Handling Errors

There are many situations in which you need to notify an error to a client that is using your API.

This client could be a browser with a frontend, a code from someone else, an IoT device, etc.

You could need to tell the client that:

- The client doesn't have enough privileges for that operation.
- The client doesn't have access to that resource.
- The item the client was trying to access doesn't exist.
- etc.

In these cases, you would normally return an **HTTP status code** in the range of **400** (from 400 to 499).

This is similar to the 200 HTTP status codes (from 200 to 299). Those "200" status codes mean that somehow there was a "success" in the request.

The status codes in the 400 range mean that there was an error from the client.

Remember all those **"404 Not Found"** errors (and jokes)?

Use `HTTPException`

To return HTTP responses with errors to the client you use `HTTPException`.

Import `HTTPException`

Python 3.8+

```
from fastapi import FastAPI, HTTPException

app = FastAPI()

items = {"foo": "The Foo Wrestlers"}


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

Raise an `HTTPException` in your code

`HTTPException` is a normal Python exception with additional data relevant for APIs.

Because it's a Python exception, you don't `return` it, you `raise` it.

This also means that if you are inside a utility function that you are calling inside of your *path operation function*, and you raise the `HTTPException` from inside of that utility function, it won't run the rest of the code in the *path operation function*, it will terminate that request right away and send the HTTP error from the `HTTPException` to the client.

The benefit of raising an exception over `return`ing a value will be more evident in the section about Dependencies and Security.

In this example, when the client requests an item by an ID that doesn't exist, raise an exception with a status code of `404`:

Python 3.8+

```
from fastapi import FastAPI, HTTPException

app = FastAPI()

items = {"foo": "The Foo Wrestlers"}


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

The resulting response

If the client requests `http://example.com/items/foo` (an `item_id` `"foo"`), that client will receive an HTTP status code of 200, and a JSON response of:

```
{
  "item": "The Foo Wrestlers"
}
```

But if the client requests `http://example.com/items/bar` (a non-existent `item_id` `"bar"`), that client will receive an HTTP status code of 404 (the "not found" error), and a JSON response of:

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

```
}
```

Tip

When raising an `HTTPException`, you can pass any value that can be converted to JSON as the parameter `detail`, not only `str`.

You could pass a `dict`, a `list`, etc.

They are handled automatically by **FastAPI** and converted to JSON.

Add custom headers

There are some situations in where it's useful to be able to add custom headers to the HTTP error. For example, for some types of security.

You probably won't need to use it directly in your code.

But in case you needed it for an advanced scenario, you can add custom headers:

Python 3.8+

```python
from fastapi import FastAPI, HTTPException

app = FastAPI()

items = {"foo": "The Foo Wrestlers"}


@app.get("/items-header/{item_id}")
async def read_item_header(item_id: str):
    if item_id not in items:
        raise HTTPException(
            status_code=404,
            detail="Item not found",
            headers={"X-Error": "There goes my error"},
        )
    return {"item": items[item_id]}
```

Install custom exception handlers

You can add custom exception handlers with the same exception utilities from Starlette.

Let's say you have a custom exception `UnicornException` that you (or a library you use) might `raise`.

And you want to handle this exception globally with FastAPI.

You could add a custom exception handler with `@app.exception_handler()`:

Python 3.8+

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


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


app = FastAPI()


@app.exception_handler(UnicornException)
async def unicorn_exception_handler(request: Request, exc: UnicornException):
    return JSONResponse(
        status_code=418,
        content={"message": f"Oops! {exc.name} did something. There goes a rainbow..."},
    )


@app.get("/unicorns/{name}")
async def read_unicorn(name: str):
    if name == "yolo":
        raise UnicornException(name=name)
    return {"unicorn_name": name}
```

Here, if you request `/unicorns/yolo`, the *path operation* will `raise a UnicornException`.

But it will be handled by the `unicorn_exception_handler`.

So, you will receive a clean error, with an HTTP status code of `418` and a JSON content of:

```
{"message": "Oops! yolo did something. There goes a rainbow..."}
```

Technical Details

You could also use `from starlette.requests import Request` and `from starlette.responses import JSONResponse`.

FastAPI provides the same `starlette.responses` as `fastapi.responses` just as a convenience for you, the developer. But most of the available responses come directly from Starlette. The same with `Request`.

Override the default exception handlers

FastAPI has some default exception handlers.

These handlers are in charge of returning the default JSON responses when you `raise` an `HTTPException` and when the request has invalid data.

You can override these exception handlers with your own.

Override request validation exceptions

When a request contains invalid data, **FastAPI** internally raises a `RequestValidationError`.

And it also includes a default exception handler for it.

To override it, import the `RequestValidationError` and use it with `@app.exception_handler(RequestValidationError)` to decorate the exception handler.

The exception handler will receive a `Request` and the exception.

Python 3.8+

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

app = FastAPI()


@app.exception_handler(StarletteHTTPException)
async def 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(str(exc), status_code=400)


@app.get("/items/{item_id}")
async def read_item(item_id: int):
    if item_id == 3:
        raise HTTPException(status_code=418, detail="Nope! I don't like 3.")
    return {"item_id": item_id}
```

Now, if you go to `/items/foo`, instead of getting the default JSON error with:

```
{
    "detail": [
        {
            "loc": [
                "path",
                "item_id"
            ],
            "msg": "value is not a valid integer",
            "type": "type_error.integer"
        }
    ]
}
```

you will get a text version, with:

```
1 validation error
path -> item_id
  value is not a valid integer (type=type_error.integer)
```

`RequestValidationError` vs `ValidationError`

Warning

These are technical details that you might skip if it's not important for you now.

`RequestValidationError` is a sub-class of Pydantic's `ValidationError`.

FastAPI uses it so that, if you use a Pydantic model in `response_model`, and your data has an error, you will see the error in your log.

But the client/user will not see it. Instead, the client will receive an "Internal Server Error" with an HTTP status code `500`.

It should be this way because if you have a Pydantic `ValidationError` in your *response* or anywhere in your code (not in the client's *request*), it's actually a bug in your code.

And while you fix it, your clients/users shouldn't have access to internal information about the error, as that could expose a security vulnerability.

Override the `HTTPException` error handler

The same way, you can override the `HTTPException` handler.

For example, you could want to return a plain text response instead of JSON for these errors:

Python 3.8+

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

app = FastAPI()


@app.exception_handler(StarletteHTTPException)
async def 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(str(exc), status_code=400)


@app.get("/items/{item_id}")
async def read_item(item_id: int):
    if item_id == 3:
        raise HTTPException(status_code=418, detail="Nope! I don't like 3.")
    return {"item_id": item_id}
```

Technical Details

You could also use `from starlette.responses import PlainTextResponse` .

FastAPI provides the same `starlette.responses` as `fastapi.responses` just as a convenience for you, the developer. But most of the available responses come directly from Starlette.

Use the `RequestValidationError` body

The `RequestValidationError` contains the `body` it received with invalid data.

You could use it while developing your app to log the body and debug it, return it to the user, etc.

Python 3.8+

```
from fastapi import FastAPI, Request, status
from fastapi.encoders import jsonable_encoder
from fastapi.exceptions import RequestValidationError
from fastapi.responses import JSONResponse
from pydantic import BaseModel

app = FastAPI()


@app.exception_handler(RequestValidationError)
async def validation_exception_handler(request: Request, exc: RequestValidationError):
    return JSONResponse(
        status_code=status.HTTP_422_UNPROCESSABLE_ENTITY,
        content=jsonable_encoder({"detail": exc.errors(), "body": exc.body}),
    )


class Item(BaseModel):
    title: str
    size: int


@app.post("/items/")
async def create_item(item: Item):
    return item
```

Now try sending an invalid item like:

```
{
  "title": "towel",
  "size": "XL"
}
```

You will receive a response telling you that the data is invalid containing the received body:

```
{
  "detail": [
    {
      "loc": [
        "body",
        "size"
```

```
      ],
      "msg": "value is not a valid integer",
      "type": "type_error.integer"
    }
  ],
  "body": {
    "title": "towel",
    "size": "XL"
  }
}
```

FastAPI's `HTTPException` vs Starlette's `HTTPException`

FastAPI has its own `HTTPException`.

And **FastAPI**'s `HTTPException` error class inherits from Starlette's `HTTPException` error class.

The only difference is that **FastAPI**'s `HTTPException` accepts any JSON-able data for the `detail` field, while Starlette's `HTTPException` only accepts strings for it.

So, you can keep raising **FastAPI**'s `HTTPException` as normally in your code.

But when you register an exception handler, you should register it for Starlette's `HTTPException`.

This way, if any part of Starlette's internal code, or a Starlette extension or plug-in, raises a Starlette `HTTPException`, your handler will be able to catch and handle it.

In this example, to be able to have both `HTTPException`s in the same code, Starlette's exceptions is renamed to `StarletteHTTPException`:

```
from starlette.exceptions import HTTPException as StarletteHTTPException
```

Reuse FastAPI's exception handlers

If you want to use the exception along with the same default exception handlers from **FastAPI**, you can import and reuse the default exception handlers from `fastapi.exception_handlers`:

Python 3.8+

```
from fastapi import FastAPI, HTTPException
from fastapi.exception_handlers import (
    http_exception_handler,
    request_validation_exception_handler,
)
from fastapi.exceptions import RequestValidationError
from starlette.exceptions import HTTPException as StarletteHTTPException

app = FastAPI()


@app.exception_handler(StarletteHTTPException)
async def custom_http_exception_handler(request, exc):
    print(f"OMG! An HTTP error!: {repr(exc)}")
    return await http_exception_handler(request, exc)


@app.exception_handler(RequestValidationError)
async def validation_exception_handler(request, exc):
    print(f"OMG! The client sent invalid data!: {exc}")
    return await request_validation_exception_handler(request, exc)


@app.get("/items/{item_id}")
async def read_item(item_id: int):
    if item_id == 3:
        raise HTTPException(status_code=418, detail="Nope! I don't like 3.")
    return {"item_id": item_id}
```

In this example you are just `print`ing the error with a very expressive message, but you get the idea. You can use the exception and then just reuse the default exception handlers.