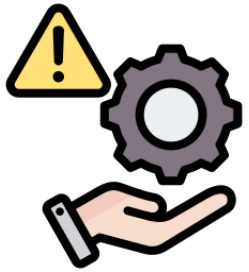
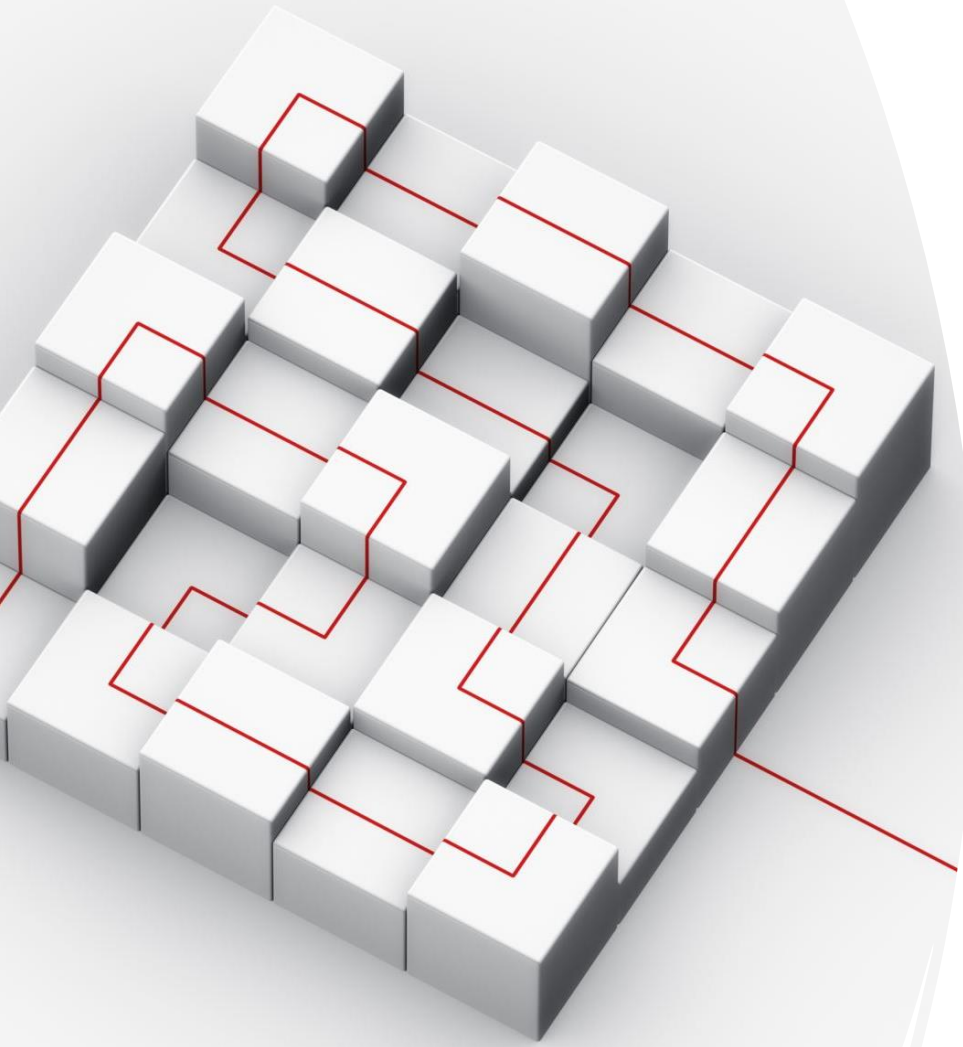
INT 161

Basic Backend Development

RESTful API
Exception Handling

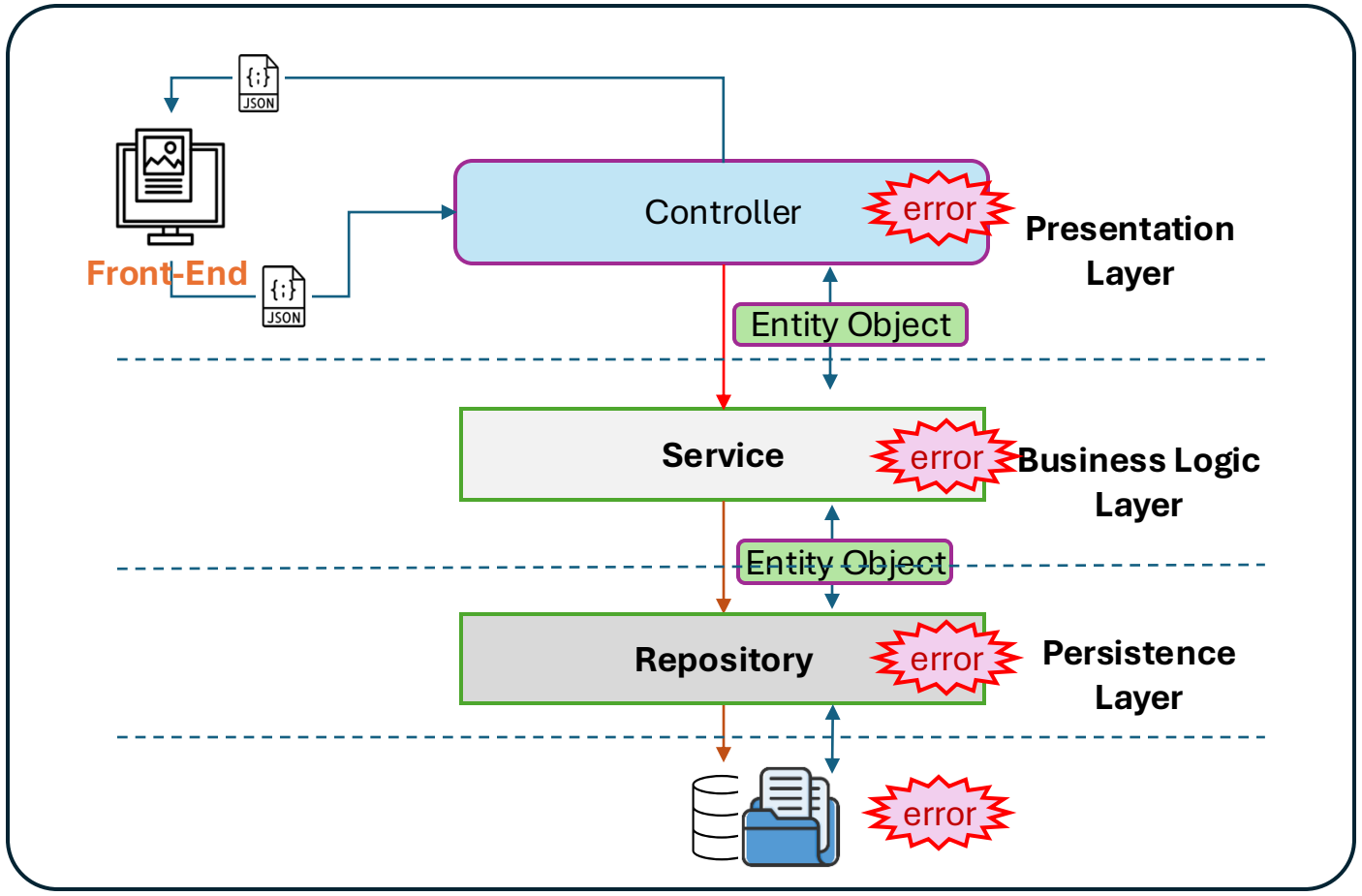




Unit Objectives

- After completing this unit, you should be able to:
  - Understand error types: programming vs operational errors
  - Learn error handling in synchronous & asynchronous code (callbacks, Promise, async/await)
  - Design global error handling in Node/Express with best practices (logging, shutdown, security)

Layered System



The **Layered System** principle means that a REST API is designed as a set of layers, where each layer has a specific role, and a client does not need to know whether it's communicating directly with the end server or through intermediaries.

This allows **scalability, flexibility, and separation of concerns**.

Error

- An error is an event or condition that disrupts the normal flow of execution in a program
  - The condition causes the system to throw an exception
  - The flow of control is interrupted and a handler will catch the error
  - Error terminate execution unless they are handled by the program
- Error is object
  - It encapsulates unexpected conditions in an object

```
const fs = require('node:fs');

function main(value=0) {
   console.log(a(value));
}
function a(value) {
   return b(value);
}
function b(value) {
   if (value > 0) {
      return ++value;
   }
   const data = fs.readFileSync('test.txt');
}

main(); //without error handling
```

Why Error Handling Matters

- Prevent app crashes / downtime
- Improve user experience (friendly error messages)
- Protect sensitive data from leaking (stack traces)
- Support debugging, monitoring, and alerting
- Error Types
  - Programming Errors → bugs, logic issues (TypeError, invalid assumptions)
  - Operational Errors → network timeouts, DB unavailable, file not found
  - Expected vs Unexpected Errors → handle expected gracefully

Error propagation

- Error propagation is a mechanism that allows errors to be passed up the call stack until they are caught and handled.
- This is crucial for debugging and ensuring that errors do not cause the entire application to crash unexpectedly.
- How errors propagate
  - When an error occurs in a function, it can either be caught and handled within that function or propagate up the call stack to the calling function.
  - If the calling function does not handle the error, it continues to propagate up the stack until it reaches the global scope, potentially causing the program to terminate.

Error Handling (Sync) : try .. catch

```
const fs = require('node:fs');

function main(value=0) {
   console.log(a(value));
}
function a(value) {
   return b(value);
}
function b(value) {
   if (value > 0) {
      return ++value;
   }
   const data = fs.readFileSync('test.txt');
}

main(); //without error handler
```

```
try {
   main(1);
   console.log('After main success');
   console.log('-----------------');
   main();
   console.log('After main error');
} catch (e) {
   console.log('Message: ', e.message);
   console.log('Status: ', e.status);
   console.log('Code: ', e.code);
   console.log('Stack Trace: ', e.stack);
}

console.log('Program was normal ended');
```

Error Handling (Async) : Promises + .catch

```
import {promises as fs} from 'node:fs';

async function main(value = 0) {
    console.log(await a(value));
}

async function a(value) {
    return await b(value);
}

async function b(value) {
    if (value > 0) {
        return ++value;
    }
    const data = await fs.readFile('data.txt', 'utf8');
    return data;
}
```

```
main(1).then(() => console.log('After main success'))
.catch(e => handler(e));
main().then(() => console.log('After main error'))
.catch(e => handler(e));

function handler(e) {
    console.log('Error handler');
    console.log('============================');
    console.log('Message: ', e.message);
    console.log('Status: ', e.status);
    console.log('Code: ', e.code);
    // console.log('Stack Trace: ', e.stack);
};
```

The standard JavaScript Error object

- message: A human-readable description of the error.
- name: The name of the error type (e.g., "Error", "TypeError", "ReferenceError").
- stack: (Non-standard but widely supported) A string representing the stack trace, indicating where the error occurred in the code.
- Additionally, when dealing with system errors in Node.js
  - code: A string representing the system error code (e.g., "EACCES" for permission denied).
  - errno: A number corresponding to the negated system error code.

Customer Error

```
  :
  :
async function b(value) {
    if (value > 0) {
        return ++value;
    }
    try {
        const data = await fs.readFile('data.txt', 'utf8');
        return data;
    } catch (e) {
        const err =  new Error('File data.txt not found');
        err.code = 'FILE_NOT_FOUND';
        err.status = 404;
        throw err;
    }
}
```

Custom Error Classes:

- For more specific error handling, you can create custom error classes (e.g., AppError) to categorize and add more context to errors.
- This can help in providing more informative error responses to clients and in internal logging.

```
class AppError extends Error {
   constructor(message, statusCode) {
      super(message);
      this.statusCode = statusCode;
      this.status = `${statusCode}`.startsWith('4') ? 'fail' : 'error';
      this.isOperational = true; // Indicate if it's an expected operational error
      Error.captureStackTrace(this, this.constructor);
   }
}
```

Express Middleware

Client (Request)
↓
Express.js Server (http)
↓
[Middleware 1] → next()
↓
[Middleware 2] → next()
↓
[Router Match] → handler
↓
[Response] → res.send() / res.json()
↓
Client (Response)

- middleware คือฟังก์ชันที่รับ (req, res, next)
- สามารถทำงานบางอย่าง เช่น log, parse body, auth, validate
- ถ้าเรียก next() ⟶ ส่งต่อไป middleware ถัดไป
- ถ้า res.send() หรือ res.end() ⟶ end (response to client)

```
// Middleware 1 - Logger
app.use((req, res, next) => {
    console.log(`${req.method} ${req.url}`)
    next()
})

//Middleware 2 - JSON parser
app.use(express.json())

app.use('/films', filmRouter);
```

Error Handling Mechanism in Express.js

- Express.js has a special mechanism to deal with errors using error-handling middleware.
  - Error Occurs
    - Your code (throw new Error, rejected Promise, etc.)
    - Middleware/Route failures (e.g., DB query error, invalid input)
  - Forward Error with next(err)
    - In Express, if you call next(err), Express will skip all normal middleware/route handlers and jump directly to an error-handling middleware.
  - Error Handling Middleware
    - Normal middleware signature: (req, res, next)
    - **Error-handling middleware signature: (err, req, res, next)**
    - These functions are used to catch and respond to errors.
  - Send Response Back, Typically, you send an appropriate HTTP status code and an error message/JSON.
    - 400 Bad Request
    - 401 Unauthorized
    - 500 Internal Server Error

Handling Errors

- The first step in handling errors is to provide a client with a proper **status code**. Additionally, we may need to provide more information in the response body.
- Basic Responses
  - The simplest way we handle errors is to respond with an appropriate status code.
  - Here are some common response codes:
    - 400 Bad Request - client sent an invalid request, such as lacking required request body or parameter
    - 401 Unauthorized - client failed to authenticate with the server
    - 403 Forbidden - client authenticated but does not have permission to access the requested resource
    - 404 Not Found - the requested resource does not exist
    - 409 Conflict - the request is conflict with the current state of the target resource.
    - 412 Precondition Failed - one or more conditions in the request header fields evaluated to false
    - 500 Internal Server Error - a generic error occurred on the server
    - 503 Service Unavailable - the requested service is not available

Standardized Response Bodies

- In an effort to standardize REST API error handling, the IETF devised RFC 7807, which creates a generalized error-handling schema.
- This schema is composed of five parts:
  - type - a URI identifier that categorizes the error
  - title - a brief, human-readable message about the error
  - status - the HTTP response code (optional)
  - detail - a human-readable explanation of the error
  - instance - a URI that identifies the specific occurrence of the error

```
{
    "type": "https:/host/errors/incorrect-user-password",
    "title": "Incorrect username or password.",
    "status": 401,
    "detail": "Authentication failed due to incorrect username or password.",
    "instance": "/login/log/abc123"
}
```

Express: Error-handling Middleware

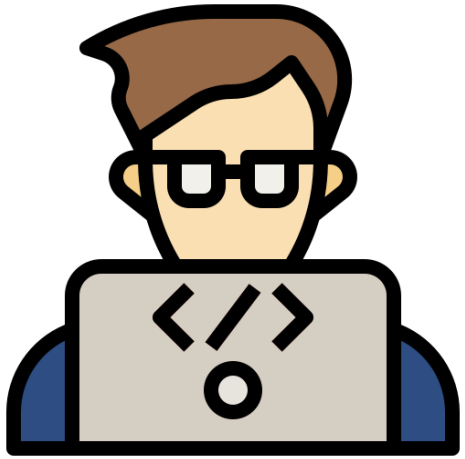
```
// error handler
app.use(function (err, req, res, next) {
    const status = err.status || 500;
    res.status(status);
    res.json(
        {
            error: err.statusCode,
            statusCode: status,
            message: err.message,
            path: req.originalUrl,
            timestamp: new Date().toLocaleString()
        }
    );
});
```

- Error-handling middleware must be defined **at the end**, after all routes/middlewares.
- If no error handler is defined, Express will send a default HTML error response (with stack trace in dev mode).

The Problem: Async Errors are Not Caught

- By default:
  - Synchronous errors (throw) → caught by Express automatically.
  - Asynchronous errors (async/await, Promise rejection) → NOT caught automatically.
  - Example: errors from Prisma queries will not reach error-handling middleware.
- Solutions:
  - Wrap async code in try/catch:
  - Create a reusable wrapper:
  - Use express-async-errors Library

Practices



Middleware Error Handler: app.js

```
// catch 404 and forward to error handler
app.use(function (req, res, next) {
  next(createError(404));
});

// error handler
app.use(function (err, req, res, next) {
  const status = err.status || 500;
  const errorCode = err.code || (err.message.toUpperCase().replace(/ /g, '_'));
  res.status(status);
  res.json({
    error: errorCode,
    status: status,
    message: err.message,
    resource: req.originalUrl,
    timestamp: new Date().toLocaleString()
  });
});
```

Remove try-catch : country-controller.js

```
var service = require('../services/country-service');

module.exports = {
    list: async function (req, res) {
        const countries = await service.getAll();
        res.json(countries);
    },
    get: async function (req, res) {
        const id = Number(req.params.id);
        const {includeCity} = req.query;
        const country = await service.getById(id, includeCity);
        res.json(country);
    },
    update: async function (req, res) {
        const data = req.body;
        const id = Number(req.params.id);
        const country = await service.update(id, data);
        res.json(country);
    }
}
```

Add error-response.js, Edit: country-service.js

```
const repo = require('../repositories/country-repository');
const errResp = require('../errors/error-response');

module.exports = {
   getAll: async function () {
      return  await repo.findAll();
   },
   getById: async function (id, includeCity = false) {
      const uniqueOne = await repo.findById(id, includeCity);
      if (!uniqueOne) throw errResp.notFoundError(id, 'Country');
      return uniqueOne ;
   },
   update: async function(id, data) {
      return await repo.update(id, data);
   }
}
```

errors/error-response.js

```
module.exports = {
   notFoundError: function (id, resource) {
      const  err = new Error(`${resource} not found for ID ${id}`);
      err.code = 'NOT_FOUND';
      err.status = 404;
      return err;
   },
}
```

Edit: country-repository.js

```
const {PrismaClient} = require("../generated/prisma");
const prisma = new PrismaClient();
module.exports = {
   findAll: async function () {
      return await prisma.country.findMany();
   },
   findById: async function (id, includeCity = false) {
      return await prisma.country.findUnique({
         where: {id: id},
         include: includeCity ? {
            cities: {
               select: {
                  id: true,
                  city: true,
               }
            }
         } : false,
      });
   },
   update: async function (id, data) {
      return await prisma.country.update({
         where: {id: id},
         data: data,
      });
   },
}
```

Test:

GET : localhost:3000/country

GET : localhost:3000/countries/94

GET : localhost:3000/countries/911 

PUT : localhost:3000/countries/1323 


```
{
    "country": "Thailand Dan Smile"
}
```

Create a reusable wrapper: `country-route.js`

```
var express = require('express');
var router = express.Router();
const controller = require('../controllers/country-controller');

const asyncWrapper =  (fn) => (req, res, next) => {
   Promise.resolve(fn(req, res, next)).catch(next);
};

router.get('/', asyncWrapper(controller.list));
router.get('/:id', asyncWrapper(controller.get));
router.put('/:id', asyncWrapper(controller.update));

module.exports = router;
```

Test:

GET : localhost:3000/**country**

GET : localhost:3000/countries/94

GET : localhost:3000/countries/911

PUT : localhost:3000/countries/1323

```
{
    "country": "Thailand Dan Smile"
}
```

GET : localhost:3000/**films**/1911

Use express-async-errors Library, app.js

```
> npm install express-async-errors
```

```
 :
var cookieParser = require('cookie-parser');
var logger = require('morgan');


require('express-async-errors');


var countryRouter = require('./routes/country-route');
var filmRouter = require('./routes/film-route');
var cors = require('cors');
var app = express();
 :
```

Test:

GET : localhost:3000/**country**

GET : localhost:3000/countries/94

GET : localhost:3000/countries/911

PUT : localhost:3000/countries/1323

```
{
    "country": "Thailand Dan Smile"
}
```

PUT : localhost:3000/countries/1

```
{
    "country": "Thailand"
}
```

GET : localhost:3000/**films**/1911

Edit Error Response, error-response.js

```
module.exports = {
   notFoundError: function (id, resource) {
      const  err = new Error(`${resource} not found for ID ${id}`);
      err.code = 'NOT_FOUND';
      err.status = 404;
      return err;
   },
   duplicateItem: function (itemName, resource) {
      const  err = new Error(`${resource} already exists for ${itemName}`);
      err.code = 'DUPLICATE_RESOURCE';
      err.status = 409;
      return err;
   }
}
```

Edit Repository, country-repository.js

```
const {PrismaClient} = require("../generated/prisma");
const prisma = new PrismaClient();

module.exports = {
   :
   :
   update: async function (id, data) {
      return await prisma.country.update({
         where: {id: id},
         data: data,
      });
   },
   findByCountryName: async function (countryName) {
      return await prisma.country.findFirst({
         where: {country : countryName},
      });
   },
}
```

Edit Repository, country-service.js

```
getById: async function (id, includeCity = false) {
   const uniqueOne = await repo.findById(id, includeCity);
   if (!uniqueOne) throw errResp.notFoundError(id, 'Country');
   return uniqueOne;
},
update: async function (id, data) {
   await this.getById(id);
   const sameNameCountry = await repo.findByCountryName(data.country);
   if (sameNameCountry && sameNameCountry.id !== id) {
      throw errResp.duplicateItem(data.country, 'Country');
   }
   return await repo.update(id, data);
}
```