Gestion des erreurs dans FastAPI

Sommaire

Pourquoi gérer les erreurs ?

Une API ne doit pas seulement fonctionner quand tout va bien. Elle doit aussi savoir :

- Répondre clairement en cas de mauvaise requête (client).
- Éviter de renvoyer des erreurs techniques (tracebacks) en production.
- Renvoyer des erreurs compréhensibles pour les développeurs clients.

Les codes d'erreur HTTP

FastAPI repose sur les **codes HTTP** pour signaler le succès ou l'échec d'une requête.

Principaux codes :

- `200 OK` → Tout s'est bien passé.
- `400 Bad Request` → Mauvaise requête (souvent données invalides).
- `401 Unauthorized` / `403 Forbidden` → Accès interdit.
- `404 Not Found` → Ressource inexistante.
- `422 Unprocessable Entity` → Données mal formées (validation automatique).

> Plus de détails sur le cours chapitre 1 partie 4 de ce cursus.

Lever une erreur avec `HTTPException`

La classe `HTTPException` est utilisée pour retourner une erreur HTTP proprement.

```
from fastapi import FastAPI, HTTPException

app = FastAPI()

items = {"pen": "A blue pen"}

@app.get("/items/{item_id}")
async def get_item(item_id: str):
    if item_id not in items:
        raise HTTPException(status_code=404, detail="Item not found")
    return {"item": items[item_id]}
```

Si l'utilisateur demande un `item_id` inexistant, FastAPI renverra automatiquement :

```
{
  "detail": "Item not found"
}
```

Personnaliser le contenu de l'erreur

Tu peux envoyer plus d'informations dans le champ `detail`, comme un dictionnaire ou une liste.

```
raise HTTPException(
    status_code=403,
    detail={"error": "forbidden", "reason": "Not enough privileges"}
)
```

FastAPI se charge de convertir le `detail` en JSON.

Ajouter des en-têtes personnalisés

Tu peux également ajouter des en-têtes HTTP si besoin :

```
raise HTTPException(
    status_code=404,
    detail="Item not found",
    headers={"X-Error": "ItemMissing"}
)
```

Cela peut être utile pour transmettre des métadonnées à un client frontend ou un service externe.

Créer ses propres exceptions

Tu peux définir des exceptions personnalisées pour des erreurs métier.

```
class AccessDeniedException(Exception):
    def __init__(self, user: str):
        self.user = user
```

Puis tu peux créer un gestionnaire global :

```
from fastapi import Request
from fastapi.responses import JSONResponse

@app.exception_handler(AccessDeniedException)
async def access_denied_handler(request: Request, exc: AccessDeniedException):
    return JSONResponse(
        status_code=403,
        content={"message": f"User {exc.user} is not allowed to access this
resource."}
    )
```

Et l'utiliser :

```
@app.get("/secure")
def secure_endpoint(user: str):
    if user != "admin":
        raise AccessDeniedException(user)
    return {"message": "Welcome, admin"}
```

Surcharge des gestionnaires d'erreurs FastAPI

FastAPI utilise déjà des gestionnaires d'erreurs internes, notamment pour :

- `HTTPException`
- `RequestValidationError` (erreur dans les données d'entrée)

Tu peux les surcharger pour unifier les messages d'erreurs.

```python
from fastapi.exceptions import RequestValidationError
from starlette.exceptions import HTTPException as StarletteHTTPException
from fastapi.responses import PlainTextResponse

@app.exception_handler(StarletteHTTPException)
async def custom_http_exception_handler(request, exc):
    return PlainTextResponse(str(exc.detail), status_code=exc.status_code)

@app.exception_handler(RequestValidationError)
async def validation_exception_handler(request, exc):
    return PlainTextResponse("Mauvaise donnée envoyée", status_code=400)
```

Résumé

- `raise HTTPException(...)` permet de signaler une erreur propre.
- Tu peux personnaliser le message (`detail`) et les en-têtes (`headers`).
- Pour les cas spécifiques, tu peux créer tes propres exceptions.
- FastAPI permet de surcharger les gestionnaires globaux pour unifier ou simplifier les retours d'erreur.