

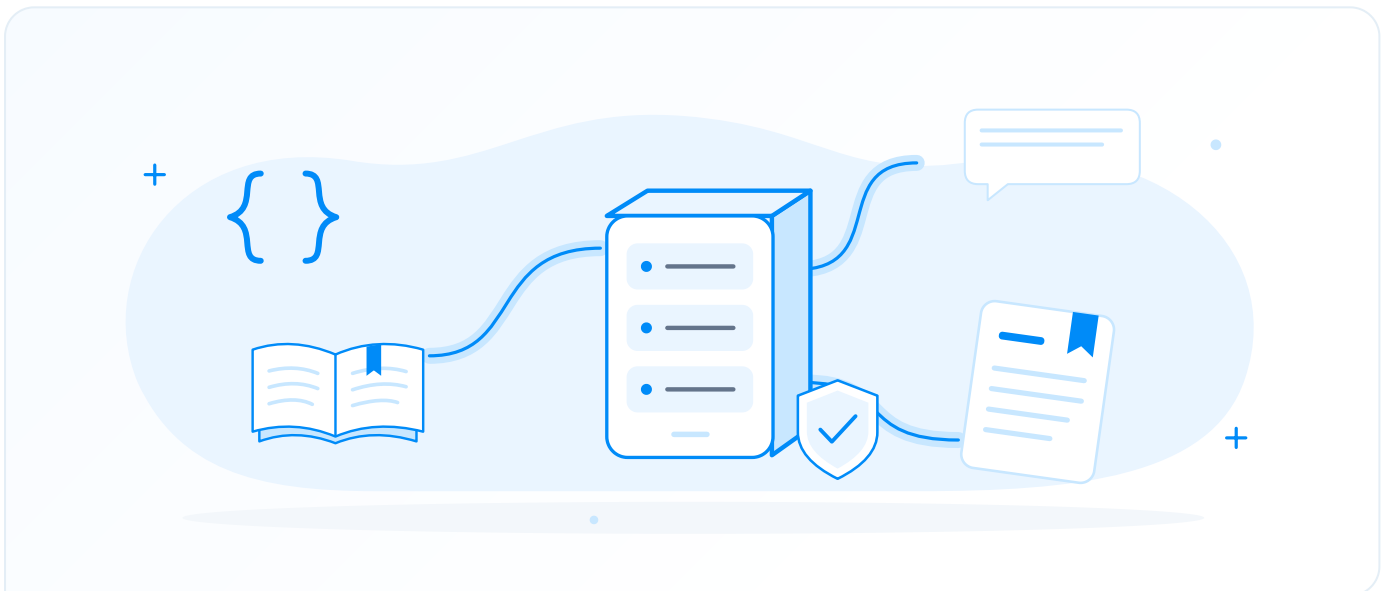
Mini-série

Agents · MCP · intégration système

Mini-série MCP avancé : ressources, HTTP, sécurité et observabilité

Étendre un serveur MCP de support avec ses trois primitives, un transport HTTP authentifié, des sorties structurées et des garde-fous observables.

Auteur **Youness Boutyour**

**DURÉE**

3 épisodes · 45 min /
épisode

NIVEAU

Intermédiaire à avancé

FORMAT

Mini-série · 3 épisodes

Agents · MCP · intégration système · publié le 2026-10-09 · hybrid

Objectifs et livrables

Après un premier serveur MCP local, l'étape suivante consiste à lui donner davantage de capacités et à comprendre ce que change une utilisation sur le réseau. Cette mini-série reprend un petit catalogue de support interne et vous accompagne jusqu'à un serveur HTTP authentifié et observable.

Nous partirons de quatre articles sur les comptes, le VPN, la facturation et la sauvegarde. Le premier épisode ajoutera cinq outils de lecture, des ressources adressables et un prompt réutilisable, puis les fera utiliser par un agent de démonstration. Le deuxième portera sur le transport HTTP et l'authentification ; le troisième sur la validation, les journaux et le contrôle des appels.

Chaque épisode poursuit le même projet. Vous vérifierez d'abord les échanges en mémoire, puis le transport réseau dans un processus séparé. Cette progression permet de comprendre les données échangées, de lire les sorties structurées et d'observer les refus d'accès autant que les appels réussis.

À la fin, vous aurez un serveur de support dont les capacités et les traces sont inspectables. Le jeton utilisé pour la démonstration reste une configuration locale d'essai ; la série précise les adaptations nécessaires à un déploiement d'entreprise. Les repères suivants présentent les responsabilités des primitives et les conventions utilisées dans les commandes.

Trois primitives, trois responsabilités

Le choix de l'interface dépend de qui décide et de ce qui doit être transporté.



Règle pratique : ne remplacez pas une ressource par un outil de lecture si l'application doit contrôler explicitement le contexte chargé.

Les trois primitives ne portent pas la même responsabilité : action, contexte adressable et modèle de message.

Une requête ne doit pas aller directement à la donnée

Chaque étape réduit une classe de risque et laisse une trace corrélable.



Ne journalisez jamais le jeton ; limitez les données de question et préférez des identifiants de corrélation.

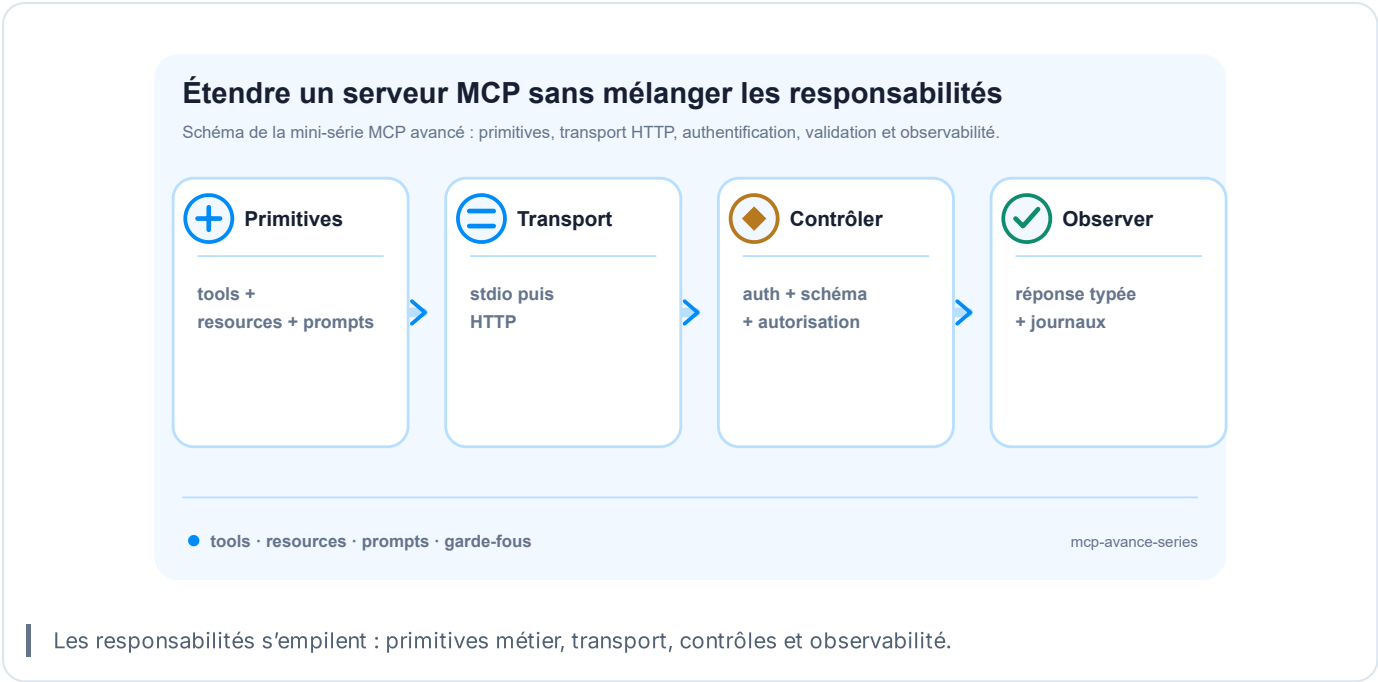
Une requête HTTP doit franchir plusieurs contrôles avant d'atteindre la logique métier.

```
$ python src/in_memory_demo.py
tools: ['search_articles', 'get_article']
resource_templates: ['kb://article/{article_id}']
prompts: ['answer_with_evidence']

structured_content:
  {'result': [{'id': 'kb-002',
    'score': 0.667}]}
```

HTTP sans jeton: 401 Unauthorized

La sortie finale rend visibles le contrat de données et les refus de sécurité.



Convention de travail

Chaque épisode repart du même dépôt. Les commandes sont exécutées depuis sa racine et le résultat affiché est une sortie réelle attendue du code fourni. Le fichier `tests/test_server.py` utilise `Client(mcp)` : cette connexion en mémoire est idéale pour tester les contrats sans démarrer un port. Le test HTTP démarre le serveur dans un processus séparé afin de vérifier le vrai transport et l'en-tête `Authorization`.

1. Les trois primitives : outils, ressources et prompts

1.1 Installer le projet et charger le catalogue

Reprenez les mêmes identifiants d'articles que dans le serveur support et ajoutez les champs utiles aux nouveaux outils. Le catalogue restera le point commun entre les épisodes : changer les données au milieu d'un test rendrait difficile de comparer stdio et HTTP. Les dates de mise à jour serviront à une règle explicite de fraîcheur, pas à certifier l'exactitude du contenu.

Préparez les dossiers de sources, tests, journaux et exports avant de lancer un serveur. Les dépendances supplémentaires couvrent les essais automatisés et le client HTTP ; elles ne créent pas les handlers métier. Utilisez le code final fourni pour assembler le projet, puis examinez les extraits dans l'ordre pour comprendre les responsabilités ajoutées à chaque étape.

Créez le projet et son environnement isolé :

```
Terminal · Bash

mkdir -p mcp-avance-series/{data,src,tests,logs,outputs}
cd mcp-avance-series
python3.11 -m venv .venv
source .venv/bin/activate
python -m pip install --upgrade pip
python -m pip install "mcp[cli] ≥ 2, < 3" "pytest ≥ 8, < 9" "httpx2 ≥ 2, < 3"
```

Le fichier `data/catalog.json` contient quatre articles approuvés. Chaque article possède un identifiant stable, des tags et une date `updated_at` utilisée par le contrôle de fraîcheur :

```
JSON

{
  "id": "kb-003",
  "title": "Comprendre un prélèvement sur une facture",
  "body": "Relevez la référence et la période de facture.",
  "tags": ["facture", "paiement", "finance"],
  "updated_at": "2026-08-01"
}
```

Sortie obtenue

Sortie / exemple

catalog.json: 4 articles approuvés

Commande de vérification

Terminal · Bash

```
python -m json.tool data/catalog.json >/dev/null && echo "catalog: PASS"
```

Sortie / exemple

catalog: PASS

1.2 Déclarer les outils à sortie structurée

Les modèles de sortie définissent des champs et des contraintes que le client pourra examiner. Ils évitent de faire dépendre l'intégration d'un texte de présentation arbitraire. La recherche reste pourtant lexicale : un score situé entre 0 et 1 respecte un contrat numérique, sans devenir une probabilité que l'article réponde à la question.

Cet extrait présente les premières déclarations sur `mcp`. La version étendue regroupe ensuite les déclarations dans une fonction d'enregistrement recevant `server` : ce sont deux états de construction, pas deux serveurs à superposer. Conservez l'assemblage du code final pour que chaque outil soit enregistré une seule fois et que tous les handlers existent avant de démarrer la boucle.

Créez `src/server.py`. La fonction `search_articles` est une action : l'agent décide de l'appeler. Son annotation `→ list[SearchHit]` permet au SDK de publier un `output_schema` et de remplir `structured_content`.

Python

```
from __future__ import annotations

import json
import logging
import re
from pathlib import Path

from pydantic import BaseModel, Field
from mcp.server.mcpserver import MCPServer
```

```

ROOT = Path(__file__).resolve().parents[1]
CATALOG = ROOT / "data" / "catalog.json"
LOG = ROOT / "logs" / "mcp.jsonl"
LOG.parent.mkdir(exist_ok=True)
logging.basicConfig(filename=LOG, level=logging.INFO, format="%(message)s")
mcp = MCPServer("support-catalog", version="1.0.0", instructions="Use search_articles before reading an
article.")

class SearchHit(BaseModel):
    id: str
    title: str
    score: float = Field(ge=0, le=1)

def load_catalog() → list[dict]:
    value = json.loads(CATALOG.read_text(encoding="utf-8"))
    if not isinstance(value, list) or not all(isinstance(item, dict) for item in value):
        raise ValueError("catalog.json doit être une liste d'objets")
    return value

def words(value: str) → set[str]:
    return set(re.findall(r"[a-z0-9ââçêëëîïôûüÿæ-]", value.casefold()))

@mcp.tool()
def search_articles(query: str, limit: int = 3) → list[SearchHit]:
    """Recherche des articles approuvés dans le catalogue de support."""
    if not query.strip():
        return []
    query_words = words(query)
    limit = max(1, min(limit, 5))
    ranked = []
    for article in load_catalog():
        text = " ".join([article["title"], article["body"], *article["tags"]])
        score = len(query_words & words(text)) / max(1, len(query_words))
        if score:
            ranked.append(SearchHit(id=article["id"], title=article["title"], score=round(score, 3)))
    return sorted(ranked, key=lambda item: (-item.score, item.id))[:limit]

```

Le prompt est un gabarit de messages sélectionnable par l'utilisateur ou l'interface. Il n'exécute pas de recherche et ne doit pas être confondu avec une instruction cachée donnée à un modèle.

● ● ● Python

```

@mcp.prompt()
def answer_with_evidence(question: str, article_id: str) → str:
    """Prépare une réponse de support qui cite un article précis."""
    if article_id not in {item["id"] for item in load_catalog()}:
        raise ValueError("article_id absent du catalogue")

```

```

return (
    "Réponds en français à la question suivante en utilisant uniquement "
    f"l'article {article_id}. Cite l'identifiant et indique si l'article "
    f"ne suffit pas. Question : {question.strip()}"
)

if __name__ == "__main__":
    mcp.run()

```

1.3 Ajouter les trois outils de lecture métier

Les trois nouveaux outils réutilisent le catalogue sans le modifier. La liste de catégories vient des tags disponibles ; les suggestions recherchent des articles voisins en excluant celui de départ ; la fraîcheur compare une date de mise à jour à une date de référence. Ils répondent donc à trois questions distinctes, avec des entrées et des sorties adaptées.

Pour la fraîcheur, gardez la même date `as_of` lorsque vous comparez les tests : utiliser la date courante ferait évoluer les résultats sans modification du code. Un statut récent ne prouve pas que la procédure est encore correcte, et un statut à revoir ne la rend pas automatiquement fausse. Les annotations de lecture décrivent l'intention des outils ; les permissions effectives restent à faire respecter par l'implémentation.

Les trois outils suivants complètent `search_articles` et `get_article` sans introduire d'écriture :

- `list_categories` retourne les tags distincts du catalogue ;
- `suggest_related_articles` réutilise les tags d'un article pour proposer des articles voisins, sans retourner l'article courant ;
- `check_article_freshness` compare `updated_at` à une date de référence et renvoie un statut `fresh` ou `review recommended`.

Ajoutez les modèles et les fonctions métier suivants dans `src/server.py`. Dans la fonction `register_server`, le paramètre du serveur s'appelle `server` ; utilisez donc `@server.tool(...)`, `@server.resource(...)` et `@server.prompt(...)` comme dans l'archive finale.

```

Python

from datetime import date
from mcp.types import ToolAnnotations

class FreshnessReport(BaseModel):
    article_id: str
    updated_at: str
    age_days: int = Field(ge=0)
    max_age_days: int = 180
    status: str

```

```

READ_ONLY_CLOSED = ToolAnnotations(
    read_only_hint=True,
    destructive_hint=False,
    idempotent_hint=True,
    open_world_hint=False,
)

@server.tool(annotations=READ_ONLY_CLOSED)
def list_categories() → list[str]:
    """Retourne les catégories du catalogue de support."""
    return sorted({tag for article in load_catalog() for tag in article["tags"]})

@server.tool(annotations=READ_ONLY_CLOSED)
def suggest_related_articles(article_id: str, limit: int = 3) → list[SearchHit]:
    """Retourne des articles liés à un article existant."""
    article = json.loads(article_impl(article_id))
    related = search_articles_impl(" ".join(article["tags"]), limit + 1)
    return [item for item in related if item.id != article_id][:max(1, min(limit, 5))]

@server.tool(annotations=READ_ONLY_CLOSED)
def check_article_freshness(article_id: str, as_of: str = "2026-10-22") → FreshnessReport:
    """Indique si un article doit être révisé selon son âge."""
    article = json.loads(article_impl(article_id))
    updated = date.fromisoformat(article["updated_at"])
    age_days = (date.fromisoformat(as_of) - updated).days
    if age_days < 0:
        raise ValueError("as_of ne peut pas précéder updated_at")
    return FreshnessReport(
        article_id=article_id,
        updated_at=article["updated_at"],
        age_days=age_days,
        status="fresh" if age_days ≤ 180 else "review recommended",
    )

```

Les annotations indiquent au client que ces outils sont idempotents, en lecture seule et limités au catalogue local. Elles restent des indications : l'allowlist et l'authentification sont les contrôles effectifs.

Sortie obtenue

■ Sortie / exemple

```

tools: ['search_articles', 'get_article', 'list_categories', 'suggest_related_articles',
'check_article_freshness']
categories: {'result': ['compte', 'connexion', 'facture', 'fichier', 'finance', 'latence', 'mot de passe',
'paiement', 'restauration', 'réseau', 'sauvegarde', 'vpn']}

```

```
related: {'result': [{'id': 'kb-002', 'title': 'Diagnostiquer une connexion VPN lente', 'score': 0.25}]}
freshness: {'article_id': 'kb-003', 'updated_at': '2026-08-01', 'age_days': 82, 'max_age_days': 180,
'status': 'fresh'}
```

Commande de vérification

Terminal · Bash

```
python src/in_memory_demo.py
```

1.4 Ajouter une ressource adressable

La ressource représente un contenu que le client peut adresser directement, par exemple avec l'URI d'un article connu. Elle complète l'outil de recherche : on découvre un identifiant, puis on peut lire le document correspondant. Le nom du paramètre dans le gabarit d'URI doit correspondre à celui reçu par le handler.

Une URI MCP n'est pas un chemin de fichier que le client peut transformer librement. Le serveur résout l'identifiant dans son catalogue. Comme la déclaration est un gabarit, inspectez aussi les modèles de ressources : une liste vide de ressources fixes ne signifie pas que la lecture de `kb://article/...` est indisponible.

Une ressource n'est pas une fonction que le modèle choisit directement. L'application hôte lit l'URI après avoir obtenu un identifiant fiable. Le placeholder de l'URI doit avoir le même nom que le paramètre Python.

Python

```
@server.resource("kb://article/{article_id}", mime_type="application/json")
def article_resource(article_id: str) → str:
    """Article de support approuvé identifié par son identifiant public."""
    return article_impl(article_id)
```

1.5 Ajouter un prompt contrôlé

Le prompt fournit un gabarit d'instructions nommé que le client peut demander avec des arguments. Son rôle est différent de celui de la ressource : il prépare la manière de répondre, mais ne constitue pas lui-même le corps de l'article. Le contrôle de l'identifiant évite de fabriquer une consigne pour une entrée qui n'existe pas.

Récupérer ce prompt ne déclenche ni une lecture automatique de toutes les sources ni une génération par un modèle. Le client doit assembler les messages, le contenu documentaire et l'appel au générateur s'il en utilise un. Vérifiez donc séparément le texte rendu par le prompt et les données de la ressource auxquelles il fait référence.

Le prompt est un gabarit de messages sélectionnable par l'utilisateur ou l'interface. Il ne réalise pas de recherche et ne doit pas être confondu avec une instruction cachée donnée à un modèle.

● ● ● Python

```
@server.prompt()
def answer_with_evidence(question: str, article_id: str) → str:
    """Prépare une réponse de support qui cite un article précis."""
    wanted = validate_article_id(article_id)
    return (
        "Réponds en français à la question suivante en utilisant uniquement "
        f"l'article {wanted}. Cite l'identifiant et indique si l'article "
        f"ne suffit pas. Question : {question.strip()}"
    )
```

1.6 Construire une boucle d'agent vérifiable

La boucle associe les résultats de plusieurs outils plutôt que de se contenter d'afficher leur liste. Elle vérifie d'abord leur présence, lance la recherche, lit l'article retenu et demande son état de fraîcheur. Suivez l'identifiant entre ces appels : les informations ajoutées doivent concerner le même article que celui utilisé pour répondre.

Le statut de fraîcheur est ici retourné avec la réponse ; il ne bloque pas automatiquement une fiche marquée à revoir. Cette décision reste une règle explicite à définir si vous changez le comportement. De même, `grounded` indique le chemin documentaire utilisé dans cette démonstration, pas une certification de vérité. Une recherche vide doit être traitée avant toute tentative de lecture.

Un agent utile ne s'arrête pas à `tools/list`. Il doit sélectionner un outil, vérifier le contrat de sortie, récupérer la preuve, contrôler la fraîcheur de la source et s'abstenir lorsqu'aucun article ne correspond. Cette boucle reste indépendante du fournisseur LLM : un modèle pourra remplacer le routeur déterministe sans modifier les outils MCP.

Créez `src/support_agent.py` :

● ● ● Python

```
from __future__ import annotations

import asyncio
import json
from typing import Any

from mcp import Client
from server import mcp

async def answer(question: str) → dict[str, Any]:
    async with Client(mcp) as client:
        available = {tool.name for tool in (await client.list_tools()).tools}
        required = {"search_articles", "get_article", "check_article_freshness"}
        if missing := required - available:
            raise RuntimeError(f"outils manquants: {sorted(missing)}")
```

```

search = await client.call_tool("search_articles", {"query": question, "limit": 3})
hits = (search.structured_content or {}).get("result", [])
if not hits:
    return {
        "status": "insufficient_context",
        "answer": "Aucun article approuvé ne permet de répondre avec suffisamment de contexte.",
        "sources": [],
    }

selected = hits[0]
article_result = await client.call_tool("get_article", {"article_id": selected["id"]})
freshness_result = await client.call_tool(
    "check_article_freshness", {"article_id": selected["id"]}
)
article = article_result.structured_content or {}
return {
    "status": "grounded",
    "answer": article["body"],
    "sources": [{"id": article["id"], "title": article["title"]}],
    "freshness": freshness_result.structured_content,
}

```

```

async def main() → None:
    for question in ("Je ne peux plus me connecter à mon compte", "Quel est le prix d'un vol vers Mars ?"):
        print(json.dumps(await answer(question), ensure_ascii=False))

if __name__ == "__main__":
    asyncio.run(main())

```

Sortie obtenue

● ● ● JSON

```

{"status":"grounded","sources":[{"id":"kb-001","title":"Réinitialiser un mot de passe oublié"},"freshness":
{"status":"fresh"}}
{"status":"insufficient_context","sources":[]}

```

L'abstention est une sortie fonctionnelle : elle évite qu'un agent transforme une recherche vide en réponse inventée. La clé `sources` permet aussi à l'interface de rendre la provenance sans parser le texte de l'agent.

Commande de vérification

Terminal · Bash

```
python src/support_agent.py
```

1.7 Tester découverte, appel, lecture et rendu

Testez les primitives séparément : découverte des outils, appel d'un outil, découverte des gabarits de ressources, lecture d'une URI et rendu d'un prompt. Utilisez les scripts de test fournis comme points d'entrée exécutables ; les fragments montrent ensuite les opérations que ces scripts enchaînent. Une liste d'outils correcte ne prouve pas encore que chaque handler renvoie le contenu attendu.

Examinez le contrat structuré autant que le texte affiché. Selon le type de retour, une liste peut être enveloppée sous `result`, alors qu'un objet expose directement ses champs. Ne déduisez pas une panne de cette seule différence. Vérifiez aussi les chemins de refus et la correspondance entre l'identifiant lu, le document retourné et le prompt demandé.

Créez `src/in_memory_demo.py` :

Python

```
import asyncio
from mcp import Client
from server import mcp

async def main() → None:
    async with Client(mcp) as client:
        tools = await client.list_tools()
        resources = await client.list_resources()
        templates = await client.list_resource_templates()
        prompts = await client.list_prompts()
        hits = await client.call_tool("search_articles", {"query": "mot de passe", "limit": 2})
        categories = await client.call_tool("list_categories", {})
        related = await client.call_tool("suggest_related_articles", {"article_id": "kb-001", "limit": 2})
        freshness = await client.call_tool("check_article_freshness", {"article_id": "kb-003"})
        article = await client.read_resource("kb://article/kb-001")
        prompt = await client.get_prompt("answer_with_evidence", {"question": "Je ne peux plus me connecter",
"article_id": "kb-001"})
        print("tools:", [item.name for item in tools.tools])
        print("resources:", [item.uri for item in resources.resources])
        print("resource_templates:", [item.uri_template for item in templates.resource_templates])
        print("prompts:", [item.name for item in prompts.prompts])
        print("structured_content:", hits.structured_content)
        print("categories:", categories.structured_content)
        print("related:", related.structured_content)
        print("freshness:", freshness.structured_content)
        print("resource_prefix:", article.contents[0].text[:48])
        print("prompt_role:", prompt.messages[0].role)
```

```
if __name__ == "__main__":  
    asyncio.run(main())
```

Sortie obtenue

■ Sortie / exemple

```
tools: ['search_articles', 'get_article', 'list_categories', 'suggest_related_articles',  
'check_article_freshness']  
resources: []  
prompts: ['answer_with_evidence']  
structured_content: {'result': [{'id': 'kb-001', 'title': 'Réinitialiser un mot de passe oublié', 'score':  
1.0}]}  
resource_prefix: {"id": "kb-001", "title": "Réinitialiser un mot de passe oublié"  
prompt_role: user
```

Une URI avec un placeholder apparaît dans `list_resource_templates`, pas dans `list_resources`. Vérifiez-le explicitement :

Terminal · Bash

```
python src/in_memory_demo.py  
python -c "import asyncio, sys; sys.path.insert(0, 'src'); from mcp import Client; from server import mcp;  
async def check():\n    async with Client(mcp) as c: print([x.uri_template for x in (await  
c.list_resource_templates()).resource_templates])\n    asyncio.run(check())"
```

Commande de vérification

Terminal · Bash

```
pytest -q
```

■ Sortie / exemple

11 passed

2. Streamable HTTP, authentification et sorties structurées

2.1 Pourquoi changer de transport

Le passage à HTTP change la manière de joindre le serveur, pas les fonctions métier à exposer. Les handlers doivent être enregistrés sur l'instance configurée pour ce transport ; créer une nouvelle instance sans les enregistrer donnerait un serveur joignable mais sans les outils attendus. Gardez le même catalogue pour comparer les réponses avec l'épisode précédent.

Le vérificateur de jeton de l'exemple sert à une démonstration locale. Ses paramètres d'émetteur et de ressource ne mettent pas en place, à eux seuls, un fournisseur OAuth complet. Démarrez le serveur dans un terminal dédié, contrôlez l'adresse et le chemin MCP, puis testez d'abord le refus sans autorisation avant d'introduire le client authentifié.

`stdio` convient lorsque l'hôte lance le serveur localement. Pour un service partagé, le client se connecte à une URL Streamable HTTP et le serveur écoute un port. Le transport ne change pas les verbes MCP : `list_tools`, `call_tool`, `read_resource` et `get_prompt` restent des méthodes du client.

Ajoutez au serveur une vérification de jeton statique uniquement pour la démonstration. `TokenVerifier` ne fabrique pas de jeton et ne remplace pas un fournisseur OAuth : il montre le point d'intégration d'un resource server.

● ● ● Python

```
from pydantic import AnyHttpUrl
from mcp.server.auth.provider import AccessToken, TokenVerifier
from mcp.server.auth.settings import AuthSettings

RESOURCE_URL = "http://127.0.0.1:8000/mcp"

class DemoTokenVerifier(TokenVerifier):
    async def verify_token(self, token: str) → AccessToken | None:
        if token != "demo-read-token":
            return None
        return AccessToken(
            token=token,
            client_id="tutorial-client",
            scopes=["support:read"],
            resource=RESOURCE_URL,
        )

mcp = MCPServer(
    "support-catalog",
    version="1.0.0",
    token_verifier=DemoTokenVerifier(),
    auth=AuthSettings(
        issuer_url=AnyHttpUrl("https://issuer.example.invalid"),
```

```

        resource_server_url=AnyHttpUrl(RESOURCE_URL),
        required_scopes=["support:read"],
        validate_token_resource=True,
    ),
)

```

Dans le dépôt final, cette configuration est activée dans `src/http_server.py`, qui réutilise les fonctions métier sans dupliquer le catalogue :

```

Python

from server import mcp

if __name__ == "__main__":
    mcp.run(transport="streamable-http", host="127.0.0.1", port=8000, json_response=True)

```

2.2 Configurer le client avec un en-tête d'authentification

Le jeton est transmis dans l'en-tête de la requête HTTP par le client configuré, et non dans les arguments de recherche. Cette séparation évite de mélanger l'accès au service avec les données métier. Conservez le même client HTTP pendant la session pour que les échanges utilisent la configuration d'authentification et les délais choisis.

Une erreur HTTP 401 intervient au niveau de l'accès au serveur ; un résultat d'outil en erreur concerne un appel qui a atteint une autre étape du traitement. Distinguez ces situations dans le diagnostic. Un délai de lecture plus long autorise l'attente d'une réponse, mais ne garantit pas qu'une opération métier sera interrompue à ce terme côté serveur.

Ne transmettez pas `headers=` directement à `streamable_http_client` en v2. Construisez un `httpx2.AsyncClient`, configurez ses délais d'attente et donnez-le au transport.

```

Python

import asyncio
import httpx2
from mcp import Client
from mcp.client.streamable_http import streamable_http_client

async def main() → None:
    async with httpx2.AsyncClient(
        headers={"Authorization": "Bearer demo-read-token"},
        timeout=httpx2.Timeout(30.0, read=300.0),
    ) as http_client:
        transport = streamable_http_client("http://127.0.0.1:8000/mcp", http_client=http_client)
        async with Client(transport) as client:

```

```
result = await client.call_tool("search_articles", {"query": "vpn lente"})
if result.is_error:
    raise RuntimeError(result.content)
print("server:", client.server_info.name)
print("structured_content:", result.structured_content)

if __name__ == "__main__":
    asyncio.run(main())
```

Sortie obtenue

■ Sortie / exemple

```
server: support-catalog
structured_content: {'result': [{'id': 'kb-002', 'title': 'Diagnostiquer une connexion VPN lente', 'score': 1.0}]}
```

Sans l'en-tête, le serveur répond `401 Unauthorized` avant de parser la requête MCP. Avec un mauvais jeton, le client reçoit également un refus HTTP et aucun outil n'est exécuté.

2.3 Vérifier le contrat de sortie

Le client commence par examiner le schéma publié, puis valide les valeurs réellement reçues. Cette vérification reste utile même lorsque serveur et client sont développés ensemble : elle rend visibles un champ manquant, un type changé ou une valeur hors contrat. Elle évite de reconstruire une structure fiable en recherchant des morceaux de texte dans une réponse libre.

Contrôlez le statut d'erreur avant de décoder la sortie comme une réussite, puis tenez compte de l'enveloppe des listes. Le test HTTP sans jeton vérifie le refus d'accès ; il ne représente pas un appel MCP métier complet. Conservez des tests distincts pour l'authentification, la découverte et la validation des réponses, afin qu'un échec indique clairement la couche concernée.

L'annotation Pydantic est une interface, pas un commentaire. Le serveur publie le schéma de sortie dans `tools/list`, puis valide la valeur renvoyée. Le client lit `structured_content` pour son code et `content` pour le contexte textuel du modèle ; il ne doit pas extraire un dictionnaire avec une expression régulière.

● ● ● Python

```
async with Client(mcp) as client:
    listing = await client.list_tools()
    tool = next(item for item in listing.tools if item.name == "search_articles")
    assert "output_schema" in tool.model_dump(exclude_none=True)
    result = await client.call_tool("search_articles", {"query": "facture"})
    assert result.is_error is False
    assert result.structured_content["result"][0]["id"] == "kb-003"
```

Commande de vérification

Dans un terminal :

```
Terminal · Bash  
  
python src/http_server.py
```

Dans un second terminal :

```
Terminal · Bash  
  
python src/http_client.py  
curl -i http://127.0.0.1:8000/mcp
```

```
Sortie / exemple  
  
HTTP/1.1 401 Unauthorized  
WWW-Authenticate: Bearer ...
```

Puis lancez le test intégré :

```
Terminal · Bash  
  
pytest -q tests/test_http_auth.py
```

```
Sortie / exemple  
  
2 passed
```

3. Sécurité, observabilité et décision avant déploiement

3.1 Valider les entrées avant la logique métier

La validation rapproche un argument reçu d'une valeur métier autorisée. Retirer les espaces périphériques peut normaliser un identifiant ; cela ne doit pas transformer une entrée inconnue en accès approximatif à un autre document. La liste explicite des identifiants évite ici de construire un chemin disque à partir d'une chaîne fournie par le client.

Vérifiez le refus avant la lecture du contenu, y compris pour une chaîne ressemblant à un chemin. Un type `str` valide dans le schéma ne suffit pas à autoriser sa valeur. Testez ensuite un identifiant admis pour confirmer que le garde-fou ne bloque pas l'usage normal : un serveur qui refuse tout ne démontre pas une validation correctement ciblée.

Un schéma MCP contrôle le type et la présence des arguments, mais les règles métier restent à coder. Dans cette démonstration :

```
Python

ALLOWED_ARTICLE_IDS = {"kb-001", "kb-002", "kb-003", "kb-004"}

def validate_article_id(article_id: str) → str:
    value = article_id.strip()
    if value not in ALLOWED_ARTICLE_IDS:
        raise ValueError("article_id hors périmètre")
    return value
```

La ressource utilise cette fonction avant toute lecture. Elle ne concatène jamais l'identifiant à un chemin de fichier. Pour un catalogue plus grand, remplacez l'ensemble en mémoire par une requête paramétrée et conservez la même règle d'allowlist.

3.2 Ajouter corrélation, latence et refus journalisés

L'identifiant de corrélation relie les événements d'un même appel : début ou décision, erreur éventuelle et durée. Le compteur monotone mesure le temps écoulé, tandis que l'horodatage situe l'événement dans le journal. Leur rôle est différent ; soustraire deux horloges murales pour estimer une durée peut rendre l'analyse fragile.

Le bloc `finally` permet d'enregistrer la durée même lorsqu'une exception survient. Vérifiez donc les traces d'un succès et d'un refus, pas seulement leur affichage côté client. Ces journaux locaux ne constituent pas une infrastructure de traçage distribuée ; gardez-les lisibles, évitez les jetons et contenus sensibles, et ne les envoyez jamais sur `stdout` lorsque le transport `stdio` l'utilise.

Les logs doivent aller vers `stderr` ou un collecteur dédié, jamais vers `stdout` d'un serveur `stdio`. Le dépôt final utilise un logger JSON minimal :

```
Python

import json
import logging
import time
import uuid

logger = logging.getLogger("mcp.audit")
```

```
def audit(event: str, **fields: object) → None:
    logger.info(json.dumps({"event": event, "ts": time.time(), **fields}, ensure_ascii=False))

def correlation_id() → str:
    return uuid.uuid4().hex[:12]
```

Dans le handler de recherche, mesurez la durée et ne journalisez ni le jeton ni le contenu complet de la question si celle-ci peut contenir une donnée personnelle :

```
● ● ● Python

started = time.perf_counter()
request_id = correlation_id()
try:
    result = search_articles(query, limit)
    audit("tool.ok", request_id=request_id, tool="search_articles", results=len(result))
    return result
except ValueError as exc:
    audit("tool.refused", request_id=request_id, tool="search_articles", reason=str(exc))
    raise
finally:
    audit("tool.latency", request_id=request_id, milliseconds=round((time.perf_counter() - started) * 1000,
2))
```

3.3 Contrôler le coût et l'absence d'écriture

Distinguez la taille d'une réponse du volume d'activité. Limiter le nombre d'articles retournés borne un résultat ; cela ne limite ni le nombre de clients ni le nombre d'appels par seconde. De même, une taille maximale de message et un délai de session ne constituent pas un budget de calcul pour chaque outil.

Les contrôles de cette étape confirment le périmètre de lecture et les comportements exercés par les tests. Avant une exposition partagée, les quotas, la concurrence, les autorisations et la gestion des secrets doivent être définis séparément. Comparez les résultats de tests aux garanties effectivement implémentées, sans présenter une liste d'outils déclarés en lecture seule comme une protection complète du service.

Le quota local du dépôt limite une requête à cinq résultats et le serveur n'expose aucun outil de mutation. Pour une action qui modifierait un ticket, utilisez un outil séparé avec un nom explicite, un périmètre d'autorisation différent et une confirmation humaine avant l'appel. N'ajoutez pas une fonction générique `run_command`, `read_file(path)` ou `execute_sql(query)` à un serveur présenté à un agent.

Ajoutez un délai et une limite de taille au niveau HTTP :

 Python

```
mcp.run(  
    transport="streamable-http",  
    host="127.0.0.1",  
    port=8000,  
    json_response=True,  
    max_request_body_size=1024 * 1024,  
    session_idle_timeout=300,  
)
```

Dans un déploiement réel, ajoutez TLS, rotation des jetons, quotas par client, limitation de débit au reverse proxy, contrôle DNS rebinding, métriques et traces distribuées. Le jeton statique de l'exemple ne doit jamais être promu en production.

Sortie obtenue

 JSON

```
{  
  "event": "tool.refused",  
  "request_id": "6fd21e1a4d0c",  
  "tool": "get_article",  
  "reason": "article_id hors  
  périmètre"  
}  
{  
  "event": "tool.latency",  
  "request_id": "6fd21e1a4d0c",  
  "milliseconds": 0.41  
}
```

Commande de vérification

 Terminal · Bash

```
pytest -q tests/test_security.py
```

 Sortie / exemple

```
4 passed
```

Les tests vérifient le refus d'un identifiant inconnu, la borne `limit`, l'absence de secret dans les logs et le refus HTTP sans jeton. Ce sont des tests de contrat : ils ne prouvent pas qu'un fournisseur OAuth, un proxy TLS ou une politique réseau sont correctement configurés.

Dépannage et diagnostics

Le client ne se connecte pas

Vérifiez que le serveur est lancé dans un terminal séparé et que l'URL se termine par `/mcp`. En v2, `Client("http://127.0.0.1:8000/mcp")` choisit Streamable HTTP ; le serveur doit donc utiliser `transport="streamable-http"`.

401 Unauthorized

Vérifiez exactement `Authorization: Bearer demo-read-token`. Un serveur `studio` ne voit pas cet en-tête : son périmètre de sécurité est le processus qui l'a lancé. Le test HTTP est donc indispensable pour valider l'authentification.

structured_content vaut None

Inspectez l'annotation de retour et `tools/list`. Une fonction annotée `→ str` produit une sortie textuelle structurée enveloppée sous `result`, alors qu'un modèle Pydantic publie ses champs. Vérifiez aussi `result.is_error` avant de lire le contenu.

Le prompt ou la ressource n'apparaît pas

Les ressources à URI paramétrée apparaissent sous `list_resource_templates`. Les prompts ne sont présents que si le décorateur `@mcp.prompt()` est exécuté lors de l'import ; gardez le bloc `if __name__ == "__main__":` uniquement autour de `mcp.run()`.

Vérification finale

Exécutez depuis la racine de l'archive :

Terminal · Bash

```
python -m compileall -q src tests
pytest -q
python src/in_memory_demo.py
```

Sortie / exemple

```
11 passed
tools: ['search_articles', 'get_article', 'list_categories', 'suggest_related_articles',
'check_article_freshness']
resource_templates: ['kb://article/{article_id}']
```

```
prompts: ['answer_with_evidence']
structured_content: {'result': [{ 'id': 'kb-001', 'title': 'Réinitialiser un mot de passe oublié', 'score': 1.0}]}
```

Le livrable est correct si la découverte des primitives, l'appel structuré, le transport HTTP, l'authentification et les refus de sécurité sont tous vérifiés. La mini-série ne dépend pas d'un modèle distant : vous pouvez ensuite remplacer le client déterministe par un agent LLM en conservant les mêmes contrats MCP.

Résultats de référence

Contrôle	Résultat de référence
Articles chargés	4
Outils découverts	5 outils de lecture
Catégories	12 tags
Article lié	kb-002 pour kb-001
Fraîcheur	kb-003 : fresh
Agent connu	réponse grounded avec kb-001
Agent hors corpus	insufficient_context
Ressource lue	kb://article/kb-001
Prompt rendu	answer_with_evidence
Authentification valide	200 puis réponse MCP
Authentification absente	401 , aucun outil exécuté
Recherche VPN	kb-002 en tête
Recherche inconnue	liste vide ou abstention explicite

Limites et prolongements

Le score lexical ne remplace pas une évaluation de recherche sur des questions annotées. Le jeton statique ne remplace pas OAuth 2.1, la rotation de secrets et la vérification d'audience. Le journal JSON ne constitue pas à lui seul une observabilité distribuée et les tests locaux ne valident pas la configuration d'un ingress ou d'un pare-feu.

Les prolongements naturels sont l'ajout d'un serveur d'autorisation conforme, de scopes séparés pour lecture et écriture, de métriques OpenTelemetry, d'un catalogue versionné et d'un test de charge. Avant d'exposer une action d'écriture à un agent, ajoutez une étape d'approbation humaine, une idempotence documentée et un mécanisme de compensation.

Références

- [MCP — architecture officielle](#)
- [MCP Python SDK v2 — démarrage](#)
- [MCP Python SDK v2 — ressources](#)
- [MCP Python SDK v2 — clients et transports](#)
- [MCP Python SDK v2 — autorisation HTTP](#)
- [MCP Python SDK v2 — sorties structurées](#)