1. Configurer le titre, la description et la version de l’instance FastAPI

Lorsque vous instanciez votre application, vous pouvez renseigner trois paramètres clés qui apparaîtront automatiquement dans la documentation Swagger (OpenAPI) :

```
from fastapi import FastAPI

app = FastAPI(
    title="Ma Super API",
    description="""
Ce service gère les ressources **Items** et **Users**.

- CRUD complet sur les items
- Authentification OAuth2
""",
    version="1.2.0",
)
```

> *Ces métadonnées se retrouvent dans l’UI Swagger à l’adresse `/docs` et dans Redoc à `/redoc`.*

---

2. Donner un name explicite à chaque route

Par défaut, FastAPI génère un identifiant pour chaque route à partir du nom de la fonction. Pour plus de clarté (et en cas de refactoring), vous pouvez fixer un nom unique :

```
@app.get("/items/{item_id}", name="items:get-one")
def get_item(item_id: int):
    ...
```

Cette valeur `name` permet, par exemple, de :

- Générer des URL via `app.url_path_for("items:get-one", item_id=42)`
- Disposer de noms cohérents dans les tests ou logs

---

3. Documenter vos méthodes et modèles avec des commentaires

3.1 Docstrings sur les endpoints

FastAPI extrait automatiquement la docstring de vos fonctions pour l’afficher dans la doc OpenAPI.

```
@app.post("/items", response_model=Item, name="items:create")
def create_item(item: ItemCreate):
    """
    Crée un nouvel item.
```

```
    - **name**: nom unique de l'item
    - **description**: description détaillée
    """
    ...
```

3.2 Commentaires dans les modèles Pydantic

Pydantic utilise les attributs `Field(...)` pour ajouter des descriptions et exemples :

```
from pydantic import BaseModel, Field

class Item(BaseModel):
    id: int = Field(..., description="Identifiant unique généré par la base")
    name: str = Field(..., max_length=100, example="Lampe de bureau")
    price: float = Field(..., gt=0, description="Prix en euros")
```

> *Les descriptions et exemples apparaîtront dans la section **Schemas** de Swagger.*

---

4. Organiser avec les tags

Les tags permettent de regrouper vos routes par thème ou ressource :

```
@app.get("/users/{user_id}", response_model=User, tags=["Users"], name="users:get-
one")
def read_user(user_id: int):
    """Récupère un utilisateur par son ID."""
    ...

@app.get("/items/{item_id}", response_model=Item, tags=["Items"], name="items:get-
one")
def read_item(item_id: int):
    """Récupère un item par son ID."""
    ...
```

Dans Swagger, vous aurez deux sections distinctes **Users** et **Items**.

---

5. Lever des HTTPException

Pour renvoyer automatiquement une réponse d'erreur standardisée :

```
from fastapi import HTTPException, status

@app.get("/items/{item_id}", response_model=Item, tags=["Items"])
def get_item(item_id: int):
    item = fake_db.get(item_id)
    if not item:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail=f"Item {item_id} non trouvé",
        )
    return item
```

> *FastAPI génère la réponse JSON :*

```
{
  "detail": "Item 42 non trouvé"
}
```

---

6. Créer un exception handler personnalisé

Vous pouvez intercepter toute exception ou une classe d'exceptions :

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

class CustomError(Exception):
    def __init__(self, name: str):
        self.name = name

@app.exception_handler(CustomError)
async def custom_error_handler(request: Request, exc: CustomError):
    return JSONResponse(
        status_code=418,
        content={"message": f"Oops! {exc.name} a déclenché une CustomError."},
    )
```

> *Ici, si une route lève `CustomError("toto")`, l'utilisateur recevra un statut HTTP **418** (I'm a teapot) avec votre message.*

---

7. Détailler les réponses possibles (`responses`)

Pour documenter formellement les différents codes de réponse et leurs schémas :

```python
from fastapi import status

@app.put(
    "/items/{item_id}",
    response_model=Item,
    responses={
        200: {"description": "Mise à jour réussie", "model": Item},
        404: {"description": "Item non trouvé", "model": ErrorResponse},
        422: {"description": "Erreur de validation des données"},
    },
    tags=["Items"],
)
def update_item(item_id: int, item: ItemUpdate):
    """
    Met à jour un item.
    """
    if item_id not in fake_db:
        raise HTTPException(status_code=404, detail="Item non trouvé")
    updated = fake_db[item_id].copy(update=item.dict())
```

```
    fake_db[item_id] = updated
    return updated
```

Ici, **ErrorResponse** peut être :

```
class ErrorResponse(BaseModel):
    code: int
    message: str
```

Résumé

- **Metadata** : `title`, `description`, `version` → visibles dans `/docs` et `/redoc`.
- **Routing** : `name` pour nommer vos endpoints.
- **Docstrings & Field** : documentation automatique des méthodes et modèles.
- **Tags** : organisation thématique de la doc.
- **HTTPException** : levée d’erreurs standard.
- **exception_handler** : gestion personnalisée des erreurs.
- **responses** : déclaration explicite des codes et modèles de réponse.

Avec ces bonnes pratiques votre API sera à la fois plus claire à consommer et plus facile à maintenir !