⚡ FastAPI Other Concepts Overview

🚨 1. Error Handling in FastAPI

✅ Built-in Exception Handling

FastAPI provides built-in exceptions like `HTTPException`.

```
from fastapi import FastAPI, HTTPException

app = FastAPI()

@app.get("/item/{item_id}")
def read_item(item_id: int):
    if item_id == 0:
        raise HTTPException(status_code=404, detail="Item not found 🚫")
    return {"item_id": item_id}
```

✅ Custom Exception Handling

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

@app.exception_handler(RequestValidationError)
async def validation_exception_handler(request: Request, exc:
RequestValidationError):
    return JSONResponse(
        status_code=422,
        content={"message": "Validation failed 😵", "details": exc.errors()},
    )
```

🔐 Best Practices

- Use descriptive `detail` in errors.
- Handle common errors (e.g., 404, 400, 500) globally.
- Avoid leaking sensitive info in error messages.

📬 2. Custom Responses

✅ Using `JSONResponse`, `HTMLResponse`, `PlainTextResponse`

```python
from fastapi.responses import JSONResponse, HTMLResponse, PlainTextResponse

@app.get("/json")
def custom_json():
    return JSONResponse(content={"msg": "Hello JSON 👋"})

@app.get("/html")
def custom_html():
    return HTMLResponse(content="<h1>Hello HTML 👩‍💻</h1>")

@app.get("/text")
def custom_text():
    return PlainTextResponse(content="Hello Text 📝")
```

🔐 Best Practices

- Use appropriate content types.
- Set custom status codes when needed (`status_code=201`).
- Always document your custom responses using `@app.get(..., responses={})`.

🗒️ 3. Headers in FastAPI

✅ Accessing Request Headers

```python
from fastapi import Header

@app.get("/headers/")
def read_headers(user_agent: str = Header(...)):
    return {"User-Agent": user_agent}
```

✅ Setting Response Headers

```python
from fastapi.responses import Response

@app.get("/set-header/")
def custom_header(response: Response):
    response.headers["X-Custom-Header"] = "FastAPI ❤️"
    return {"message": "Custom header sent!"}
```

🔐 Best Practices

- Avoid sensitive data in headers.

- Use standard naming conventions (`X-Custom-Header`, etc).

🍪 4. Cookies in FastAPI

✅ Setting Cookies

```
@app.get("/set-cookie/")
def set_cookie(response: Response):
    response.set_cookie(key="session_id", value="abc123", httponly=True)
    return {"message": "🍪 Cookie set!"}
```

✅ Reading Cookies

```
from fastapi import Cookie

@app.get("/get-cookie/")
def get_cookie(session_id: str = Cookie(None)):
    return {"session_id": session_id}
```

🔐 Best Practices

- Use `httponly=True` to prevent JavaScript access.
- Set `secure=True` for HTTPS.
- Prefer JWTs or sessions over plain cookies.

📝 5. Form Data Handling

✅ Receiving Form Data

```
from fastapi import Form

@app.post("/login/")
def login(username: str = Form(...), password: str = Form(...)):
    return {"username": username, "message": "Login form submitted ✅"}
```

🔐 Best Practices

- Always use `Form` for `x-www-form-urlencoded` POST data.
- Use HTTPS to protect credentials.
- Consider CSRF protection for forms in production apps.

🔄 6. CORS (Cross-Origin Resource Sharing)

✅ Enable CORS Middleware

```
from fastapi.middleware.cors import CORSMiddleware

app = FastAPI()

origins = [
    "http://localhost:3000",  # React frontend
    "https://myapp.com",      # Production domain
]

app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,          # or ["*"] for all
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

🔐 Best Practices

- In production, restrict `allow_origins` to known domains.
- Don't use `["*"]` for `allow_credentials=True` (violates spec).
- Allow only necessary headers and methods.

🔁 Flow Overview Diagram (Text-Based)

```
Frontend 🔄 ⟶ CORS Middleware 🌍
          ⬇
     FastAPI Endpoint 🧠
          ⬇
   - Read Headers/Cookies
   - Parse Form Data
   - Handle Errors (Custom/Built-in)
   - Generate Response
          ⬇
    Response with Headers/Cookies ⬅️ BACK
```

📚 Summary Table

Feature	Module	Key Functionality
Error Handling	`HTTPException`, `exception_handler()`	Global & custom error responses
Custom Responses	`JSONResponse`, `HTMLResponse`	Control response format and headers
Headers	`Header`, `Response.headers`	Read and write HTTP headers
Cookies	`Cookie`, `response.set_cookie()`	Store sessions or metadata in cookies
Form Data	`Form`	Parse form submissions
CORS	`CORSMiddleware`	Secure frontend-backend communication

✅ Final Best Practices Checklist

- ☑️ 🔐 Handle all exceptions gracefully.
- ☑️ 💬 Provide meaningful error messages.
- ☑️ 📦 Use correct response types.
- ☑️ 🧁 Handle cookies with security flags (`HttpOnly`, `Secure`).
- ☑️ 🌫️ Use CORS to secure frontend-backend interaction.
- ☑️ 📄 Validate all form and query data.

🌐 HTTP Status Codes Cheat Sheet with FastAPI 💡

🔵 1xx — **INFORMATIONAL RESPONSES**

> ✅ Request received, continuing process.

Code	Meaning	Description
100	Continue	Client should continue the request.
101	Switching Protocols	Server agrees to change protocols.
102	Processing (WebDAV)	Server has accepted request but not completed it.

🧠 **FastAPI Tip**: Rarely used directly.

🟢 2xx — **SUCCESSFUL RESPONSES**

> 🎉 The request was successfully received, understood, and accepted.

Code	Meaning	Description
200	OK	Standard success for GET/PUT/DELETE.

Code	Meaning	Description
201	Created	Used after successful POST (e.g., resource created).
202	Accepted	Request accepted but not completed yet.
204	No Content	Success, but no response body (e.g., DELETE).

📦 **FastAPI Usage Example**:

```
from fastapi import status

@app.post("/user", status_code=status.HTTP_201_CREATED)
def create_user():
    return {"message": "User created ✅"}
```

🟡 3xx — REDIRECTION MESSAGES

> 🔁 The client must take additional action.

Code	Meaning	Description
301	Moved Permanently	Resource has a new URL. Update your links!
302	Found (Temporary)	Temporarily moved to another URI.
304	Not Modified	Cached version can be used.
307	Temporary Redirect	Like 302 but preserves method (POST/GET).
308	Permanent Redirect	Like 301 but with method preservation.

📦 **FastAPI Example**:

```
from fastapi.responses import RedirectResponse

@app.get("/old-link")
def old_link():
    return RedirectResponse(url="/new-link", status_code=307)
```

🔴 4xx — CLIENT ERROR RESPONSES

> 🚫 The request contains bad syntax or cannot be fulfilled.

Code	Meaning	Description
400	Bad Request	Client error (validation, format, etc).
401	Unauthorized	Authentication required.
403	Forbidden	Authenticated but not allowed.
404	Not Found	Resource doesn't exist.
405	Method Not Allowed	Method (GET/POST/etc) not supported.
409	Conflict	Resource conflict (duplicate user, etc).
422	Unprocessable Entity	Validation failed (e.g., FastAPI Pydantic errors).
429	Too Many Requests	Rate limit hit.

📦 **FastAPI Example (422 Validation)**:

```
from fastapi import HTTPException

@app.post("/login")
def login(username: str):
    if username == "":
        raise HTTPException(status_code=400, detail="Username is required 🚫")
```

🔥 5xx — SERVER ERROR RESPONSES

> 💥 The server failed to fulfill a valid request.

Code	Meaning	Description
500	Internal Server Error	Generic server error.
501	Not Implemented	Server doesn't support the requested feature.
502	Bad Gateway	Invalid response from upstream server.
503	Service Unavailable	Server is down or overloaded.
504	Gateway Timeout	Upstream server timed out.

📦 **FastAPI Example (500 Error)**:

```
@app.get("/crash")
def cause_crash():
    raise Exception("💥 Boom! Something broke.")
```

🚦 Frequently Used HTTP Status Codes (Dev Cheatsheet)

Type	Codes	Description
✅ Success	`200`, `201`, `204`	Standard API success responses.
❌ Client	`400`, `401`, `403`, `404`, `422`	Validation and auth errors.
🔁 Redirect	`301`, `302`, `307`, `308`	Redirection flows.
🚨 Server	`500`, `503`	Server crash or overload.

🧪 FastAPI Usage Summary

✅ You can use **status codes** directly via:

```
from fastapi import status

@app.post("/resource", status_code=status.HTTP_201_CREATED)
def create():
    return {"msg": "Created"}
```

Or set manually in `JSONResponse`:

```
from fastapi.responses import JSONResponse

return JSONResponse(
    status_code=404,
    content={"success": False, "message": "Not Found"}
)
```

✅ Best Practices Summary

- 🎯 Always return proper status codes based on result.
- 🔒 Use 401 for auth and 403 for access control.
- 🧾 Use 422 for form/Pydantic validation errors.
- 💬 Combine with `ApiResponse` or `ApiException` classes for consistent shape.
- 🚨 Never expose stack traces or sensitive data in error responses in production.

🚨 1. Error Handling in FastAPI

🔹 Custom Exception Classes

```python
# 👾 Define a custom exception
class StoryException(Exception):
    def __init__(self, name: str):
        self.name = name
```

🔹 Add Exception Handler

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

@app.exception_handler(StoryException)
async def story_exception_handler(request: Request, exc: StoryException):
    return JSONResponse(
        status_code=418,  # 🔥 Custom status code
        content={"detail": f"Oops! {exc.name} caused an issue 💀"}
    )
```

✅ Usage

```python
@app.get("/story/")
def get_story(flag: bool = False):
    if not flag:
        raise StoryException(name="Missing Story Flag 🏳️")
    return {"message": "Here's your story! 📖"}
```

📦 2. Custom Response

🔹 PlainTextResponse, HTMLResponse, FileResponse

```python
from fastapi.responses import HTMLResponse, PlainTextResponse

@app.get("/html", response_class=HTMLResponse)
def get_html():
    return "<h1>Hello 🌍, this is raw HTML!</h1>"

@app.get("/text", response_class=PlainTextResponse)
def get_text():
    return "This is plain text ✉️"
```

🔹 FileResponse (Download files)

```python
from fastapi.responses import FileResponse

@app.get("/download")
def download_file():
    return FileResponse("my_resume.pdf", media_type='application/pdf',
filename="Resume.pdf")
```

🗒️ 3. Headers

- Access Headers from Request

```python
from fastapi import Request

@app.get("/headers/")
def read_headers(request: Request):
    headers = request.headers
    user_agent = headers.get("user-agent")
    return {"user_agent": user_agent}
```

- Send Custom Headers in Response

```python
from fastapi.responses import JSONResponse

@app.get("/custom-header")
def custom_header():
    content = {"message": "Hello with headers!"}
    headers = {"X-Powered-By": "FastAPI ⚡"}
    return JSONResponse(content=content, headers=headers)
```

🍪 4. Cookies

- Set a Cookie 🍪

```python
from fastapi import Response

@app.get("/set-cookie")
def set_cookie(response: Response):
    response.set_cookie(key="token", value="abc123")
    return {"message": "Cookie set ✅"}
```

🔹 Read a Cookie

```
from fastapi import Cookie

@app.get("/get-cookie")
def get_cookie(token: str = Cookie(default=None)):
    return {"your_token": token}
```

📝 5. Form Data

Used in **HTML forms** and `application/x-www-form-urlencoded` requests.

🔹 Form Parsing with `Form`

```
from fastapi import Form

@app.post("/login")
def login(username: str = Form(...), password: str = Form(...)):
    return {"username": username, "message": "Logged in ✅"}
```

📌 `Form(...)` means the field is **required**.

🔄 6. CORS (Cross-Origin Resource Sharing)

🔐 What is CORS?

CORS controls who can access your API. It's especially important when frontend (e.g., React) is hosted on a different domain than backend (FastAPI).

🔹 Enable CORS Middleware

```
from fastapi.middleware.cors import CORSMiddleware

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],  # Or restrict to ["https://your-frontend.com"]
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

🎓 Summary Table

Topic	Use Case	Key Function/Class
Error Handling	Custom errors & responses	`@app.exception_handler`
Custom Response	Return HTML, Text, Files	`HTMLResponse`, `FileResponse`
Headers	Metadata from client or sent by server	`request.headers`, `JSONResponse(headers=...)`
Cookies	Session or auth tokens	`response.set_cookie()`, `Cookie()`
Form Data	HTML form submissions	`Form()`
CORS	Allow cross-origin frontend requests	`CORSMiddleware`

🚨 FastAPI — Exceptions & Error Handling

🔹 What Are Exceptions?

> Exceptions in FastAPI allow us to **notify the client** that **something went wrong** in a clean and structured way.

📌 When you `raise` an exception, it **stops the execution** of your code at that point and returns a well-defined **error response** to the client.

🧱 Built-in Exception — `HTTPException`

FastAPI provides a **built-in exception**:

```
from fastapi import HTTPException
```

👮 Example: Raise a 404 Not Found Error

```
from fastapi import FastAPI, HTTPException

app = FastAPI()

fake_users_db = {"1": "Alice", "2": "Bob"}

@app.get("/user/{user_id}")
def get_user(user_id: str):
```

```
    user = fake_users_db.get(user_id)
    if not user:
        # 🚨 Raise an HTTP Exception
        raise HTTPException(
            status_code=404,  # Not Found ❌
            detail=f"User with id {user_id} not found"
        )
    return {"user": user}
```

🔍 Output If User Not Found:

```
{
  "detail": "User with id 6 not found"
}
```

🧠 Explanation:

- `status_code=404`: Tells the client this is a **"Not Found"** error.
- `detail`: Gives human-readable error info.
- ✅ FastAPI automatically structures this as a proper JSON error response.

🔢 Common HTTP Status Codes

Code	Name	Use Case
400	Bad Request	Invalid input, wrong format 🚫
401	Unauthorized	Missing or bad auth header 🔐
403	Forbidden	Valid credentials, but access denied 🚷
404	Not Found	Resource doesn't exist 🔎
409	Conflict	Duplicate resource or version clash ⚔️
422	Unprocessable Entity	Validation errors (FastAPI default) ❌
500	Internal Server Error	Server-side crash 💥

🎯 Best Practice: Raise Early

✅ Raising exceptions early prevents further code from executing. Example:

```
if not valid_data:
    raise HTTPException(status_code=400, detail="Invalid input data 🧨")
```

```
# ⛔ this line won't run if exception is raised
```

✍️ Custom Exception Example (Optional)

For more structured error handling:

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

@app.exception_handler(MyCustomError)
def my_custom_handler(request, exc: MyCustomError):
    return JSONResponse(status_code=418, content={"detail": f"Oops! {exc.name} is
invalid 🚫"})
```

📦 Summary

Feature	Description
`HTTPException`	FastAPI's built-in error-raising mechanism
`status_code`	HTTP status (e.g. 404 for Not Found)
`detail`	Message shown to the client
Stops execution	Yes — further code won't run after `raise`
Returns JSON response	Yes — structured automatically by FastAPI

🧠 Real-World Use Case

```
@app.get("/product/{product_id}")
def get_product(product_id: int):
    product = db.get_product_by_id(product_id)
    if not product:
        raise HTTPException(
            status_code=404,
            detail=f"Product with id {product_id} not found 😔"
        )
    return product
```

🚀 FastAPI — Deep Dive into Response Handling

📝 1. ✅ Standard Response

FastAPI allows returning any of the following as a valid response:

✅ Python Types → Automatically converted into JSON by FastAPI

- `dict`
- `list`
- `pydantic models`
- `ORM objects` (if serialized)
- Even `str`, `int`, etc.

```
@app.get("/greet")
def greet():
    return {"message": "Hello, FastAPI 👋"}
```

FastAPI ⏩ Automatically converts this dict into:

```
{
  "message": "Hello, FastAPI 👋"
}
```

🧊 2. 🔧 Customizing `Response`

Use `fastapi.responses.Response` or `HTMLResponse`, `PlainTextResponse`, etc., to **manually control** what gets returned.

Example: Return HTML Content

```
from fastapi.responses import Response

@app.get("/html")
def get_html():
    html = "<h1>🚀 Welcome to FastAPI</h1>"
    return Response(content=html, media_type="text/html")
```

Why use Custom Response?

✅ Full control over:

- `Headers`
- `Cookies`
- `Streaming`
- `Media Type` (HTML, XML, JSON, etc.)
- No auto data conversion
- Better docs + Swagger behavior

🎁 3. Types of Responses

Type	Class	Use case
`text/plain`	`PlainTextResponse`	Return plain text 📄
`text/html`	`HTMLResponse`	Render raw HTML content 🌐
`application/xml`	`Response(media_type="application/xml")`	For XML APIs 📄
`application/json`	Default	JSON APIs
Files	`FileResponse`	Serve static files 📂
Streaming	`StreamingResponse`	Big file/data streaming 💾

📬 4. Add Request Headers

You can **read headers** sent by the client using the `Header` class.

```
from fastapi import Header
from typing import Optional

@app.get("/")
def get_with_header(custom_header: Optional[str] = Header(None)):
    return {"Received-Header": custom_header}
```

🔁 Converts `custom-header` automatically from `custom_header`.

📥 List of Headers

```
from typing import List

@app.get("/")
def get_headers(custom_header: Optional[List[str]] = Header(None)):
    return {"custom_header": custom_header}
```

📤 5. Set Custom Response Headers

```
from fastapi import Response

@app.get("/users")
def get_all_users(response: Response):
    response.headers["x-custom-header"] = "abc123"
    return {"users": ["Alice", "Bob"]}
```

✅ Add metadata, tracking ID, token expiry info, etc.

🍪 6. Cookies

Cookies = tiny pieces of info stored on client browsers.

🔸 Set Cookie

```
@app.get("/set-cookie")
def set_cookie(response: Response):
    response.set_cookie(key="cookie", value="test-cookie-value 🍪")
    return {"message": "Cookie has been set"}
```

🔸 Get Cookie

```
from fastapi import Cookie

@app.get("/get-cookie")
def get_cookie(cookie: Optional[str] = Cookie(None)):
    return {"Your cookie value": cookie}
```

📝 7. Form Data (HTML)

Used in frontend `<form>` submissions.

```
<form method="post">
  <input name="username" />
  <input name="password" />
</form>
```

📥 Read Form Data

```
from fastapi import Form

@app.post("/login")
def login(username: str = Form(...), password: str = Form(...)):
    return {"username": username}
```

🔴 Media Type = `application/x-www-form-urlencoded` or `multipart/form-data` (for files)

🔀 8. CORS (Cross-Origin Resource Sharing)

💡 Allows your backend (e.g., `localhost:8000`) to talk to a frontend on another domain (e.g., `localhost:3000`)

Enable CORS

```
from fastapi.middleware.cors import CORSMiddleware

app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:3000"],  # frontend
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

🧠 Why?

Without this, browsers block cross-origin AJAX calls due to **same-origin policy**.

🧠 Summary Cheat Sheet

Feature	Class/Function	Purpose
Return JSON	default	JSON APIs
HTML response	`Response(..., "text/html")`	Web rendering
Custom headers	`Response.headers[...]`	Add metadata
Read headers	`Header(...)`	Accept client headers
Set cookies	`response.set_cookie()`	Store info on client
Read cookies	`Cookie(...)`	Access client-side cookies

Feature	Class/Function	Purpose
Form data	`Form(...)`	Read HTML form input
CORS	`CORSMiddleware`	Allow frontend-backend bridge