

Contents

Étape 5 : analyse d'un incident	1
Le scénario	1
Prérequis	2
Étape 5.0 : reproduire l'incident (2 min)	2
Étape 5.1 : le symptôme (3 min)	2
Étape 5.2 : les logs du proxy et du front (4 min)	3
Étape 5.3 : l'état des services (2 min)	3
Étape 5.4 : les logs de l'API (3 min)	3
Étape 5.5 : la cause racine (3 min)	4
Étape 5.6 : corriger et vérifier (3 min)	4
Étape 5.7 : rédiger le post-mortem (10 min)	5
Je valide avec le formateur	6
Questions de réflexion	6

Étape 5 : analyse d'un incident

Quand : jour 4, fin d'après-midi | **Durée** : ~25 min **Objectif** : face à une production en panne, remonter **avec méthode** du symptôme à la cause, corriger, vérifier, puis rédiger un **post-mortem sans blâme** dans docs/INCIDENTS.md.

Le scénario

14h. Une PR vient d'être fusionnée ; le pipeline a déployé la nouvelle version. Au même moment, quelqu'un a modifié à la main le fichier .env de la production. Quelques minutes plus tard, les messages arrivent : « La page s'affiche, mais la liste des tâches ne se charge plus », « J'ai une erreur 502 ».

Vous êtes d'astreinte. À vous de jouer.

Vocabulaire

Terme	Signification
502 Bad Gateway	Réponse d'un proxy qui n'a pas obtenu de réponse valable du serveur placé derrière lui : « j'ai transmis, mais derrière, ça ne répond pas »
Cause racine	La cause sur laquelle on peut agir pour que l'incident ne se reproduise pas
Post-mortem sans blâme	Le compte rendu écrit d'un incident, centré sur les processus , jamais sur les personnes
Les « 5 pourquoi »	Demander « pourquoi ? » plusieurs fois de suite, jusqu'à une cause qu'on peut corriger

Prérequis

- L'étape 4, partie A, est terminée : la stack taskflow-prod tourne dans ~/taskflow-prod et répond en HTTPS.
- La variable FORMATION pointe vers votre clone du dépôt de la formation (export FORMATION=~/formation-dwwwm-devops).
- Une feuille ou un fichier texte pour noter **l'heure et chaque commande** : ce sera la chronologie du post-mortem.

Si la reproduction est impossible (stack absente, poste en panne), travaillez avec les journaux fournis dans [incident/logs/](#) : chaque étape ci-dessous indique le fichier à ouvrir. Les logs sont en heure UTC (temps universel, deux heures de moins que Paris en octobre).

Étape 5.0 : reproduire l'incident (2 min)

Lancez le script **sans le lire** (il contient la solution) :

```
date +%H:%M
bash "$FORMATION/tp/fil-rouge-taskflow/incident/declencher-incident.sh"
```

Résultat attendu : quelques lignes de Docker Compose, puis :

==> Un « déploiement » vient d'avoir lieu. Attendez 30 secondes, puis ouvrez <https://taskflow.localhost> (ou votre port HTTPS) et suivez l'énoncé de l'étape 5.

Notez l'heure : c'est le début de l'incident.

Si ça ne marche pas - compose.prod.yaml introuvable : votre dossier de déploiement n'est pas ~/taskflow-prod. Précisez-le : DEPLOY_DIR=/chemin/absolu
bash "\$FORMATION/tp/fil-rouge-taskflow/incident/declencher-incident.sh".
- l'incident est déjà déclenché : il est en cours, passez à l'étape 5.1. - Rien ne fonctionne : utilisez les journaux de [incident/logs/](#).

Étape 5.1 : le symptôme (3 min)

Ouvrez <https://taskflow.localhost> (rechargez avec Ctrl+Shift+R), puis dans le terminal :

```
curl -sk --resolve taskflow.localhost:443:127.0.0.1 https://taskflow.localhost/health
```

(Avec HTTPS_PORT=8443 : --resolve taskflow.localhost:8443:127.0.0.1 https://taskflow.localhost)

Résultat attendu : la page s'affiche, mais avec le message « Erreur de l'API (HTTP 502) » et le pied de page « API injoignable ». curl renvoie une page HTML 502 Bad Gateway signée nginx.

Notez : **qu'est-ce qui marche** (la page, le style) et **qu'est-ce qui ne marche pas** (/api/..., /health) ?

Sans reproduction : [incident/logs/proxy.log](#). Quelle est la première ligne en 502 ? Quelles URL répondent encore 200 ?

Pourquoi commencer par là ? Le symptôme délimite la zone de recherche : si la page s'affiche, le proxy et le front fonctionnent ; le problème est **derrière** eux.

Étape 5.2 : les logs du proxy et du front (4 min)

```
cd ~/taskflow-prod
docker compose -f compose.prod.yaml logs --since 15m proxy | tail -n 20
docker compose -f compose.prod.yaml logs --since 15m front | tail -n 20
```

Résultat attendu :

- proxy : des lignes d'accès avec le code 502, mais **aucune** ligne [error] : le proxy a bien joint le front, c'est le front qui lui a répondu 502 ;
- front : des lignes [error] du type api could not be resolved (3: Host not found) ou connect() failed (111: Connection refused) while connecting to upstream ... upstream: "http://...:3000/..." : c'est le front qui n'arrive pas à joindre **l'API**.

Sans reproduction : [incident/logs/proxy.log](#) et [incident/logs/front.log](#).

Pourquoi ? Un 502 dit seulement « derrière moi, ça ne répond pas ». On suit la chaîne proxy → front → api → db maillon par maillon, et chaque log désigne le maillon suivant.

Étape 5.3 : l'état des services (2 min)

```
docker compose -f compose.prod.yaml ps
```

Résultat attendu : db, front et proxy sont Up ... (healthy), mais api est en Restarting (1) ... (ou alterne entre Up Less than a second et Restarting). Relancez la commande deux ou trois fois pour le constater.

Sans reproduction : [incident/logs/compose-ps.txt](#). Comparez aussi les **tags** des images de api et de front : que remarquez-vous ?

Pourquoi ? Restarting (1) : le processus s'arrête avec le code d'erreur 1, et Docker le relance sans cesse (restart: unless-stopped). L'API n'est jamais healthy : c'est le maillon cassé.

Étape 5.4 : les logs de l'API (3 min)

```
docker compose -f compose.prod.yaml logs --since 15m api | tail -n 5
```

Résultat attendu : la même ligne JSON répétée à chaque redémarrage :

```
api-1 | {"time":"...","level":"fatal","message":"Démarrage impossible : connect ECOM
```

Selon votre installation, le texte après « Démarrage impossible : » peut être **vide** : sur certaines machines, localhost correspond à deux adresses (127.0.0.1 et ::1), et Node.js renvoie alors une erreur groupée sans message. Le champ hint reste le même.

Ouvrez ~/taskflow/api/src/config.js : quelle est la valeur **par défaut** de databaseUrl ?

Sans reproduction : [incident/logs/api.log](#).

Pourquoi ? 127.0.0.1:5432, c'est localhost : la valeur par défaut du code, prévue pour le développement hors Docker. Dans le conteneur, la base n'est pas sur localhost mais sur db. Hypothèse : **DATABASE_URL n'arrive pas jusqu'au conteneur.**

Étape 5.5 : la cause racine (3 min)

Vérifiez l'hypothèse : quelles variables le conteneur a-t-il reçues, et que contient le .env ?

```
docker inspect taskflow-prod-api-1 --format '{{range .Config.Env}}{{println .}}{{end}}' |  
grep -n DATABASE .env
```

Comparez avec ~/taskflow/.env.example. Trouvez la différence, et formulez la cause en une phrase.

Sans reproduction : [incident/logs/env-api.txt](#).

Si ça ne marche pas - No such object: taskflow-prod-api-1 : le nom exact est donné par docker compose -f compose.prod.yaml ps (colonne NAME).

Pourquoi ? Une hypothèse à la fois, une vérification à la fois : on ne corrige rien tant que la cause n'est pas **prouvée**.

Étape 5.6 : corriger et vérifier (3 min)

Corrigez le nom de la variable dans .env avec nano .env, puis recréez l'API avec la version actuellement déployée :

```
IMAGE_TAG=$(cat .deployed_tag) docker compose -f compose.prod.yaml up -d --force-recreate
```

(Équivalent : ./deploy.sh "\$(cat .deployed_tag)".) Attendez une vingtaine de secondes, puis :

```
docker compose -f compose.prod.yaml ps  
curl -sk --resolve taskflow.localhost:443:127.0.0.1 https://taskflow.localhost/health  
date +%H:%M
```

Résultat attendu : les quatre services sont healthy, /health répond {"status": "ok", "db": "up", ...}, et la liste des tâches s'affiche de nouveau dans le navigateur. Notez l'heure : c'est la fin de l'incident.

Le script a gardé une copie de la configuration d'origine, qui contient vos secrets : supprimez-la.

```
rm .env.bak
```

Si ça ne marche pas - api redémarre toujours : relisez la ligne dans .env (nom **exact**, pas d'espace autour du =), puis relancez la commande `up -d --force-recreate api`. - Demandez au formateur : il peut tout remettre en état avec l'option `-- restaurer du script`.

Étape 5.7 : rédiger le post-mortem (10 min)

Dans votre dépôt :

```
cd ~/taskflow
git switch main && git pull
git switch -c docs/incident
code docs/INCIDENTS.md
```

Copiez ce modèle dans docs/INCIDENTS.md et remplissez-le avec **vos** heures et **vos** observations. N'ouvrez le docs/INCIDENTS.md de la solution (qui contient un exemple rédigé) qu'**après** avoir écrit le vôtre.

Gestion des incidents

Quand quelque chose casse en production, on répare d'abord, puis on comprend et on écrit le post-mortem. Un post-mortem est sans blâme : on parle de processus et de systèmes, pas de personnes.

AAAA-MM-JJ - Titre court de l'incident

```
**Gravité** : faible / moyenne / élevée
**Durée** : début hh:mm - fin hh:mm (durée totale)
**Auteur(s)** : ...
```

Résumé

Deux ou trois phrases : ce qui s'est passé et quel impact pour les utilisateurs.

Chronologie

- hh:mm - Déploiement / changement à l'origine de l'incident
- hh:mm - Premier signal (alerte, message d'un utilisateur...)
- hh:mm - Diagnostic : ce qu'on a regardé, ce qu'on a trouvé
- hh:mm - Correction appliquée
- hh:mm - Retour à la normale confirmé

Cause racine

Les « 5 pourquoi » : pourquoi le site renvoyait-il 502 ? Pourquoi l'API ne démarrait-elle pas ? Continuez jusqu'à une cause qu'on peut corriger.

Ce qui a bien fonctionné

```
- ...
```

Ce qui a moins bien fonctionné

```

- ...

### Actions
| Action | Responsable | Échéance | Statut |
| --- | --- | --- | --- |
| ... | ... | ... | à faire |

```

Attendu dans votre post-mortem :

- une chronologie avec **les commandes** que vous avez lancées et ce qu'elles vous ont appris ;
- une cause racine qui va **au-delà** de « faute de frappe » (pourquoi une faute de frappe a-t-elle pu casser la production sans que personne ne soit prévenu ?) ;
- **au moins deux actions** concrètes et vérifiables (pas « faire plus attention »).

Puis :

```

git add docs/INCIDENTS.md
git commit -m "docs: ajoute le post-mortem de l'incident 502"
git push -u origin docs/incident

```

Ouvrez la PR et demandez une revue à votre binôme.

Pourquoi ? « Pierre a mal tapé la variable » : c'est un blâme, et rien ne change. « Le .env de production est modifié à la main sans vérification » : c'est un processus, et on peut le corriger (comparer automatiquement le .env avec .env.example, faire échouer l'API au démarrage si une variable obligatoire manque, être alerté).

Je valide avec le formateur

- ☐ J'ai suivi la méthode dans l'ordre : symptôme → logs du proxy (et du front) → compose ps → logs de l'API → cause.
- ☐ Je sais expliquer pourquoi l'API cherchait la base sur 127.0.0.1.
- ☐ La production est réparée : quatre services healthy, /health à ok, .env.bak supprimé.
- ☐ Mon docs/INCIDENTS.md contient une chronologie horodatée, les « 5 pourquoi » et au moins deux actions concrètes.
- ☐ La PR docs/incident est ouverte et relue.

Questions de réflexion

1. Le pipeline a affiché un job rouge, mais ce sont les utilisateurs qui ont prévenu. Que faudrait-il mettre en place pour être alerté **avant** eux ? (Pensez à la démo Uptime Kuma.)
2. Pourquoi une valeur par défaut pratique en développement (localhost) est-elle dangereuse en production ? Que proposeriez-vous de changer dans api/src/config.js ?
3. Le rollback de l'étape 4 (redéployer le tag précédent) aurait-il réparé cet incident ? Pourquoi ?
4. Comment deploy.sh pourrait-il vérifier, avant de lancer quoi que ce soit, que .env contient toutes les variables de .env.example ?

Cet incident est un excellent exemple à raconter lors de la mise en situation : symptôme, méthode, cause, actions.