api` est résolue vers `postgresql`, seul provider de `database` de l'ensemble. Hamac crée l'instance `postgresql-hr-api-db` ; au déploiement, `DATABASE_URL` vaut `postgresql://<12 lettres>:<32 caractères>@postgresql-hr-api-db:5432/hr_api_db`.
- `payroll_report` reçoit `HR_API_URL=http://hr-api:8080`, que Hamac construit à partir du nom d'hôte de `hr_api` et de son premier port.
- `payroll_report` et `hr_api` sont dans la même zone : le flux entre eux ne demande aucune règle de filtrage. Entre les zones `public` et `rh`, tout est refusé : aucun flux n'est déclaré, et `hamac_blp` ne permet pas à `public` de lire `rh`.
- Le planificateur propose trois nœuds : un pour `public`, deux pour `rh`, car les deux répliques de `hr_api` doivent être sur des domaines de panne distincts.
- Si `portal` déclarait `consumes api { service hr_api. }`, l'ensemble serait refusé (**`E040`**) : `portal` (zone `public`) ouvrirait une connexion vers la zone `rh`, plus sensible, ce que `hamac_blp` ne permet pas.
- Une route qui exigerait le rôle `admn`, mal orthographié, serait refusée (**`E045`**) : aucun patron ne déclare ce rôle.
- `zone dmz.` dans un service serait refusée (**`E042`**) : le patron ne définit pas cette zone.

SECTION 3

Manifestes LOAF

3.1 Fichier et vocabulaire

Un fichier de manifestes commence par la version du langage qu'il suit, puis charge le vocabulaire de Hamac par `use` :

```
loaf 0.1.
use hamac 0.2.

service portal {
  runtime http.
  replicas 2.
  port 8080 {}
}
```

Un manifeste est un bloc du vocabulaire. Le sujet du bloc est le nom du manifeste : ici, le service `portal`. Un fichier **PEUT** contenir plusieurs manifestes, de sortes différentes ou non. Un bloc ou un champ que le vocabulaire ne déclare pas est refusé par le langage (**L001**), comme la faute de frappe `replica 3.`

3.2 Ensemble de fichiers

Hamac charge toujours des manifestes ensemble : les fichiers donnés à une commande forment un **ensemble**, dont le modèle est calculé en une fois. Un répertoire donné en argument vaut les fichiers `.loaf` qu'il contient, à toute profondeur :

```
hamac validate examples/hr-stack providers
```

- Une référence d'un manifeste à un autre (une stack à ses services, un profil à ses bundles, une consommation au service qu'elle nomme) est vérifiée par le langage : le manifeste nommé **DOIT** faire partie de l'ensemble (**L002** sinon).
- Un même bloc ouvert dans plusieurs fichiers réunit leurs contenus. Un fichier peut ainsi compléter un manifeste d'un autre fichier ; comme deux définitions indépendantes d'un même nom se réunissent de la même façon, Hamac le signale par l'avertissement **W003**. Deux valeurs pour un champ qui n'en admet qu'une sont refusées (**L006**).

```
% web.loaf
loaf 0.1.
use hamac 0.2.

service web {
  runtime http.
  artifact "./bin/web" {
    sha256 "0000000000000000000000000000000000000000000000000000000000000000".
    format native.
    arch amd64.
  }
}
```

```
% web-prod.loaf : W003 « service web is defined in web-prod.loaf and web.loaf »
loaf 0.1.
use hamac 0.2.

service web { replicas 3. }
```

3.3 Modules et bibliothèques

Le vocabulaire et les patrons (§8) sont des modules LOAF : des fichiers qui déclarent leur nom et leur version (`module exemple_rh 1.0.0.`), chargés par `use exemple_rh 1.0.` depuis l'ensemble ou depuis un répertoire de bibliothèque. Hamac cherche les modules, dans cet ordre :

1. dans le répertoire du vocabulaire : celui qu'installe le paquet `hamac-manifest`, ou celui que nomme la variable `HAMAC_VOCABULARY` ;
2. dans les répertoires donnés par l'option `-L DIR` (répétable) ;
3. dans `./patterns`, puis dans `~/.hamac/patterns`, s'ils existent.

La sélection des versions, la compatibilité entre une contrainte et une version et le fichier d'empreintes `loaf.sum` suivent la spécification de LOAF ; un module introuvable ou des contraintes incompatibles sont refusés (**L009**).

3.4 Décrire, traduire, agir

Trois rôles se partagent les fichiers d'un ensemble :

Rôle	Ce qu'il écrit
Décrire	Les manifestes (services, patrons, stacks…) posent des faits : une zone, un niveau, « isoler les zones », « ce service est critique ». Leurs auteurs choisissent parmi les champs du vocabulaire et des modules qu'ils chargent.
Traduire	Les modules (`hamac_blp`, `hamac_ha`, ou ceux d'une organisation) contiennent les règles : ils ramènent leurs notions aux *primitives* que Hamac sait appliquer, comme l'ordre des zones (`zone_leq`, §8.2), les enclaves (`enclave`, §8.3) ou les primitives de placement (§8.5).
Agir	Hamac n'interprète que les primitives : il en déduit le filtrage entre zones, le placement des répliques et la configuration de la bordure.

Une notion nouvelle qui se ramène aux primitives existantes s'ajoute par un module, sans changer Hamac ; une primitive nouvelle demande une version de Hamac. Le langage permet d'écrire une règle dans n'importe quel fichier ; sa place est néanmoins dans un module, où elle est versionnée, relue et partagée.

3.5 Outils

L'outil `loaf`, qui accompagne la spécification de LOAF, s'applique aux manifestes de Hamac, le répertoire du vocabulaire étant donné comme bibliothèque (`-L`) :

- `loaf fmt` réécrit des fichiers dans la forme canonique du langage ;
- `loaf export` écrit le modèle d'un ensemble dans le format d'échange de LOAF, une ligne JSON par fait, avec l'emplacement des instructions qui le dénotent :

```
{"args":["hr_api","rh"],"at":[{"column":3,"file":"hr-api.loaf","line":13}],"pred":"service.zone"}
```

3.6 Stabilité

La version du module `hamac` suit les règles de compatibilité de LOAF : un ajout de champ optionnel ou de valeur d'énumération est une version mineure, un retrait ou un changement de forme une version majeure. Un manifeste qui charge `use hamac 0.2.` reste valide avec toute version 0.2.x du vocabulaire.

Certains champs sont **expérimental** : leur sens peut encore changer dans une version mineure. Leur usage produit l'avertissement **W001**, une fois par manifeste et par groupe de champs, sauf si l'ensemble contient une stack qui écrit `accept_experimental` (§13). Les autres champs sont **stable**.

Bloc	Champs expérimentaux
`service`	`storage`, `backup`, `secret`, le placement (`need`, `spread`, `affinity`, `anti_affinity`), `exposure`, et le champ `mode` d'une consommation
`pattern`	`assurance`
`infrastructure`	`certificate`
`provisioning_profile`	`assurance`
`bundle`	`lifecycle`

SECTION 4

Conventions du vocabulaire

4.1 Lecture des tableaux de champs

Chaque section décrit un bloc par un tableau dont la première colonne reprend la déclaration du champ dans le vocabulaire : son nom, sa marque de cardinalité, la sorte de ses valeurs. Les marques ont le sens que leur donne LOAF :

Marque	Nombre de valeurs par bloc
aucune	exactement une : le champ est obligatoire (**L007** s'il manque, **L006** s'il en a deux)
`?`	au plus une
`*`	un nombre quelconque

Un champ sans sorte (`public.`, `required.`) est un drapeau, présent ou absent. `f* k => v` déclare une table, qui associe une valeur à chaque clé ; `f* [s]` déclare une liste, écrite entre crochets. Les sortes `string`, `int`, `name` et `text` sont celles de LOAF ; une sorte qui porte le nom d'un bloc (`service`, `bundle`) désigne les manifestes de ce bloc présents dans l'ensemble ; une sorte qui porte le nom d'une énumération en désigne les valeurs. Un sous-bloc (`port* int { … }`) a pour sujet une valeur de la sorte indiquée.

Hamac applique la valeur indiquée comme « défaut » quand un champ optionnel est absent.

4.2 Noms

- Le nom d'un manifeste est une constante LOAF (`hr_api`). Hamac l'emploie tel quel partout : dans les références entre manifestes, ses diagnostics, ses sorties et les valeurs qu'il engendre (la base attribuée à `hr_api` s'appelle `hr_api_db`).
- Seul un **nom d'hôte** change de forme, parce qu'il n'admet pas `_` : celui d'un service, d'une instance de provider ou d'un nœud remplace `_` par `-` (`hr-api`, `postgresql-hr-api-db`). Un consommateur reçoit le nom d'hôte du service qu'il consomme par son gabarit (`{~.host}`, §6.3). Le nom d'un service **DOIT** donc ne contenir que des minuscules, des chiffres et `_`, ni commencer ni finir par `_`, et ne pas dépasser 63 caractères (**E004**).
- Les valeurs des énumérations s'écrivent avec `_` là où le nom usuel porte `-` : `oci_image`, `measured_tpm2`, `docker_compose`.
- Un champ à plusieurs valeurs porte un nom au singulier ; chaque instruction en donne une : `port 8080 { … }`, `secret session_key { … }`, `need public_ipv4; ssd.`
- Un objet unique est un sous-bloc quand il a un identifiant naturel, qui en est le sujet : `artifact`, `health_check`, `metrics`, `backup`, `os`. Sinon ses champs sont ceux du bloc qui le contient (ressources, placement, mise à l'échelle), préfixés quand leur nom serait ambigu (`memory_limit`, `readiness_timeout`).

4.3 Quantités et durées

Une quantité s'écrit comme un couple (entier, unité) : `memory_limit 512, mi.` pour 512 Mio. Les unités sont des énumérations :

Sorte	Valeurs	Sens
`size_unit`	`ki` · `mi` · `gi` · `ti`	Kio, Mio, Gio, Tio : unités binaires (ci-dessous)
`cpu_unit`	`core` · `m`	cœurs ; millicœurs, un cœur valant 1000 m (`500, m` : un demi-cœur)
`time_unit`	`s` · `min` · `h` · `d`	secondes, minutes, heures, jours

Unités binaires. Les tailles s'expriment en multiples de 1024, selon les préfixes binaires de la norme CEI 60027-2 :

Unité	Nom	Octets
`ki`	kibioctet (Kio)	1024
`mi`	mébioctet (Mio)	1024 × 1024 = 1 048 576

Unité	Nom	Octets
`gi`	gibioctet (Gio)	1024^3 = 1 073 741 824
`ti`	tébioctet (Tio)	1024^4 = 1 099 511 627 776

Ces unités ne sont pas le kilooctet, le mégaoctet ou le gigaoctet du système international (ko, Mo, Go), qui valent 1000, 1000^2 et 1000^3 octets : 1 Gio vaut environ 1,074 Go, et l'écart croît avec l'unité. Une taille connue en unités décimales se convertit en arrondissant au-dessus : un volume de 500 Go (500 × 1000^3 octets, soit environ 465,7 Gio) s'écrit `size 466, gi.` Les délais des sondes (`health_check`, `readiness_interval`, `readiness_timeout`) sont des entiers, en secondes.

4.4 Métadonnées

Tous les blocs de manifeste acceptent les mêmes métadonnées :

Déclaration	Sens
`description? string`	texte libre
`label* string => string`	clés et valeurs courtes, pour les sélecteurs : `label "team", "rh".`
`annotation* string => string`	données libres, que Hamac ignore ; un générateur tiers peut y placer les siennes

4.5 Termes rendus par Hamac

Certaines valeurs ne sont connues qu'au déploiement : un mot de passe tiré au hasard, le nom d'une instance, la valeur d'un secret. Le vocabulaire les déclare comme des termes que Hamac rend (déclarations `render` de LOAF). Ils restent opaques pour les règles du langage, et peuvent paraître partout où un champ attend une valeur de sorte `text`, seuls ou dans un gabarit `fmt`.

Hamac crée une **instance** d'un provider pour chaque consommation qui le retient : chaque consommateur reçoit sa propre base, ses propres identifiants. Les termes suivants désignent des valeurs de cette instance.

Terme	Valeur rendue	Usage
`gen(alnum, 32)`, `gen(alpha, 12)`	32 caractères alphanumériques, 12 lettres minuscules, tirés au hasard	entrées d'un provider
`input(user)`	la valeur de l'entrée `user` de l'instance	champs d'un provider
`consumer(name)`	le nom du service consommateur	entrées d'un provider
`instance(name)`	le nom de l'instance, qui est aussi son nom d'hôte	champs d'un provider
`host(t)`	le nom d'hôte du service `t`	injection d'une consommation de `t`, contrat de la capacité de `t`
`secret(x)`	la valeur du secret `x` déclaré par le service	variables d'environnement et injections

- Le tirage aléatoire utilise le générateur cryptographique du système ; l'option `--seed` des commandes qui résolvent les manifestes le rend reproductible, pour les tests seulement.
- `host(t)` employé ailleurs que dans une consommation de `t` ou que dans le contrat de la capacité de `t`, et `instance(x)` pour un autre attribut que `name`, n'ont pas de valeur : **`E020`**.
- La valeur d'un `secret(x)` n'est écrite dans aucun fichier produit par Hamac : l'agent l'injecte au démarrage du service.

SECTION 5

Service

stable

Un service est une unité déployable : un artefact, son exécution, ses ressources. Cette section décrit les champs d'exécution ; les sections suivantes décrivent les dépendances (§6), les données (§7) et la sécurité (§9).

5.1 Exécution

Déclaration	Sens, défaut et contrôles
`runtime runtime`	nature de la charge : `http` · `cli` · `batch` · `eventloop` · `custom`. `http` active les sondes HTTP et l'exposition
`artifact? string { … }`	ce qui s'exécute ; le sujet est un chemin, une URL ou une référence d'image OCI (ci-dessous)
`replicas? int`	nombre d'instances ; défaut 1, au moins 1 (**E016**)
`port* int { protocol? port_protocol. }`	port écouté par le service ; protocole `tcp` (défaut) ou `udp`
`environment* string => text`	variables d'environnement ; une valeur peut être un terme rendu ou un gabarit (§4.5)

Le sous-bloc `artifact` porte :

Déclaration	Sens et contrôles
`sha256 string`	empreinte de l'artefact : 64 caractères hexadécimaux (**E011**)
`format artifact_format`	`oci_image` · `native` · `incus_vm_image` · `incus_system_image`
`arch* arch`	architectures du processeur où l'artefact s'exécute : `amd64` · `arm64` ; défaut `amd64`. Seule une image OCI en a plusieurs (image multi-architecture) ; un autre format en a une seule (**E008**)

Le format détermine l'exécutant : un `oci_image` s'exécute dans un conteneur, un binaire `native` (Linux) directement, un `incus_vm_image` dans une machine virtuelle, un `incus_system_image` dans un conteneur système. Une image OCI **DEVRAIT** être épinglée par son empreinte, écrite dans le sujet (`registry.example.org/rh/portal@sha256:…`), sinon **W010**.

Architecture. Une réplique ne s'exécute que sur un nœud d'une des architectures de son artefact. Les répliques d'un service ont toutes la même architecture ; pour une image multi-architecture, le planificateur la choisit. Il propose des nœuds de chaque architecture nécessaire, et une infrastructure imposée par la stack **DOIT** en offrir à chaque service dans sa zone (§12).

5.2 Ressources et mise à l'échelle

Déclaration	Sens et contrôles
`memory_request? int, size_unit`	mémoire réservée
`memory_limit? int, size_unit`	mémoire maximale
`cpu_request? int, cpu_unit`	processeur réservé
`cpu_limit? int, cpu_unit`	processeur maximal
`min_replicas? int`	borne basse de la mise à l'échelle
`max_replicas? int`	borne haute ; le dimensionnement retient cette valeur
`cpu_threshold? int`	seuil d'utilisation du processeur, en %
`memory_threshold? int`	seuil d'utilisation de la mémoire, en %

- Une réservation **DOIT** être inférieure ou égale à sa limite (**E014**).
- `min_replicas` et `max_replicas` vont ensemble (**E006**), avec 1 ≤ `min_replicas` ≤ `max_replicas` (**E016**).

5.3 Sondes, métriques et journaux

Déclaration	Sens, défaut et contrôles
`health_check? string { … }`	sonde de santé ; le sujet est le chemin HTTP sondé. Champs `interval? int` (défaut 30), `timeout? int` (défaut 5), `retries? int` (défaut 3). Un `timeout` qui n'est pas inférieur à l'`interval` donne **W011**
`metrics? metrics_provider { … }`	point de collecte ; le sujet est le collecteur (`prometheus`). Champs `path string` et `port? int`
`log_format? log_format`	`json` · `text` (défaut)
`log_level? name`	niveau de journalisation ; défaut `info`

5.4 Exemple

D'après `examples/enterprise-stack/web_api.loaf` :

```
service web_api {
  runtime http.
  replicas 3.
  memory_request 256, mi.
  memory_limit 512, mi.
  cpu_limit 1, core.
  min_replicas 3.
  max_replicas 10.
  cpu_threshold 70.

  artifact "./bin/web_api" {
    sha256 "0000000000000000000000000000000000000000000000000000000000000000".
    format native.
    arch amd64.
  }

  port 8080 { protocol tcp. }
  environment "LOG_DIR", "/var/log/web_api".

  health_check "/health" {
    interval 10.
    timeout 3.
  }
  metrics prometheus { path "/metrics". }
  log_format json.
}
```

SECTION 6

Capacités, providers et consommations

stable

Une **capacité** est un service générique qu'une application consomme sans en choisir l'implémentation : une base de données, un cache. Un **provider** est un service qui fournit une capacité ; il se décrit par un manifeste `service` ordinaire.

6.1 Provider

Déclaration	Sens, défaut et contrôles
`capability? name`	la capacité fournie
`input* name => text`	entrées : valeurs propres à chaque instance, engendrées une fois par consommation : `input password, gen(alnum, 32).`
`readiness* [text]`	commande de disponibilité, exécutée dans l'instance : `readiness ["pg_isready", "-U", input(user)].`
`readiness_interval? int`	délai entre deux essais ; défaut 5
`readiness_timeout? int`	délai maximal ; défaut 30

`readiness_interval` ou `readiness_timeout` sans commande `readiness` donne **E006**. Dans les champs du provider, `input(x)` désigne la valeur de l'entrée `x` de l'instance, `instance(name)` son nom et `consumer(name)` le nom du consommateur (§4.5).

6.2 Contrats des capacités

Le **contrat** d'une capacité est la liste des attributs qu'en offre tout provider. Le vocabulaire déclare un bloc par contrat ; un provider le remplit dans un bloc de même nom, sous son propre nom.

Contrat	Attributs
`database`	`host text`, `port int`, `user text`, `password text`, `dbname text`
`cache`	`host text`, `port int`, `password text`

Un service qui fournit `database` (resp. `cache`) **DOIT** remplir le bloc de ce contrat, et un bloc de contrat n'est admis que pour un service qui fournit la capacité correspondante (**E024**). Une capacité qui n'a pas de contrat dans le vocabulaire peut être déclarée et consommée, mais n'offre aucun attribut aux gabarits de ses consommateurs.

Le provider PostgreSQL de `providers/postgresql.loaf` :

```
service postgresql {
  runtime http.
  capability database.
  input user, gen(alpha, 12).
  input password, gen(alnum, 32).
  input dbname, cat(consumer(name), "_db").
  readiness ["pg_isready", "-U", input(user)].
  readiness_interval 5.
  readiness_timeout 30.
  environment "POSTGRES_USER", input(user).
  environment "POSTGRES_PASSWORD", input(password).
  environment "POSTGRES_DB", input(dbname).

  artifact "postgres:15-alpine" {
    sha256 "0000000000000000000000000000000000000000000000000000000000000000".
    format oci_image.
  }

  port 5432 { protocol tcp. }
}
```

```
database postgresql {
  host instance(name).
  port 5432.
  user input(user).
  password input(password).
  dbname input(dbname).
}
```

6.3 Consommations

Un service déclare chacune de ses dépendances par un sous-bloc `consumes`, dont le sujet est un nom local au service. Une consommation de service déclare aussi le flux réseau correspondant (§9.1).

Déclaration	Sens, défaut et contrôles
`capability? name`	dépendance envers une capacité, résolue vers un provider
`service? service`	dépendance envers un service nommé de l'ensemble
`inject* string => text`	injections : variables d'environnement du consommateur, construites à partir de ce qu'offre le service consommé
`mode? consume_mode`	**expérimental** `request` (défaut) : le consommateur ouvre les connexions ; `push` : le service consommé émet, le consommateur reçoit

Une consommation nomme exactement un de `capability` et `service` (**E013**).

Provider retenu. Hamac calcule le champ `provider` de chaque consommation, qu'aucun manifeste n'écrit. Pour une consommation de service, c'est le service nommé. Pour une consommation de capacité, c'est le provider que la stack impose (§13), sinon le seul service de l'ensemble qui fournit cette capacité. En présence d'une stack, une capacité sans provider, ou fournie par plusieurs services sans choix de la stack, est une erreur (**E023**).

Ce que lisent les gabarits. Dans un sous-bloc `consumes`, `~` désigne la consommation. Elle offre aux gabarits de ses injections :

Attribut	Valeur
`~.database.user`, `~.database.host` …	l'attribut du contrat `database` rempli par le provider retenu, pour une consommation de cette capacité
`~.cache.password` …	de même pour `cache`
`~.host`	le nom d'hôte du service consommé, pour une consommation de service
`~.port`	le premier port, c'est-à-dire le plus petit, que déclare le service consommé

Le langage vérifie chaque attribut contre la déclaration du contrat (**L001** pour un attribut que le contrat ne déclare pas). Une injection dont un attribut n'a pas de valeur, par exemple parce que l'ensemble ne contient aucun provider de la capacité, est refusée (**L014**). Un service qui consomme une capacité se valide avec ses providers, dans le même ensemble.

Deux consommations, extraites de `examples/hr-stack/` :

```
service hr_api {
  consumes db {
    capability database.
    inject "DATABASE_URL",
           fmt("postgresql://{~.database.user}:{~.database.password}@{~.database.host}:
{~.database.port}/{~.database.dbname}").
  }
}

service payroll_report {
  consumes api {
    service hr_api.
    inject "HR_API_URL", fmt("http://{~.host}:{~.port}").
```

```
  }
}
```

Le gabarit de `hr_api` est rempli par le modèle avec les attributs du provider retenu : `{~.database.user}` devient `input(user)` de `postgresql`, que Hamac remplace au déploiement par l'identifiant engendré pour l'instance de `hr_api`.

`hamac resolve` montre ce câblage. Avec `--seed`, réservé aux tests, les valeurs tirées au hasard sont reproductibles :

```
Providers to instantiate:
  postgresql-hr-api-db (postgres:15-alpine@sha256:54e0dd8e…)
    env: POSTGRES_DB=hr_api_db
    env: POSTGRES_PASSWORD=z72SYu7MDwnY8NCONRJsMfb5QNVfDvXs
    env: POSTGRES_USER=pottxvuimhan

Wiring:
  hr_api.db -> postgresql
    inject: DATABASE_URL=postgresql://pottxvuimhan:z72SYu7MDwnY8NCONRJsMfb5QNVfDvXs@postgresql-hr-
api-db:5432/hr_api_db
  payroll_report.api -> hr_api (service)
    inject: HR_API_URL=http://hr-api:8080
```

Le nom d'hôte de l'instance, `postgresql-hr-api-db`, réunit le provider, le consommateur et la consommation ; un second consommateur de `database` recevrait sa propre instance, avec d'autres identifiants.

SECTION 7

Données : stockage, sauvegarde, secrets

expérimental

> **RÈGLE**
> **Stockage objet par défaut.** Les fichiers, pièces jointes et artefacts d'un service **DEVRAIT** être placés dans un stockage objet, consommé comme une capacité (§6). Un service sans état local peut être déplacé sans contrainte. `storage` décrit uniquement les données qui doivent rester sur un système de fichiers.

7.1 Volumes et sauvegarde

Déclaration	Sens, défaut et contrôles
`storage* name { … }`	volume de système de fichiers ; le sujet est son nom
`mount string`	point de montage, chemin absolu (**E031**)
`class storage_class`	`ephemeral` : perdu au redémarrage ; `local` : stocké sur le nœud, perdu avec lui ; `replicated` : le service réplique lui-même ses données entre ses répliques, comme PostgreSQL sous Patroni
`size? int, size_unit`	taille ; obligatoire hors `ephemeral` (**E032**)
`rebuild? rebuild`	pour `local` : `from_backup` · `from_object` · `from_replica` · `none`, la façon de reconstruire le volume sur un autre nœud ; avec `none`, ses données sont perdues avec le nœud
`backup? string { … }`	politique de sauvegarde ; le sujet est la planification, une expression cron
`retention* retention_period => int`	nombre d'instantanés conservés par période : `hourly` · `daily` · `weekly` · `monthly` · `yearly`
`pre* [string]`, `post* [string]`	commandes exécutées dans le service avant et après l'instantané, pour sa cohérence
`verify? verify`	`none` (défaut) · `restore_test` : restauration vérifiée à chaque cycle

Un volume `local` sans `rebuild`, dans un service sans `backup`, est refusé (**E033**) : le sort de ses données en cas de perte du nœud doit être déclaré.

La classe d'un volume dit ce qu'il advient de ses données quand le nœud qui porte le service disparaît :

Volume	Après la perte du nœud
`ephemeral`	rien à reconstruire : le volume repart vide sur un autre nœud
`local`, `rebuild from_backup`	le volume est restauré depuis la dernière sauvegarde du service
`local`, `rebuild from_replica`	le volume est recopié depuis une autre réplique du service
`local`, `rebuild from_object`	le volume est reconstruit depuis le stockage objet du service
`local`, `rebuild none`	les données sont perdues ; le service reste sur son nœud tant que celui-ci existe
`replicated`	le service reconstruit lui-même la réplique perdue à partir des autres

Dans l'exemple (§2), le cache de `payroll_report` est `local` avec `rebuild none` : il se recalcule, sa perte ne coûte que du temps.

7.2 Secrets

Déclaration	Sens, défaut et contrôles
`secret* name { … }`	secret du service ; le sujet est son nom, que désigne `secret(x)`
`ref? string`	référence d'un secret existant, `<backend>:<chemin>[#<clé>]` (**E007** sans `:`) ; backends `openbao` et `sops`

Déclaration	Sens, défaut et contrôles
`generate? gen_kind, int`	secret créé par Hamac au premier déploiement et stocké dans le backend par défaut : sorte de caractères (`alpha` · `alnum`) et longueur
`rotate? int, time_unit`	période de rotation ; absent : pas de rotation automatique
`as? secret_as`	`env` (défaut) : variable d'environnement ; `file` : fichier en lecture seule sous `/run/secrets/<nom>`

- Un secret a exactement un de `ref` et `generate` (**E034**).
- `secret(x)` dans une variable d'environnement ou une injection, sans secret `x` déclaré par le service : **E020**.
- Une variable d'environnement dont le nom se termine par `PASSWORD`, `TOKEN`, `SECRET` ou `_KEY` et dont la valeur est une chaîne littérale non vide donne **W012** : un secret se déclare par `secret`.

7.3 Exemple

Extrait d'un service de forge logicielle :

```
service forge {
  environment "SMTP_PASSWORD", secret(smtp_password).
  environment "SESSION_KEY", secret(session_key).

  secret smtp_password { ref "openbao:kv/mail/smtp#password". }
  secret session_key {
    generate alnum, 64.
    rotate 90, d.
    as file.
  }

  storage repositories {
    mount "/data/git".
    class local.
    size 50, gi.
    rebuild from_backup.
  }
  storage work {
    mount "/tmp/work".
    class ephemeral.
  }
  backup "*/30 * * * *" {
    retention hourly, 24.
    retention daily, 7.
    retention weekly, 4.
    pre ["forgejo", "dump", "--type", "tar"].
    verify restore_test.
  }
}
```

SECTION 8

Patrons

Un **patron** fixe les choix de sécurité d'une organisation : ses zones, le modèle de sécurité qui dit quelle zone peut lire quelle autre, des planchers d'assurance et des exigences de placement. Hamac n'impose aucun modèle.

8.1 Un patron est un module

Un patron est le bloc `pattern` d'un module LOAF, qui en porte la version. La stack le charge par `use` (§3.3) ; un patron qui en complète un autre charge le module de celui-ci. Les patrons de l'ensemble sont ceux des modules que ses fichiers chargent.

Déclaration	Sens, défaut et contrôles
`zone* name { … }`	une zone (§8.2) ; son étiquette, et si elle est une enclave (§8.3), viennent des modèles de sécurité
`assurance* name, axis => assurance_level`	**expérimental** plancher d'assurance d'une zone sur un axe, pour un patron qui doit répondre à une exigence (§8.4)

S'y ajoutent les champs des modules que le patron charge, écrits avec le préfixe de leur module : `blp::levels`, `rbac::role` pour les modèles de sécurité (§8.2), `ha::isolate_zones` pour le placement (§8.5).

Le patron de `examples/hr-stack/patterns/exemple_rh.loaf` :

```
loaf 0.1.
module exemple_rh 1.0.0.
use hamac 0.2.
use hamac_blp 0.2.      % le modèle Bell-LaPadula
use hamac_rbac 0.2.     % les rôles
use hamac_ha 0.2.       % disponibilité et isolation

pattern exemple_rh {
  blp::levels [public, internal, confidential].
  blp::category rh; finance.
  rbac::role rh; comptable; admin.

  zone public { blp::level public. }
  zone internal { blp::level internal. }

  zone rh {
    blp::level confidential.
    blp::category rh; finance.
  }

  % Deux zones ne partagent jamais une machine.
  ha::isolate_zones.
  % Un service à état, de préférence sur trois domaines de panne.
  ha::prefer_stateful_domains 3.
}
```

8.2 Zones et modèles de sécurité

Une **zone** est un ensemble de machines et de services traités de la même façon : même réseau, même filtrage, mêmes exigences. Chaque service est dans une zone (`zone`, §9.1), `default` s'il n'en déclare pas. Dès que les patrons de l'ensemble définissent des zones, la zone d'un service **DOIT** être l'une d'elles (**E042**).

Ouvrir une connexion vers un service revient à le lire. Ce qu'une zone peut lire est la relation `zone_leq(A, B)` : la zone `B` peut lire la zone `A`. Chaque zone se lit elle-même ; le reste de la relation vient des **modèles de sécurité** que chargent les patrons. Sans modèle, un flux ne relie donc que des services d'une même zone. Hamac s'appuie sur cette seule relation pour accepter ou refuser les flux (§9.1) et pour en déduire le filtrage entre zones.

Un modèle de sécurité est un module LOAF qui :

- ajoute ses champs aux patrons, aux zones et aux routes, dans son espace de noms : `blp::level` dans une zone, `rbac::role` dans une route ;
- définit, à partir d'eux, la relation `zone_leq` ;
- signale les incohérences de ses champs par ses propres diagnostics.

Hamac livre avec le vocabulaire des modules prêts à l'emploi : les modèles `hamac_blp` (Bell-LaPadula avec compartiments) et `hamac_rbac` (rôles), le module de placement `hamac_ha`, et le patron `hamac_default`. Le document « Modules livrés avec Hamac » les décrit ; les exemples de ce document en emploient les champs. Une organisation peut écrire d'autres modules, ou combiner les siens avec ceux-ci. Ce modèle minimal, par exemple, donne l'ordre des zones par une simple liste :

```
module ordre 1.0.0.
use hamac 0.2.

namespace ordre {
  block pattern {
    reads* name, name.     % reads b, a : la zone b lit la zone a
  }
  ::zone_leq(A, B) :- pattern(P), P.reads(B, A).
}
```

Un patron qui le charge écrit `ordre::reads bureau, atelier.` : un service de la zone `bureau` peut consommer un service de la zone `atelier`, mais pas l'inverse (**E040**).

8.2.1 Patrons composés

Un patron qui en complète un autre :

```
loaf 0.1.
module anssi_hygiene 1.0.0.
use hamac 0.2.
use hamac_blp 0.2.
use hamac_ha 0.2.

pattern anssi_hygiene {
  blp::levels [public, default, internal, secure].
  ha::isolate_zones.
}
```

```
loaf 0.1.
module anssi_renforce 1.0.0.
use hamac 0.2.
use hamac_blp 0.2.
use anssi_hygiene 1.0.

pattern anssi_renforce {
  blp::levels [public, default, internal, secure].
  blp::enclave_from secure.        % secure et au-delà : derrière une diode
  blp::category finance; rh.

  zone public { blp::level public. }
  zone default { blp::level default. }
  zone internal { blp::level internal. }
  zone rh {
    blp::level internal.
    blp::category rh.
  }
  zone secure { blp::level secure. }

  assurance internal, boot, confidential_vm.
  assurance internal, operator, secnumcloud.
  assurance secure, boot, measured_tpm2.
  assurance secure, physical, site.
}
```

8.3 Enclaves

Hamac filtre toujours les connexions entre zones (§9.1). Une **enclave** va plus loin : c'est une zone derrière une diode, une passerelle physique unidirectionnelle par laquelle les données entrent dans l'enclave et n'en sortent pas.

- À travers une diode, aucune requête ne peut recevoir de réponse. Une consommation entre une enclave et une autre zone **DOIT** donc être de mode `push` (**E041**) : le service consommé émet ce que le consommateur reçoit (réplication, flux d'événements, fichiers).
- Une enclave a son propre plan de contrôle, puisqu'un contrôleur extérieur n'en recevrait pas les rapports. Hamac produit une infrastructure par enclave, et ses nœuds sont isolés. Les manifestes signés entrent par la diode ; l'état de l'enclave n'en sort que par une déclassification explicite.

Qu'une zone soit une enclave découle de la politique de sécurité : c'est la primitive `enclave(Z)`, que produisent les modèles de sécurité (§3.4). `hamac_blp` la déduit d'un seuil, `blp::enclave_from secure.` faisant des enclaves de toutes les zones de niveau `secure` et au-delà ; un modèle d'intégrité (Biba) pourrait de même isoler les zones des automates d'un site industriel.

8.4 Axes d'assurance

expérimental

Trois axes à valeurs ordonnées, fixés par le vocabulaire (énumérations `axis` et `assurance_level`) :

Axe	Niveaux, du plus faible au plus fort
`boot`	`none` < `provider_vtpm` < `confidential_vm` < `measured_tpm2`
`operator`	`any` < `eu` < `secnumcloud` < `self`
`physical`	`third_party` < `dedicated_rack` < `site`

L'assurance d'une machine dit ce qu'on peut affirmer de son intégrité, de qui l'exploite et d'où elle se trouve. Une machine offre un niveau par axe, déclaré par l'inventaire, par le nœud d'infrastructure (§12) ou par le profil qui l'a installée (§11).

> **RÈGLE**
> **Le plus sûr par défaut.** Hamac place chaque zone sur les machines les plus sûres disponibles, en servant d'abord les zones les plus sensibles selon le modèle de sécurité (`zone_leq`). Une zone placée sur des machines moins sûres que d'autres du parc donne un avertissement.

Plancher. Un patron qui doit répondre à une exigence (hébergement certifié, hygiène renforcée, diffusion restreinte) fixe un niveau minimal par zone et par axe : `assurance secure, boot, measured_tpm2.` exige ce niveau de toute machine de la zone `secure` (primitive `floor`, §8.5). Un placement qui ne peut le respecter échoue. Un axe sans plancher n'impose rien. Un niveau qui n'appartient pas à son axe donne **E007** ; une zone que le patron ne définit pas, alors qu'il en définit, donne **E042**.

Pour une topologie qu'il propose, Hamac donne aux nœuds de chaque zone son plancher : sur chaque axe, le plus haut de ceux que fixent les patrons. Le plan l'indique avec les nœuds :

```
Proposed topology:
  1x node [secure] amd64 — 4096Mi RAM, 4 CPU, boot >= measured_tpm2, physical >= site
```

> **NOTE**
> Le planificateur de cette version ne choisit pas encore les machines d'un parc existant selon leur assurance : sur un parc connu (infrastructure imposée, inventaire), ni la règle du plus sûr ni le respect des planchers n'agissent encore sur le placement.

> **NOTE**

> `provider_vtpm` et `confidential_vm` se distinguent sur un point : dans une machine virtuelle classique, c'est l'hyperviseur du fournisseur qui mesure le démarrage ; dans une machine virtuelle confidentielle (AMD SEV-SNP, Intel TDX), le processeur l'atteste et la mesure est opposable à l'hyperviseur.

8.5 Exigences de placement

expérimental

Le **planificateur** (`hamac plan`) choisit la topologie d'une stack : il place chaque réplique de chaque service à déployer sur un nœud de sa zone, dans la capacité des nœuds, avec le moins de nœuds possible. Il respecte en outre les *primitives de placement* que définit l'ensemble de fichiers (§3.4) :

Primitive	Exigence
`separate(S, T)`	les services `S` et `T` ne partagent jamais une machine
`min_domains(S, N)`	les répliques de `S` sont sur au moins `N` domaines de panne
`floor(Z, axe, niveau)`	plancher d'assurance de la zone `Z` (§8.4)

`prefer_separate` et `prefer_min_domains` en sont les préférences : le planificateur les respecte s'il le peut, avant de réduire le nombre de nœuds, et signale chacune de celles qu'il ne respecte pas. Les champs du vocabulaire s'y ramènent (`anti_affinity`, §9.3 ; `assurance` d'un patron), et un module en fournit d'autres déclarations.

Le module `hamac_ha`, livré avec le vocabulaire, en fournit pour la disponibilité et l'isolation : `ha::isolate_zones` au patron (deux zones ne partagent jamais une machine, par des `separate`), `ha::critical` à un service, `ha::stateful_domains` pour les services à état (par `min_domains`).

Quand aucune topologie ne respecte les primitives, le planificateur échoue en donnant celles que viole la topologie qui en viole le moins, avec leur témoin :

```
plan: no placement of the services satisfies min_domains(db,3) (db)
```

SECTION 9

Sécurité, placement et exposition d'un service

9.1 Zone et flux

Déclaration	Sens et contrôles
`zone? name`	la zone du service, définie par les patrons de l'ensemble (**E042** sinon) ; défaut `default`

Chaque consommation de service (`consumes … { service t. }`) déclare un flux réseau, du consommateur vers le service consommé.

> RÈGLE
> **Lecture.** Ouvrir une connexion vers un service revient à le lire. Le modèle de sécurité des patrons **DOIT** donc permettre à la zone du consommateur de lire celle du consommé (`zone_leq` , §8.2). Dans le cas contraire, le flux est refusé (**E040**).

- Seuls les flux déclarés sont ouverts, vers les seuls ports déclarés du consommé. Tout autre trafic entre zones est refusé ; le filtrage est avec état.
- Si les deux zones diffèrent et que l'une d'elles est une enclave (§8.3), le flux **DOIT** être de mode `push` (**E041**).

Dans l'exemple (§2), si `hr_api` (zone `rh`) consommait le portail (zone `public`), il lirait vers le bas, ce que le modèle `hamac_blp` permet :

```
service hr_api {
  consumes site { service portal. }
}
```

Le planificateur ouvre alors ce seul flux, et refuse le reste :

```
DENY public -> rh (no read up)
ALLOW rh -> public (hr_api -> portal)
Stateful: replies to an allowed connection pass.
```

Le flux inverse, `portal` qui consommerait `hr_api` , est refusé dès la validation :

```
portal.loaf:20:5: [E040] portal (zone public) cannot consume hr_api (zone rh):
the security model of the patterns does not let zone public read zone rh; […]
```

Écrire vers le bas, c'est-à-dire qu'un service sensible envoie des données à une zone moins sensible, ne relève pas du réseau : une réponse à une connexion autorisée passe, et ce qu'elle contient est l'affaire de l'application et du contrôle d'accès de ses routes.

9.2 Routes

Un service HTTP déclare l'autorisation de ses routes, vérifiée en bordure avant que la requête ne l'atteigne.

Déclaration	Sens et contrôles
`route* string { … }`	le sujet est le chemin ; `*` en fin de segment couvre tout suffixe (`/salary/*`). Pas d'autre joker
`method* string`	méthodes HTTP ; aucune : toutes
`public`	aucun jeton exigé

Les conditions d'accès sont les champs que les modèles de sécurité ajoutent aux routes (§8.2) : `blp::classification` et `blp::category` pour `hamac_blp` , `rbac::role` pour `hamac_rbac` . Une route `public` n'en porte aucun (**E043**).

Sémantique. Une requête vers une route est autorisée si et seulement si le jeton présenté satisfait les conditions de chacun des modèles.

Dès qu'un service déclare une route, les chemins qu'aucune route ne couvre sont refusés. Si plusieurs routes couvrent un chemin, la plus spécifique s'applique : préfixe le plus long, puis méthodes explicites. Un service qui déclare des routes **DOIT** avoir une zone (**E006**).

Mise en œuvre. Le provider d'autorisation traduit les conditions des routes en règles Biscuit : pour `hamac_blp`, une comparaison d'entiers pour le niveau et une vérification par catégorie ; pour `hamac_rbac`, une jointure sur les rôles. La vérification a lieu en bordure, dans Sozu. Il revient au provider d'établir que cette traduction accepte exactement les accès que les modèles permettent.

Les routes de `examples/hr-stack/hr-api.loaf`, dans la zone `rh` du patron `exemple_rh` :

```
service hr_api {
  zone rh.

  route "/health" { public. }

  route "/employees/*" {
    method "GET".
    blp::classification internal.
    blp::category rh.
    rbac::role rh; admin.
  }

  route "/employees/salaries" {
    blp::classification confidential.
    blp::category rh; finance.
    rbac::role comptable; admin.
  }
}
```

9.3 Placement

expérimental

Déclaration	Sens, défaut et contrôles
`need* name`	capacités que le nœud **DOIT** offrir, telles que l'inventaire les déclare (`public_ipv4`, `ssd`)
`spread? spread`	répartition des répliques : `none` (défaut) · `host` · `failure_domain`
`affinity* service`, `anti_affinity* service`	services à rapprocher ou à séparer ; `anti_affinity` se ramène à la primitive `separate` (§8.5)

Le vocabulaire ne prévoit pas de placement par nom d'hôte.

```
service forge {
  need public_ipv4; ssd.
  spread failure_domain.
  anti_affinity postgresql.
}
```

9.4 Exposition

expérimental

Une exposition publie un port du service en bordure. Sans flux déclaré, c'est la seule façon de rendre un service joignable depuis l'extérieur de sa zone.

Déclaration	Sens, défaut et contrôles
`exposure* string { … }`	le sujet est le nom DNS public
`port int`	un port déclaré par `port` (**E035**)

Déclaration	Sens, défaut et contrôles
`path_prefix? string`	défaut `/`
`protocol? exposure_protocol`	`http` (défaut) · `tcp`
`tls? tls`	`acme` (défaut) · `internal` · `passthrough`

Un service exposé en `http` qui déclare des routes voit chaque requête validée par la bordure selon §9.2.

```
service forge {
  port 3000 {}
  port 2222 {}

  exposure "forge.example.org" {
    port 3000.
    tls acme.
  }
  exposure "ssh.forge.example.org" {
    port 2222.
    protocol tcp.
    tls passthrough.
  }
}
```

SECTION 10

Bundle

stable

Un bundle est une brique réutilisable d'un profil de provisioning : un répertoire qui contient le manifeste `bundle.loaf` et un gabarit Jinja2 de fragment cloud-init. La spécification du provisioning (`doc/typst/provisioning.typ`) décrit où Hamac cherche les bundles, le rendu des gabarits et la fusion des fragments.

```
templates/bundles/dev_workstation/
├── bundle.loaf
├── cloud-init.yaml.j2           # gabarit par défaut
└── cloud-init.<family>.yaml.j2  # variante pour une famille d'OS (optionnel)
```

Déclaration	Sens, défaut et contrôles
`version? string`	version du bundle ; défaut `0.1.0`
`param* name { … }`	paramètre accepté ; le sujet est son nom
`type param_type`	`string` · `int` · `bool` · `list_string` · `list_int` · `object`
`required`	le profil **DOIT** en donner la valeur ; sinon le rendu échoue
`depends_on* bundle`	bundles à appliquer avant celui-ci : un profil qui l'applique les liste avant lui (**E012**)
`lifecycle? lifecycle`	expérimental `install` (défaut) · `converge`

Extrait de `templates/bundles/dev_workstation/bundle.loaf` :

```
bundle dev_workstation {
  version "0.1.0".
  description "Poste de dev minimal : utilisateur sudo, SSH, docker, dotfiles optionnels".

  param username {
    type string.
    required.
  }
  param ssh_public_keys { type list_string. }
  param timezone { type string. }
}
```

Cycle de vie. `lifecycle install` : le bundle est appliqué une fois, au premier démarrage. `lifecycle converge` : il est appliqué à l'installation, puis en cours de vie par l'agent. Un bundle `converge` **DOIT** décrire un état à atteindre ; il est appliqué de façon idempotente, et tout écart constaté est signalé comme une dérive. Ce mode permet de prendre en charge une machine que Hamac n'a pas installée.

SECTION 11

Profil de provisioning

stable

Un profil de provisioning décrit le système installé sur une machine : une image d'OS et une liste de bundles paramétrés. Il ne nomme aucune machine ; un nœud d'infrastructure le désigne par son nom (§12). La spécification du provisioning décrit comment Hamac l'applique : démarrage PXE, rendu et fusion du cloud-init, écriture de l'image sur le disque.

Déclaration	Sens, défaut et contrôles
`os string { … }`	image d'OS (ci-dessous) ; le sujet est son identifiant, à usage humain
`bundles* [bundle]`	les bundles appliqués, dans l'ordre ; un bundle listé deux fois, ou avant un bundle dont il dépend (`depends_on`), donne **E012**
`bundle* bundle { param* name { … } }`	paramètres d'un bundle de la liste : chaque `param` a soit `value? text` (paramètre scalaire), soit `items* [text]` (paramètre liste), exactement un des deux (**E007**). Des paramètres pour un bundle que `bundles` ne liste pas donnent **E007**
`cloud_init_extra? string`	fragment cloud-init, en YAML, fusionné après ceux des bundles
`assurance* axis => assurance_level`	expérimental niveaux d'assurance qu'offre une machine installée par ce profil (§8.4)

Un profil sans bundle ni `cloud_init_extra` ne change rien à l'image installée (**W013**). Un profil nomme ses bundles : il se valide avec eux, dans le même ensemble.

Le sous-bloc `os` porte :

Déclaration	Sens, défaut et contrôles
`url string`	URL de l'image de référence, consommée telle quelle par les machines virtuelles
`sha256 string`	empreinte de l'image, 64 caractères hexadécimaux (**E011**)
`format os_format`	`qcow2` · `raw` · `iso` · `netinstall` ; `netinstall` : noyau et initrd d'installation, suivis d'une installation automatisée
`arch? arch`	architecture de l'image ; défaut `amd64`
`family? name`	famille d'OS ; défaut `debian` . Choisit la variante du gabarit cloud-init des bundles
`raw_zst_url? string`	image disque brute compressée en zstd, dérivée de `url` , pour l'écriture sur une machine physique
`raw_zst_sha256? string`	empreinte de `raw_zst_url`
`bmap_url? string`	carte des blocs utiles (`bmaptool create`) associée à `raw_zst_url`

D'après `provisioning-profiles/dev-workstation-demo.loaf` :

```
provisioning_profile dev_workstation_demo {
  cloud_init_extra "{runcmd: [systemctl enable --now ssh]}".

  os "debian-12-amd64-genericcloud" {
    url "https://cdimage.debian.org/cdimage/cloud/bookworm/latest/debian-12-genericcloud-amd64.qcow
2".
    sha256 "0000000000000000000000000000000000000000000000000000000000000000".
    format qcow2.
    family debian.
  }

  bundles [dev_workstation].

  bundle dev_workstation {
    param username { value "devuser". }
    param timezone { value "Europe/Paris". }
```

```
    param ssh_public_keys {
      items ["ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAIEXAMPLE example@hamac"].
    }
  }
}
```

SECTION 12

Infrastructure

stable

Une infrastructure décrit une topologie : des nœuds, des segments réseau et le filtrage entre zones. Le planificateur de Hamac la produit à partir d'une stack ; un opérateur peut la relire, la figer ou la modifier. Une infrastructure modifiée est revérifiée contre les patrons de la stack avant tout déploiement.

Déclaration	Sens, défaut et contrôles
`backend backend`	`physical` · `incus` · `qemu` · `kubernetes` · `docker_compose`
`min_nodes? int`, `total_ram? int, size_unit`, `total_cpu? int`	exigences minimales
`node* name { … }`	un rôle de nœud ; le sujet est le rôle
`zone? name`	la zone que ces nœuds accueillent ; défaut : le rôle
`arch? arch`	architecture du processeur ; défaut `amd64`. Celle de l'image d'OS de son profil (**E009**)
`count? int`	nombre de nœuds ; défaut 1
`ram? int, size_unit`, `cpu? int`	capacité de chaque nœud
`assurance* axis => assurance_level`	ce qu'offrent ses machines, axe par axe (§8.4) ; un niveau hors de son axe donne **E007**
`os_image? string`, `os_sha256? string`	image d'OS du nœud, ignorée quand le nœud a un profil
`isolated`	nœud réservé à une zone
`provisioning_profile? provisioning_profile`	profil qui installe le nœud (§11)
`segment* name { cidr string. vlan? int. }`	segment réseau
`allow* name, name, service, service`	ouvre les connexions de la zone source vers la zone destination, pour le flux déclaré (consommateur, service consommé) qui le justifie
`deny* name, name`	refus explicite d'une zone source vers une zone destination
`deny_reason* name, name => string`	motif d'un refus
`warning* string`	avertissements du planificateur
`certificate? string`	expérimental référence au certificat de résolution

Le filtrage est avec état. Une règle `allow` nomme toujours le flux déclaré qui la justifie ; tout trafic entre zones qu'aucune règle n'ouvre est refusé.

```
infrastructure si_exemple {
  backend incus.
  min_nodes 3.
  total_ram 48, gi.
  total_cpu 12.

  node worker {
    zone internal.
    count 3.
    ram 16, gi.
    cpu 4.
    assurance boot, confidential_vm.
  }

  segment internal {
    cidr "10.0.2.0/24".
    vlan 20.
  }
```

```
  allow internal, default, forge, postgresql_ha.
  deny default, internal.
  deny_reason default, internal, "no read up".
}
```

SECTION 13

Stack

stable

Une stack réunit les services à déployer ensemble. Ses patrons sont ceux des modules qu'elle charge par `use` (§8) ; sans patron, elle ne définit aucune zone, et ses services ne s'ouvrent de flux qu'à l'intérieur de leur zone. Le patron `hamac_default`, livré avec Hamac, en donne quatre, prêtes à l'emploi (`use hamac_default 0.2.`).

Déclaration	Sens, défaut et contrôles
`accept_experimental`	supprime **W001** pour tout l'ensemble
`service* service { … }`	un service déployé par la stack ; un service de l'ensemble que la stack ne nomme pas est un provider (§6)
`replicas? int`	remplace le `replicas` du service
`consumes* name { provider service. }`	impose le provider d'une consommation du service. La consommation **DOIT** exister (**E022**), et le provider imposé **DOIT** fournir la capacité consommée (**E023**)
`infrastructure? infrastructure`	infrastructure imposée : elle **DOIT** offrir à chaque service un nœud de sa zone et d'une de ses architectures (**E046**) ; absente : calculée par le planificateur

Une stack sans service donne **W002**.

L'ensemble de l'exemple réunit cette stack, le service `forge` des sections précédentes, les patrons `anssi_hygiene` et `anssi_renforce` (§8.2), l'infrastructure `si_exemple` (§12) et trois providers : `postgresql`, un second provider de `database` nommé `postgresql_ha`, et un provider de `cache`.

```
loaf 0.1.
use hamac 0.2.
use anssi_renforce 1.0.

stack si_exemple {
  description "Forge logicielle".
  accept_experimental.

  service forge {
    replicas 2.
    consumes db { provider postgresql_ha. }
  }

  infrastructure si_exemple.
}
```

Deux services de l'ensemble fournissent `database` : sans le choix de la stack, la consommation `db` de `forge` donnerait **E023**.

SECTION 14

Validation

14.1 Trois étages

1. **Langage.** LOAF vérifie la syntaxe, le vocabulaire (blocs et champs déclarés), les sortes des valeurs, les références entre manifestes, la cardinalité des champs et les gabarits. Ses refus portent un code `Lnnn` défini par la spécification de LOAF ; une erreur lexicale ou de syntaxe porte le code **`E001`**.
2. **Ensemble de fichiers.** Les règles du vocabulaire et les contrôles de Hamac portent sur chaque manifeste et sur l'ensemble : contrats des capacités, secrets, volumes, empreintes, quantités. Commande : `hamac validate <fichiers ou répertoires…>`.
3. **Stack.** Quand l'ensemble contient une stack, Hamac vérifie aussi le choix des providers, les rôles qu'exigent les routes et l'infrastructure qu'elle impose. `hamac validate` applique cet étage dès qu'une stack figure dans l'ensemble ; `hamac plan` l'applique toujours, avant de calculer la topologie.

Un ensemble que le langage refuse avant toute évaluation (syntaxe, vocabulaire, sortes) n'est pas soumis aux étages suivants ; les autres diagnostics d'un ensemble sont rapportés ensemble. Chaque diagnostic donne un code, l'emplacement de l'instruction fautive (fichier, ligne, colonne) et un message :

```
manifest.loaf:11:3: [E013] consumes db of web: exactly one of capability and service
```

Une erreur (`E…`, `L…`) fait échouer la commande ; un avertissement (`W…`) ne la fait échouer qu'avec `--strict`.

14.2 Refus du langage

Les fautes que LOAF refuse lui-même portent le code du langage, défini par sa spécification. Les plus fréquentes dans des manifestes Hamac :

Code	Exemple dans un manifeste Hamac
`L001`	bloc ou champ inconnu (`servcie`, `replica 3.`) ; champ calculé écrit à la main (`provider` dans une consommation) ; attribut absent d'un contrat (`{~.database.login}`)
`L002`	valeur hors de son énumération (`format docker.`) ; référence à un manifeste absent de l'ensemble (service d'une stack, bundle d'un profil)
`L006`	deux valeurs pour un champ qui n'en admet qu'une, par exemple écrites par deux fichiers
`L007`	champ obligatoire absent (`runtime`, `sha256` d'un artefact)
`L008`	fichier sans `loaf 0.1.`
`L009`	`use` d'un patron introuvable, ou versions incompatibles
`L014`	gabarit sans valeur : consommation d'une capacité sans provider dans l'ensemble

14.3 Catalogue

Les codes de Hamac sont les suivants ; l'étage est celui de §14. Chaque code est couvert par un cas de `tests/hamac/catalogue.t/cases/`, dont le nom commence par le code. Le validateur **DOIT** refuser chaque cas (avec `--strict` pour les avertissements) en émettant ce code, et **DOIT** accepter les exemples d'`examples/` ; la suite de tests et la CI le vérifient.

Code	Étage	Condition
`E001`	1	erreur lexicale ou de syntaxe ; fichier illisible ; vocabulaire introuvable
`E004`	2	nom de service qui n'est pas un nom DNS
`E006`	2	champ qui en exige un autre : `min_replicas` sans `max_replicas` ou l'inverse ; `readiness_interval` ou `readiness_timeout` sans `readiness` ; route sans `zone` ; catégories `hamac_blp` sans niveaux
`E007`	2	valeur incorrecte : niveau d'assurance hors de son axe ; paramètre de bundle sans, ou avec à la fois, `value` et `items`, que le bundle ne déclare pas, ou donné sous une autre forme que celle

Code	Étage	Condition
		de son type ; paramètres d'un bundle que `bundles` ne liste pas ; référence de secret sans backend
E008	2	artefact autre qu'une image OCI déclaré pour plusieurs architectures
E009	2	nœud d'une autre architecture que l'image d'OS de son profil
E011	2	`sha256` d'un artefact ou d'une image d'OS qui n'est pas 64 caractères hexadécimaux
E012	2	bundle listé deux fois dans `bundles`, ou avant un bundle dont il dépend
E013	2	consommation sans, ou avec à la fois, `capability` et `service`
E014	2	réservation de ressource supérieure à sa limite
E016	2	`replicas` inférieur à 1, `min_replicas` et `max_replicas` incohérents, `ha::stateful_domains` inférieur à 1
E020	2	référence non liée : `secret(x)` sans secret déclaré ; `host(t)` hors d'une consommation de `t` ; terme que Hamac ne sait pas rendre
E022	3	la stack impose le provider d'une consommation que le service ne déclare pas
E023	3	aucun provider pour une capacité, plusieurs sans choix de la stack, ou provider imposé qui ne fournit pas la capacité
E024	2	provider d'une capacité sans le bloc de son contrat, ou bloc de contrat d'un service qui ne fournit pas la capacité
E031	2	`mount` qui n'est pas un chemin absolu
E032	2	`size` absent pour un volume persistant
E033	2	volume `local` sans `rebuild`, dans un service sans `backup`
E034	2	secret sans, ou avec à la fois, `ref` et `generate`
E035	2	port d'une exposition absent des `port` du service
E040	2	flux vers une zone que le modèle de sécurité ne permet pas à la zone du consommateur de lire
E041	2	flux de mode `request` entre une enclave et une autre zone
E042	2	zone ou niveau inconnu : zone d'un service que les patrons ne définissent pas, exigence d'assurance pour une zone que le patron ne définit pas ; pour `hamac_blp`, niveau ou catégorie d'une zone, ou classification d'une route, que les patrons ne déclarent pas
E043	2	route `public` qui exige aussi des conditions d'un modèle (classification, catégories, rôles)
E044	2	`hamac_blp` : route classée au-dessus de la zone de son service, ou catégorie absente de cette zone
E045	3	route qui exige un rôle qu'aucun patron ne déclare
E046	3	infrastructure imposée sans nœud de la zone d'un service et d'une de ses architectures
E050	2	`hamac_blp` : patrons aux niveaux différents
E051	2	`hamac_blp` : zone étiquetée différemment par deux patrons
W001	2	champ expérimental, sans stack qui écrit `accept_experimental`
W002	2	stack sans service
W003	2	manifeste défini dans plusieurs fichiers
W010	2	image OCI non épinglée par empreinte
W011	2	`timeout` d'un `health_check` qui n'est pas inférieur à son `interval`
W012	2	valeur d'environnement qui ressemble à un secret littéral
W013	2	profil de provisioning sans bundle ni `cloud_init_extra`

Financement

« Ce projet a été financé par le Gouvernement dans le cadre du plan France 2030 opéré par Cap Digital et financé par l'Union européenne – NextGeneration EU »
« Ce projet a été financé par le gouvernement dans le cadre de France 2030 »
« Financé par l'Union européenne - Next Generation EU dans le cadre du plan France Relance »







