Error Handling

🎯 Goal

We'll implement these equivalents:

Express Component	FastAPI Equivalent
`ApiError.js`	✅ `api_exceptions.py` (custom exception class)
`ApiResponse.js`	✅ `api_response.py` (standard response wrapper)
`asyncHandler.js`	✅ FastAPI handles async natively, but we can use `try-except` decorators for reuse

📁 Folder Structure (Suggested)

```
app/
├── main.py
├── routes/
│   └── user.py
├── core/
│   ├── api_exceptions.py
│   ├── api_response.py
│   └── error_handler.py
```

📦 1. `api_exceptions.py` (🌋 Equivalent of ApiError.js)

```
# core/api_exceptions.py

class ApiException(Exception):
    def __init__(self, status_code=500, message="Something went wrong",
errors=None):
        self.status_code = status_code
        self.message = message
        self.success = False
        self.errors = errors or []
        super().__init__(self.message)
```

📦 2. `api_response.py` (📦 Equivalent of ApiResponse.js)

```python
# core/api_response.py

from fastapi.responses import JSONResponse

class ApiResponse:
    def __init__(self, data=None, message="Success", status_code=200):
        self.status_code = status_code
        self.message = message
        self.data = data
        self.success = status_code < 400

    def send(self):
        return JSONResponse(
            status_code=self.status_code,
            content={
                "success": self.success,
                "message": self.message,
                "data": self.data
            }
        )
```

🧤 3. Global Error Handler (🧯 Equivalent of Express `.use(errorHandler)`)

```python
# core/error_handler.py

from fastapi import Request
from fastapi.responses import JSONResponse
from core.api_exceptions import ApiException
from fastapi.exceptions import RequestValidationError
from starlette.status import HTTP_500_INTERNAL_SERVER_ERROR

def register_exception_handlers(app):

    @app.exception_handler(ApiException)
    async def api_exception_handler(request: Request, exc: ApiException):
        return JSONResponse(
            status_code=exc.status_code,
            content={
                "success": False,
                "message": exc.message,
                "errors": exc.errors,
                "data": None
            }
        )
```

```
    @app.exception_handler(RequestValidationError)
    async def validation_exception_handler(request: Request, exc:
RequestValidationError):
        return JSONResponse(
            status_code=422,
            content={
                "success": False,
                "message": "Validation Error",
                "errors": exc.errors(),
                "data": None
            }
        )

    @app.exception_handler(Exception)
    async def general_exception_handler(request: Request, exc: Exception):
        return JSONResponse(
            status_code=HTTP_500_INTERNAL_SERVER_ERROR,
            content={
                "success": False,
                "message": str(exc),
                "errors": [],
                "data": None
            }
        )
```

⚙️ 4. `main.py` – Tie It All Together

```
# main.py

from fastapi import FastAPI
from core.error_handler import register_exception_handlers
from routes import user

app = FastAPI()

# Register global exception handlers
register_exception_handlers(app)

# Include routes
app.include_router(user.router)
```

🧪 5. Sample Route with Custom Response & Error

```python
# routes/user.py

from fastapi import APIRouter
from core.api_response import ApiResponse
from core.api_exceptions import ApiException

router = APIRouter(
    prefix="/user",
    tags=["User"]
)

@router.get("/profile")
async def get_user():
    # Simulate a condition
    raise ApiException(status_code=404, message="User not found 🕵️‍♂️")

@router.get("/welcome")
async def welcome_user():
    response = ApiResponse(data={"name": "Darshan"}, message="Welcome 🎉")
    return response.send()
```

✅ Final API Response Examples

✅ Success Response

```json
{
  "success": true,
  "message": "Welcome 🎉",
  "data": {
    "name": "Darshan"
  }
}
```

❌ Error Response (Custom)

```json
{
  "success": false,
  "message": "User not found 🕵️‍♂️",
  "errors": [],
  "data": null
}
```

✅ Advantages of This Setup

🔁 Reusable 📦 Consistent Response Shape 🚨 Centralized Error Handling 🧪 Test Friendly & Scalable 💻 Developer-Friendly Debugging

---

async/await natively. Unlike Express.js where you need `asyncHandler()` to catch errors in async functions (since unhandled promise rejections can crash the app), FastAPI's internal engine (Starlette + ASGI) **already handles async errors properly**.

So, technically, you **don't need an `asyncHandler` like in Express**.

✅ But... What if you want **middleware-like async wrappers**?

You **can** implement a reusable `async_handler` decorator in FastAPI for:

- Logging errors 🌐
- Converting raw exceptions into your custom `ApiException` 🚨
- Centralizing error wrapping across multiple endpoints 📦

⚙️ Create `async_handler` Decorator (Optional)

```
# core/async_handler.py

from functools import wraps
from core.api_exceptions import ApiException

def async_handler(func):
    @wraps(func)
    async def wrapper(*args, **kwargs):
        try:
            return await func(*args, **kwargs)
        except ApiException as ae:
            raise ae  # Let FastAPI handle this via your global handler
        except Exception as e:
            # Convert unhandled errors into your custom ApiException
            raise ApiException(status_code=500, message=str(e))
    return wrapper
```

🧪 Use It in Routes (Optional)

```
# routes/user.py

from fastapi import APIRouter
```

```python
from core.api_response import ApiResponse
from core.api_exceptions import ApiException
from core.async_handler import async_handler

router = APIRouter(prefix="/user", tags=["User"])

@router.get("/profile")
@async_handler
async def get_user():
    # Simulating error
    raise ApiException(status_code=404, message="User not found 💥")

@router.get("/safe")
@async_handler
async def safe_route():
    # Simulating unknown error
    1 / 0  # This will raise ZeroDivisionError
```

🔄 Flow with `async_handler`

```
Request ⟶ Route Handler ⟶ async_handler Decorator
                              ⬇️
                    Try → Await function
                    Catch → Wrap unknown error in ApiException
                              ⬇️
                   Global error_handler catches ApiException
```

✅ Summary

Use Case	Needed in FastAPI?	How to Implement
Catch async route errors	❌ Handled natively	
Uniform error wrapping	✅ Optional decorator	
Centralized error format	✅ Via global handler	

🔥 Custom Exception Handling in FastAPI – Notes & Code

📘 Why Use Custom Exceptions?

- 🧠 Clean separation of error logic.
- 🧩 Encapsulates custom logic and metadata.
- 🧠 Easier to debug and maintain.
- 🚦 Enables specific error codes (e.g., `418 I'm a teapot` 🫖).

🧱 Structure of Custom Exception Handling

✅ Key Components:

1. **Custom Exception class** – must inherit from `Exception`.
2. **Exception Handler function** – decorated with `@app.exception_handler`.
3. **Registering the handler** – FastAPI will auto-map based on exception type.

🧪 Example: `StoryException` with Code

```python
# main.py or your router file

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

app = FastAPI()

# 🔹 1. Define a custom exception
class StoryException(Exception):
    def __init__(self, name: str):
        self.name = name

# 🔹 2. Provide an exception handler
@app.exception_handler(StoryException)
async def story_exception_handler(request: Request, exc: StoryException):
    return JSONResponse(
        status_code=418,  # 🔥 I'm a teapot (fun status code for demo)
        content={
            "success": False,
            "message": f"Oops! {exc.name} is not allowed in our story. 📚",
            "data": None
        }
    )

# 🔹 3. Route that triggers the custom exception
@app.get("/story/{name}")
async def read_story(name: str):
    if name.lower() == "badwolf":
        raise StoryException(name=name)
    return {"message": f"{name} is a great story character! 🎭"}
```

💬 Output Examples

✅ Successful Call:

`GET /story/harry`

```
{
  "message": "harry is a great story character! 🎭"
}
```

❌ Custom Exception Raised:

`GET /story/badwolf`

```
{
  "success": false,
  "message": "Oops! badwolf is not allowed in our story. 🚫",
  "data": null
}
```

✅ Best Practices for Custom Exceptions

Practice	Description
✅ Inherit from `Exception`	Base all your custom exceptions on the built-in class
✅ Include helpful attributes	Like `.name`, `.error_code`, or `.user_id`
✅ Use descriptive messages	Especially in `detail` or `message` field
🔁 Reuse with global exception handler	Useful for logging, formatting, or localization
🔐 Avoid exposing sensitive information	Don't return stack traces or internal details in production responses

---

🧩 Optional: Raise from Anywhere

You can raise your custom exception in any route, service, or utility:

```
def validate_story_character(name: str):
    if name.lower() == "badwolf":
        raise StoryException(name)
```

Then in route:

```
@app.get("/validate/{name}")
def validate(name: str):
    validate_story_character(name)
    return {"msg": "Character is valid!"}
```

🔄 Summary Flow

```
User Request → Route Function → Raise Custom Exception
                        ↓
             Custom Exception Handler (decorated)
                        ↓
                JSON Response with status + message
```

💡 Real-World Use Cases

Exception Name	Use Case
`UserNotFoundException`	When user is not found in DB
`TokenExpiredException`	For expired JWTs or session tokens
`PermissionDenied`	Role-based access control
`PaymentDeclined`	E-commerce payment error
`StoryException`	Fun use-case like the example above

🧰 Need a Custom BaseException Class?

You can even abstract this:

```
class CustomBaseException(Exception):
    def __init__(self, message: str, status_code: int = 400):
        self.message = message
        self.status_code = status_code
        super().__init__(message)
```

✅ Unified Custom Exception Handling in FastAPI (Express-Style)

We'll combine:

- 🔥 Your `ApiException` class
- 🎯 Custom exceptions (like `StoryException`)
- 📦 Centralized handler
- 🧩 Reusable pattern for scalable APIs

📁 Folder Structure (Recommended)

```
core/
├── api_exceptions.py     # Base exception class (ApiException)
├── custom_exceptions.py  # All your domain-specific custom exceptions
├── error_handler.py      # Global exception handler logic
```

🔸 `core/api_exceptions.py` (Base like ApiError.js)

```python
# core/api_exceptions.py

class ApiException(Exception):
    def __init__(self, message: str = "Something went wrong", status_code: int =
500, errors: list = None):
        self.status_code = status_code
        self.message = message
        self.success = False
        self.errors = errors or []
        super().__init__(message)
```

🔸 `core/custom_exceptions.py` (Like domain-specific error classes)

```python
# core/custom_exceptions.py

from core.api_exceptions import ApiException

# 🎭 Example: Story-specific error
class StoryException(ApiException):
    def __init__(self, name: str):
```

```
        super().__init__(
            message=f"Oops! '{name}' is not allowed in our story. 📚",
            status_code=418,  # I'm a teapot 😄
            errors=[f"Invalid character: {name}"]
        )

# 👤 Example: User-specific error
class UserNotFoundException(ApiException):
    def __init__(self, user_id: str):
        super().__init__(
            message=f"User with ID {user_id} not found 🚫",
            status_code=404,
            errors=[f"Missing user: {user_id}"]
        )
```

🔸 core/error_handler.py (Handles all ApiException globally)

```
# core/error_handler.py

from fastapi import Request
from fastapi.responses import JSONResponse
from fastapi.exceptions import RequestValidationError
from core.api_exceptions import ApiException
from starlette.status import HTTP_500_INTERNAL_SERVER_ERROR

def register_exception_handlers(app):

    @app.exception_handler(ApiException)
    async def handle_api_exception(request: Request, exc: ApiException):
        return JSONResponse(
            status_code=exc.status_code,
            content={
                "success": False,
                "message": exc.message,
                "errors": exc.errors,
                "data": None
            }
        )

    @app.exception_handler(RequestValidationError)
    async def handle_validation_error(request: Request, exc:
RequestValidationError):
        return JSONResponse(
            status_code=422,
            content={
                "success": False,
                "message": "Validation failed",
                "errors": exc.errors(),
```

```
                "data": None
            }
        )

    @app.exception_handler(Exception)
    async def handle_unexpected_error(request: Request, exc: Exception):
        return JSONResponse(
            status_code=HTTP_500_INTERNAL_SERVER_ERROR,
            content={
                "success": False,
                "message": "Internal server error",
                "errors": [str(exc)],
                "data": None
            }
        )
```

◆ `main.py` (Register exception handler globally)

```
# main.py

from fastapi import FastAPI
from core.error_handler import register_exception_handlers
from routes import story

app = FastAPI()

register_exception_handlers(app)  # 💥 Plug in your global exception system
app.include_router(story.router)
```

◆ Example Usage: `routes/story.py`

```
# routes/story.py

from fastapi import APIRouter
from core.custom_exceptions import StoryException

router = APIRouter(prefix="/story", tags=["Story"])

@router.get("/{name}")
async def get_story(name: str):
    if name.lower() == "badwolf":
        raise StoryException(name)
    return {"message": f"{name} is an awesome character! 🎭"}
```

🧪 Sample Response

```
{
  "success": false,
  "message": "Oops! 'badwolf' is not allowed in our story. 📚",
  "errors": ["Invalid character: badwolf"],
  "data": null
}
```

🧠 Benefits of This Unified Pattern

✅ **Express-like developer experience** ✅ **Reusable error logic with class-based exceptions** ✅ **Custom status codes and error messages** ✅ **Plug-and-play across multiple domains (auth, payments, stories, etc.)** ✅ **Centralized error formatting**

✅ Bonus: Custom Decorator `@async_handler` (Optional)

You can still wrap your route with this to catch unknown exceptions:

```
# core/async_handler.py

from functools import wraps
from core.api_exceptions import ApiException

def async_handler(func):
    @wraps(func)
    async def wrapper(*args, **kwargs):
        try:
            return await func(*args, **kwargs)
        except ApiException:
            raise  # Let global handler take over
        except Exception as e:
            raise ApiException(message=str(e), status_code=500)
    return wrapper
```
