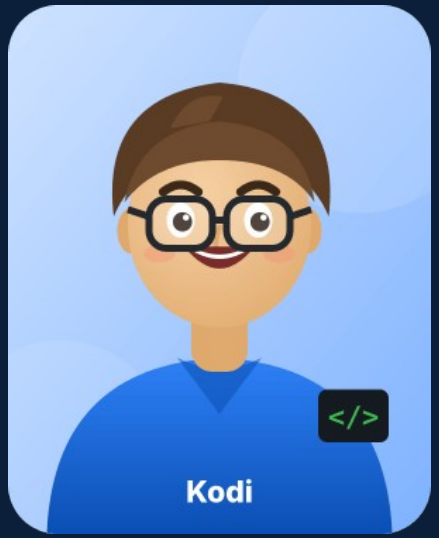
HEF-110 · MISSION 05 / 10

Tests API et contrats

Une API est une promesse faite à d'autres programmes. Un contrat testé garantit que la promesse tient quand le code change.





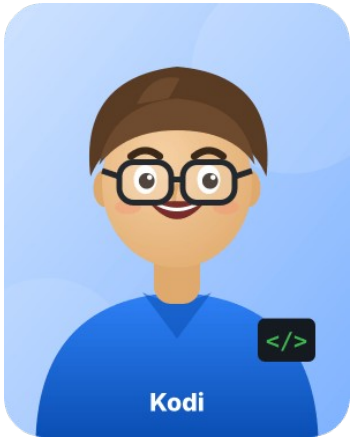
Badge : Gardien du contrat

100 min · intermédiaire

L'histoire

L'équipe mobile de HashBox appelle `GET /cours/HEF-010`. Mardi, Kodi renomme le champ `titre` en `intitule` « pour plus de clarté ». Mercredi, l'application mobile affiche « undefined » partout. Personne n'a cassé de test : l'API répondait bien… autre chose que ce que ses clients attendaient.

> *« Une API, c'est un contrat, dit Maya. Le changer sans le dire, c'est comme changer la forme d'une prise électrique. Un **test de contrat** détecte ce genre de rupture avant que les clients ne la subissent. »*



Tu apprends à tester une API de bout en bout (statuts, formats, erreurs), à écrire un contrat vérifiable, et à faire évoluer une API sans casser ses clients.

Ta mission

À la fin, je serai capable de…

1. tester une API HTTP avec un vrai serveur local et un client
2. vérifier statuts, en-têtes, corps JSON, cas d'erreur
3. écrire un schéma de contrat et le valider automatiquement
4. distinguer changements compatibles et ruptures
5. décrire le principe des tests de contrat côté consommateur (Pact)

Défi d'échauffement

Défi d'échauffement

Une prise électrique a un contrat : forme, tension, fréquence. Un fabricant décide de rendre la prise 2 mm plus large « parce que c'est plus joli ». Qui est cassé ? Comment chaque fabricant d'appareil peut-il vérifier à l'avance que son appareil continuera de s'y brancher ?

Le concept

Que tester dans une API ?

Aspect	Exemples
Statut HTTP	200 OK, 201 créé, 400 requête invalide, 401/403, 404 introuvable, 409 conflit, 429, 500
Corps	structure JSON, types, champs obligatoires et optionnels
En-têtes	`Content-Type`, `Cache-Control`, `Location`
Erreurs	format cohérent, message utile sans détail interne
Sécurité	authentification requise, autorisation, entrées malveillantes (HEF-100)
Idempotence	un `PUT` répété donne le même résultat
Performance	temps de réponse raisonnable

Le concept

Tester avec un vrai serveur local

Pour un test d'API, on démarre le serveur sur **localhost**, port **éphémère** (`0` = le système en choisit un libre), dans un **thread**, et on l'appelle avec un client HTTP. Aucun accès à Internet.

```
setUpClass → démarrer le serveur   |   tests → appels HTTP   |   tearDownClass → a
   rrêter
```

Le contrat d'une API

Un **contrat** décrit précisément ce que l'API promet :

- les **routes** et **méthodes** ;
- la **forme** des requêtes et réponses (champs, types, obligatoires/optionnels) ;

Le concept

Le contrat d'une API (suite)

- les **codes d'erreur** et leur format ;
- les règles de **versionnement**.

Il se décrit en **schéma** (OpenAPI / JSON Schema) que des outils vérifient automatiquement.

Un validateur de schéma simplifié

Un schéma minimal :

```
{"type": "object", "required": ["code", "titre", "places"],
 "properties": {"code": "str", "titre": "str", "places": "int"}}
```

Une réponse **conforme** a tous les champs obligatoires avec les bons types. Les champs **en plus** sont tolérés (principe de robustesse : les clients ignorent ce qu'ils ne connaissent pas).

Le concept

Compatible ou cassant ?

Changement	Verdict
Ajouter un champ **optionnel** dans la réponse	✅ compatible
Ajouter un champ **obligatoire** dans la **requête**	❌ cassant
Supprimer ou renommer un champ de réponse	❌ cassant
Changer un type (`int` → `str`)	❌ cassant
Ajouter une nouvelle route	✅ compatible
Rendre une valeur d'énumération plus restrictive	❌ cassant
Changer un code de statut	❌ généralement cassant

Le concept

Compatible ou cassant ? (suite)

Pour une rupture voulue : **nouvelle version** (`/v2/…`), période de transition, avertissement (en-tête `Deprecation`).

Tests de contrat côté consommateur

Dans une architecture de plusieurs services, le **consommateur** déclare ce qu'il utilise réellement (champs, formats) ; le **fournisseur** exécute ces attentes à chaque changement. Des outils (Pact) automatisent cela. Le principe :

1. le consommateur écrit : « j'ai besoin de `code` (texte) et `titre` (texte) » ;
2. ce contrat est partagé ;
3. le fournisseur rejoue le contrat contre sa vraie API dans son pipeline ;
4. une rupture fait échouer la CI du fournisseur **avant** le déploiement.

Le concept

Tests de contrat côté consommateur (suite)

Avantage : on teste l'**accord** entre services sans lancer tout le système.

Données de test et environnements

- Une API de test **jetable**, avec données fabriquées ;
- jamais la production ;
- authentification de test, jeton factice ;
- nettoyage ou isolation par test.

🧠 ***Astuce*** *: écris le test d'API* ***du point de vue du client*** *: il ne sait rien de ton code, seulement de ta promesse.*

Exemple pas à pas

Problème : démarrer une mini-API HTTP locale, la tester (statuts, JSON, erreurs), valider son contrat avec un schéma, et détecter une rupture de contrat.

```
import io, json, threading, unittest, urllib.error, urllib.request
from http.server import BaseHTTPRequestHandler, HTTPServer

COURS = {"HEF-010": {"code": "HEF-010", "titre": "Algorithmes", "places": 20}}

class Handler(BaseHTTPRequestHandler):
    titre_champ = "titre"                                  # sert plus loin à sim
   uler une rupture

    def _repondre(self, statut, corps, entetes=None):
        donnees = json.dumps(corps).encode()
        self.send_response(statut)
        self.send_header("Content-Type", "application/json; charset=utf-8")
        self.send_header("Content-Length", str(len(donnees)))
        for k, v in (entetes or {}).items():
```

Exemple pas à pas

Exemple pas à pas (suite)

```
            self.send_header(k, v)
        self.end_headers()
        self.wfile.write(donnees)

    def do_GET(self):
        morceaux = self.path.strip("/").split("/")
        if morceaux[0] == "cours" and len(morceaux) == 2:
            cours = COURS.get(morceaux[1])
            if cours is None:
                return self._repondre(404, {"erreur": "cours introuvable"})
            visible = dict(cours)
            if self.titre_champ != "titre":                # simule le renommage
du champ
                visible[self.titre_champ] = visible.pop("titre")
            return self._repondre(200, visible)
        self._repondre(404, {"erreur": "route inconnue"})

    def do_POST(self):
        if self.path != "/cours":
```

Exemple pas à pas

Exemple pas à pas (suite)

```
            return self._repondre(404, {"erreur": "route inconnue"})
        taille = int(self.headers.get("Content-Length", 0))
        try:
            corps = json.loads(self.rfile.read(taille))
        except json.JSONDecodeError:
            return self._repondre(400, {"erreur": "JSON invalide"})
        if not isinstance(corps, dict) or not isinstance(corps.get("code"), str) o
    r not isinstance(corps.get("titre"), str):
            return self._repondre(400, {"erreur": "code et titre (texte) requis"})
        if corps["code"] in COURS:
            return self._repondre(409, {"erreur": "code déjà utilisé"})
        COURS[corps["code"]] = {"code": corps["code"], "titre": corps["titre"], "p
    laces": int(corps.get("places", 10))}
        self._repondre(201, COURS[corps["code"]], {"Location": f"/cours/{corps['co
    de']}"})

     def log_message(self, *args):                          # silence
        pass
```

Exemple pas à pas

Exemple pas à pas (suite)

```
# ---------- mini-validateur de contrat ----------
TYPES = {"str": str, "int": int}

def valider(corps, schema):
    erreurs = []
    for champ in schema["required"]:
        if champ not in corps:
            erreurs.append(f"champ obligatoire absent : {champ}")
        elif not isinstance(corps[champ], TYPES[schema["properties"][champ]]) or i
   sinstance(corps[champ], bool):
            erreurs.append(f"type incorrect pour {champ}")
    return erreurs

CONTRAT_COURS = {"required": ["code", "titre", "places"], "properties": {"code": "
   str", "titre": "str", "places": "int"}}

def appel(methode, base, chemin, corps=None):
    donnees = None if corps is None else (corps if isinstance(corps, bytes) else j
   son.dumps(corps).encode())
```

Exemple pas à pas (suite)

```
    requete = urllib.request.Request(base + chemin, data=donnees, method=methode,
   headers={"Content-Type": "application/json"})
    try:
        with urllib.request.urlopen(requete, timeout=5) as r:
            return r.status, dict(r.headers), json.loads(r.read())
    except urllib.error.HTTPError as e:
        return e.code, dict(e.headers), json.loads(e.read())

class TestApiCours(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        cls.serveur = HTTPServer(("127.0.0.1", 0), Handler)           # port 0 = p
   ort libre choisi par le système
        cls.base = f"http://127.0.0.1:{cls.serveur.server_address[1]}"
        threading.Thread(target=cls.serveur.serve_forever, daemon=True).start()

    @classmethod
    def tearDownClass(cls):
        cls.serveur.shutdown()
```

Exemple pas à pas

Exemple pas à pas (suite)

```
        cls.serveur.server_close()

    def test_lire_un_cours_existant(self):
        statut, entetes, corps = appel("GET", self.base, "/cours/HEF-010")
        self.assertEqual(statut, 200)
        self.assertTrue(entetes["Content-Type"].startswith("application/json"))
        self.assertEqual(valider(corps, CONTRAT_COURS), [])                # res
    pecte le contrat
        self.assertEqual(corps["titre"], "Algorithmes")

    def test_cours_inconnu_donne_404_avec_message(self):
        statut, _, corps = appel("GET", self.base, "/cours/NOPE")
        self.assertEqual((statut, corps), (404, {"erreur": "cours introuvable"}))

    def test_creer_un_cours_renvoie_201_et_location(self):
        statut, entetes, corps = appel("POST", self.base, "/cours", {"code": "HEF-
    777", "titre": "Nouveau", "places": 5})
        self.assertEqual(statut, 201)
        self.assertEqual(entetes["Location"], "/cours/HEF-777")
```

Exemple pas à pas

Exemple pas à pas (suite)

```
        self.assertEqual(valider(corps, CONTRAT_COURS), [])

    def test_creer_deux_fois_donne_409(self):
        appel("POST", self.base, "/cours", {"code": "HEF-888", "titre": "Doublon"}
    )
        statut, _, corps = appel("POST", self.base, "/cours", {"code": "HEF-888",
    "titre": "Doublon"})
        self.assertEqual(statut, 409)

    def test_requetes_invalides_donnent_400(self):
        for corps in (b"pas du json", {"code": 12, "titre": "x"}, {"titre": "sans
    code"}, [1, 2]):
            with self.subTest(corps=corps):
                statut, _, reponse = appel("POST", self.base, "/cours", corps)
                self.assertEqual(statut, 400)
                self.assertIn("erreur", reponse)

resultat = unittest.TextTestRunner(stream=io.StringIO()).run(unittest.defaultTestL
    oader.loadTestsFromTestCase(TestApiCours))
```

Exemple pas à pas (suite)

```
print("tests d'API :", resultat.testsRun, "| réussite :", resultat.wasSuccessful()
   )
assert resultat.wasSuccessful(), resultat.failures + resultat.errors

# ---------- rupture de contrat : le champ « titre » est renommé « intitule » ----
   ------
Handler.titre_champ = "intitule"
resultat = unittest.TextTestRunner(stream=io.StringIO()).run(unittest.defaultTestL
   oader.loadTestsFromTestCase(TestApiCours))
print("après le renommage du champ :", len(resultat.failures), "échec(s) de contra
   t détecté(s)")
assert len(resultat.failures) >= 1
```

Lecture :

- Le serveur écoute sur `127.0.0.1` et un **port libre** ; aucun accès au réseau externe.
- Les tests parlent comme un **client** : méthode, chemin, statut, en-têtes, JSON.

Exemple pas à pas (suite)

- `valider(corps, CONTRAT_COURS)` encode le contrat : champs obligatoires et types.
- Après le **renommage** de `titre` en `intitule`, le test de contrat échoue : c'est exactement la rupture qui avait cassé l'application mobile, mais détectée avant le déploiement.

À toi, avec aide

À toi, avec aide

Écris `comparer_contrats(ancien, nouveau)` qui compare deux schémas (comme `CONTRAT_COURS`) pour une **réponse** d'API et classe les changements en `compatibles` et `cassants` selon le tableau du cours :

- champ ajouté (optionnel ou obligatoire dans la réponse) → compatible ;
- champ retiré → cassant ;
- type modifié → cassant.

Teste avec 4 paires de schémas.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Compare les ensembles de clés de `properties`.

Indice 2

Un champ présent dans les deux mais avec un type différent est cassant.

Indice 3

Renvoie deux listes de messages.

À toi, avec aide · réponse

Réponse

```
def comparer_contrats(ancien, nouveau):
    compat, cassant = [], []
    a, n = ancien["properties"], nouveau["properties"]
    for champ in sorted(set(n) - set(a)):
        compat.append(f"champ ajouté : {champ}")
    for champ in sorted(set(a) - set(n)):
        cassant.append(f"champ retiré : {champ}")
    for champ in sorted(set(a) & set(n)):
        if a[champ] != n[champ]:
            cassant.append(f"type modifié : {champ} ({a[champ]} → {n[champ]})")
    return compat, cassant

base = {"required": ["code", "titre"], "properties": {"code": "str", "titre": "str
   "}}
assert comparer_contrats(base, {"required": ["code", "titre"], "properties": {"cod
   e": "str", "titre": "str", "places": "int"}}) == (["champ ajouté : places"], []
   )
assert comparer_contrats(base, {"required": ["code"], "properties": {"code": "str"
   }}) == ([], ["champ retiré : titre"])
```

À toi, avec aide · réponse

Réponse (suite)

```
assert comparer_contrats(base, {"required": ["code", "titre"], "properties": {"cod
   e": "int", "titre": "str"}}) == ([], ["type modifié : code (str → int)"])
assert comparer_contrats(base, base) == ([], [])
print("4 comparaisons correctes")
```

Brancher cette fonction dans la CI (sur l'ancien contrat versionné et le nouveau) bloque automatiquement les ruptures involontaires.

À toi, en solo

Test de contrat côté consommateur. Écris :

1. un « consommateur » `AffichageCours` qui n'utilise que `code` et `titre` d'une réponse ;
2. un **contrat de consommateur** (dictionnaire) listant exactement les champs et types dont il dépend ;
3. une fonction `verifier_fournisseur(contrat, reponse)` qui vérifie la réponse du fournisseur ;
4. trois versions de fournisseur : A (ajoute `places`), B (renomme `titre`), C (ajoute `places` et change `code` en entier) ;
5. montre que A **passe** (pas de rupture pour ce consommateur), B et C **échouent**, et explique pourquoi un contrat « du consommateur » est plus précis que le schéma complet du fournisseur.

À toi, en solo · réponse

Pistes de correction

```
CONTRAT_AFFICHAGE = {"code": "str", "titre": "str"}          # ce dont le consomma
   teur dépend, ni plus ni moins

def verifier_fournisseur(contrat, reponse):
    problemes = []
    for champ, type_attendu in contrat.items():
        if champ not in reponse:
            problemes.append(f"manque {champ}")
        elif type(reponse[champ]).__name__ != type_attendu:
            problemes.append(f"{champ} : {type(reponse[champ]).__name__} au lieu d
   e {type_attendu}")
    return problemes

class AffichageCours:
    def ligne(self, reponse):
        return f"{reponse['code']} — {reponse['titre']}"

A = {"code": "HEF-010", "titre": "Algorithmes", "places": 20}
B = {"code": "HEF-010", "intitule": "Algorithmes", "places": 20}
```

À toi, en solo · réponse

Pistes de correction (suite)

```
C = {"code": 10, "titre": "Algorithmes", "places": 20}

assert verifier_fournisseur(CONTRAT_AFFICHAGE, A) == []
assert verifier_fournisseur(CONTRAT_AFFICHAGE, B) == ["manque titre"]
assert verifier_fournisseur(CONTRAT_AFFICHAGE, C) == ["code : int au lieu de str"]
assert AffichageCours().ligne(A) == "HEF-010 — Algorithmes"
print("A compatible ; B et C rompent le contrat du consommateur")
```

Le contrat du consommateur ne mentionne **pas** `places` : le fournisseur peut le modifier ou le supprimer sans casser ce client. À l'inverse, avec le schéma complet du fournisseur, chaque modification paraîtrait dangereuse. Le contrat du consommateur dit **qui dépend de quoi**.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Tester seulement le cas heureux (200)	Les erreurs sont aussi le contrat	Tester 400, 401/403, 404, 409 avec leur format
Tester l'API en production	Données réelles modifiées	Serveur de test avec données fabriquées
Renommer ou retirer un champ « sans impact »	Les clients sont cassés	Compatibilité ascendante, versionnement, test de contrat
Ignorer le format des erreurs	Les clients ne peuvent pas les traiter	Un format d'erreur unique et testé

Mini-boss

Mini-boss

Contrat et tests de l'API de HashBox. Implémente (avec `http.server`) une API de membres : `GET /membres/<id>`, `POST /membres`, `PUT /membres/<id>` (idempotent), `DELETE /membres/<id>`, avec authentification par jeton (`Authorization: Bearer …`).

1. écris le **contrat** (schémas de requêtes/réponses, codes d'erreur, format unique d'erreur) ;
2. écris 15+ tests d'API : statuts, en-têtes, corps, erreurs, idempotence du `PUT`, authentification manquante/incorrecte (401), droits (403) ;
3. écris le test de **non-régression de contrat** (`comparer_contrats` sur les schémas versionnés) ;
4. écris un **contrat de consommateur** pour une application mobile ;
5. simule 3 évolutions (ajout de champ, renommage, ajout d'un champ obligatoire dans la requête) et classe-les.

Mini-boss · réponse

Pistes de correction

Un bon travail : serveur lancé en `setUpClass` sur un port libre, client `urllib`, assertions précises sur statut + en-têtes + corps ; `PUT` répété deux fois donne la même réponse ; 401 sans jeton, 403 avec un jeton sans droit ; format d'erreur uniforme `{"erreur": "…"}`. Évolutions : ajout d'un champ de réponse = compatible ; renommage = cassant ; champ obligatoire de requête = cassant. Le contrat de consommateur ne contient que les champs réellement lus par l'application mobile.

Quiz éclair

Quiz éclair

1. Un changement compatible d'une API est :

A. le renommage d'un champ de réponse
B. le changement du type d'un champ
C. l'ajout d'un champ optionnel dans la réponse

2. Le port `0` pour un serveur de test signifie :

A. le système choisit un port libre
B. aucun port
C. le port de production

3. Un test de contrat côté consommateur :

A. remplace tous les tests

Quiz éclair

Quiz éclair (suite)

B. déclare ce dont le client dépend et le fait vérifier par le fournisseur
C. teste la vitesse du réseau

4. 🔁 *Révision mission 04* : Un test d'intégration vérifie surtout :

A. la vitesse du processeur
B. l'orthographe des messages
C. que des composants réels fonctionnent ensemble

5. 🔁 *Révision mission 04* : Pour isoler les tests de base de données, on peut :

A. créer une base neuve (ou annuler la transaction) pour chaque test
B. partager une seule base entre tous les tests
C. tester en production

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Quelle API (interne ou externe) utilises-tu dont un changement silencieux te casserait ?
2. Pourquoi vaut-il mieux que le consommateur déclare son contrat plutôt que le fournisseur devine ses clients ?

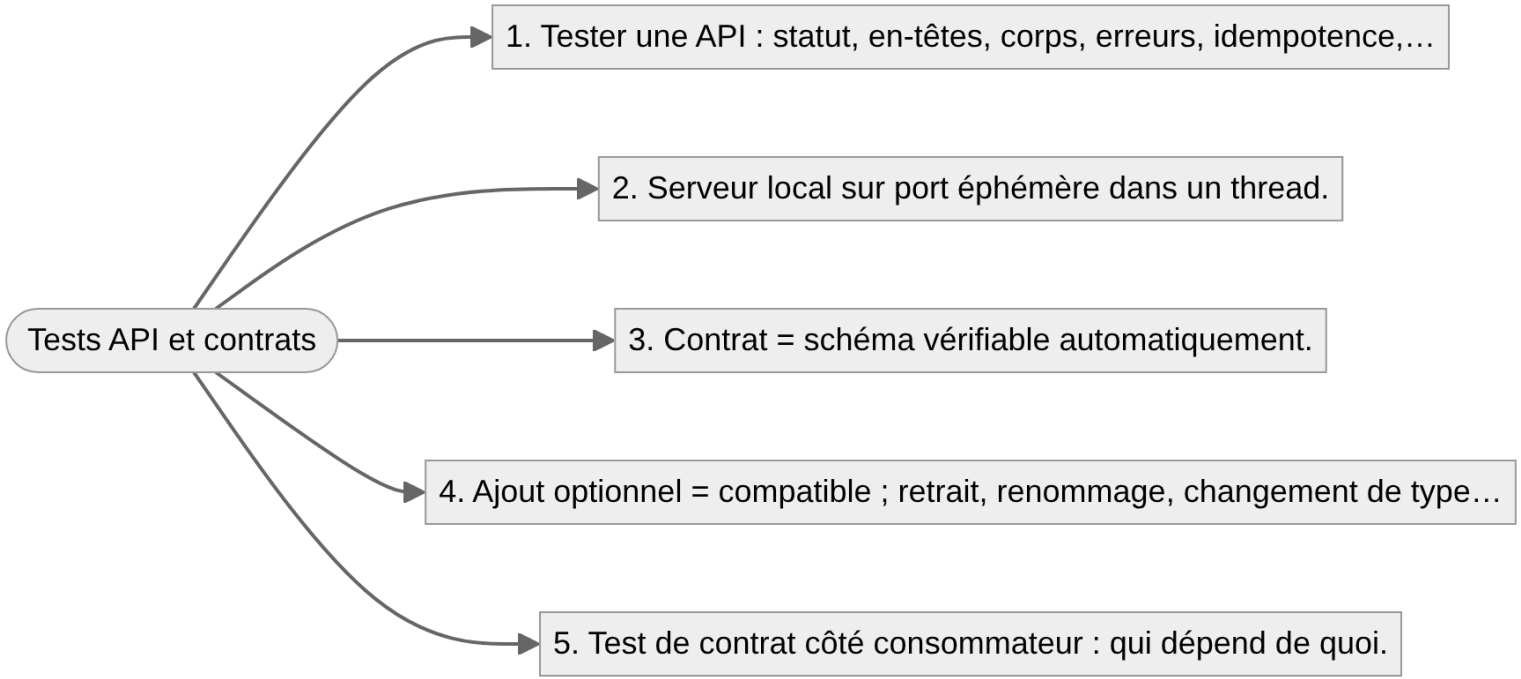
À retenir

À retenir

1. Tester une API : statut, en-têtes, corps, erreurs, idempotence, sécurité.
2. Serveur local sur port éphémère dans un thread.
3. Contrat = schéma vérifiable automatiquement.
4. Ajout optionnel = compatible ; retrait, renommage, changement de type = cassant.
5. Test de contrat côté consommateur : qui dépend de quoi.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Gardien du contrat

Prochaine mission : Tests E2E

