Lab 1- 2331200153

QUESTION 1 – BACKEND ARCHITECTURE & CONTROL

1. Describe overall architecture:

- Client: Web application clients that use the system API. Send request and display responses from the API to end users (librarians, members, admins).

- ASP.NET Core Web API:

+ Controllers

+ Models

+ Dtos

+ Interfaces

+Services

+ Middleware

Handle request with REST API endpoints. Contains Controllers (endpoints), Services (business logic), Repositories (data access(dto,models)), Dto (data transfer object), Middleware (security, error handling).

- Database:

+Books

+Users

+BookBorrow

+ Authors

+ Catergories

+ Staff

Managing Entities: Books, Users, Transactions, Book reservations, Authors, Catergories, Staff.

Relationships between components:

Request Flow:

Client -> API Request ->API Controller ->Business Service ->Repository ->Database

Response Flow:

Database ->Repository ->Business Service ->API Controller ->API Response ->Client

Design Principles:

+ Separation: separate responsibilities for each layer

+ Dependency Injection: Loose coupling between components

+ Repository: Abstracts data access logic

+ Service Layer: Contain validation and business rule

2.**Design the business workflow**

1. Book Borrowing Flow (POST /api/borrow)

Client -> Controller -> Service -> Repository -> Database -> Response

+ Controller validate request model, authenticates user

+ Service validate member exists, member active, book available, member borrowing limit

+ Repository create transaction record

2. Book Return Workflow (POST /api/return)

Client -> Controller -> Service -> Repository -> Database -> Response

+ Service validate transaction exist

+ Repository updates transaction, update Book, calculate late fees

+ Database commit

+ Response with fee detail

3. Book Reservation Workflow (POST /api/reserve)

Client -> Controller -> Service -> Repository -> Database → Response

+ Service validate book borrowed, member eligible

+ Repository create reservation with queue

+ Database commit

+ Response find available date

4. Member Registration (POST /api/members)

Client -> Controller -> Service -> Repository -> Database -> Response

+ Controller validate input

+ Service check duplicate field

+ Repository create acc

+ Database commit

+ Response with member ID, card number

5. Fine Payment (POST /api/fines/pay)

Client -> Controller -> Service -> Repository -> Database -> Response

+ Service see if fine exists and unpaid

+ Repository updates Fine status to 'Paid', records payment

+ Database commits

+ Response with receipt

6. Search Books (GET /api/books/search)

Client -> Controller -> Service -> Repository -> Database -> Response

+ Client send query

+ Controller validate

+ Service apply filtering

+ Repository query using pagination

+ Database return output

+ Response with book

7.Manage user flow:

Client -> Controller -> Service -> Repository -> Database -> Response

+ Client send request

+ Controller handle request

+ Service mange business rule and flow

+ Repository manage quesry

+ Database store and retrieve user data

+ Resposne return result

3. Risk analysis without testing

- Borrowing Limit Bypass: Members can borrow more than the maximum allowed books like borrowing 6 while limit is 5

- Concurrent Book Borrowing: Multiple members borrow the same book at the same time because lack of proper locking mechanic

- Data Inconsistency: Transaction create but book status not updated, or otherway around

- Unauthorized Access: User borrow book for other or access endpoint without proper authentication/authorization

- Input Validation Failure: malicious input (SQL injection), or invalid Id

- Incorrect Fine Calculation: Late fee calculated incorrectly cause of multiple issu, business calculation error.

- Performance issues: System becomes slow under high load due to not optimized query, not using index for large db

- 3 Most critical Risk:

- Data Inconsistency

+ Technical Risk:

Transaction failure: Incomplete rollback make db in inconsistent state

Cache shows old data while database has new data

Cant track who changed what

+ Business Risk:

User frustrated

Wrong fine calculation

Loss of trust

- Unauthorized Access

+ Technical Risk:

Weak passwords, no addition secure layer(jwt,otp)

Member might have access to admin function

No protection for viewing other people info

+ Business Risk:

User personal info might get stolen

People might delete book or delete their fine

User leave cause of security issue

- Performance issues

+ Technical Risk:

Slow database query

System overload cause of too much data

No limit on amout of data load per request

+ Business Risk:

Page take long time to load

System might crash from too many user

Frustrate user

4.Role of testing

How testing helps the backend to:

- Be easier to maintain: Testing make easier to keep the system work without error

+ Find problems quickly: When something breaks, tests will help where it break instead of guess

+ Understand the code: Tests show how the code is supposed to work

+ Safe updates: When updating library or frameworks or changing a portion of code, tests check if everything still works

+ Less time debugging: Tests catch bugs so we don't have to find them later

- **Be easier to change:** Testing gives assurance to change code without breaking old code

+ Safety: Tests catch mistakes when changing code

+ Try new things cause test will warn if something breaks

+ Refactor: Improve code structure knowing tests will catch errors

+ Add features: New features wont accidentally break old features if good test and prevention is in place

- **Reduce production defects:** Testing catch bugs before pushing prod

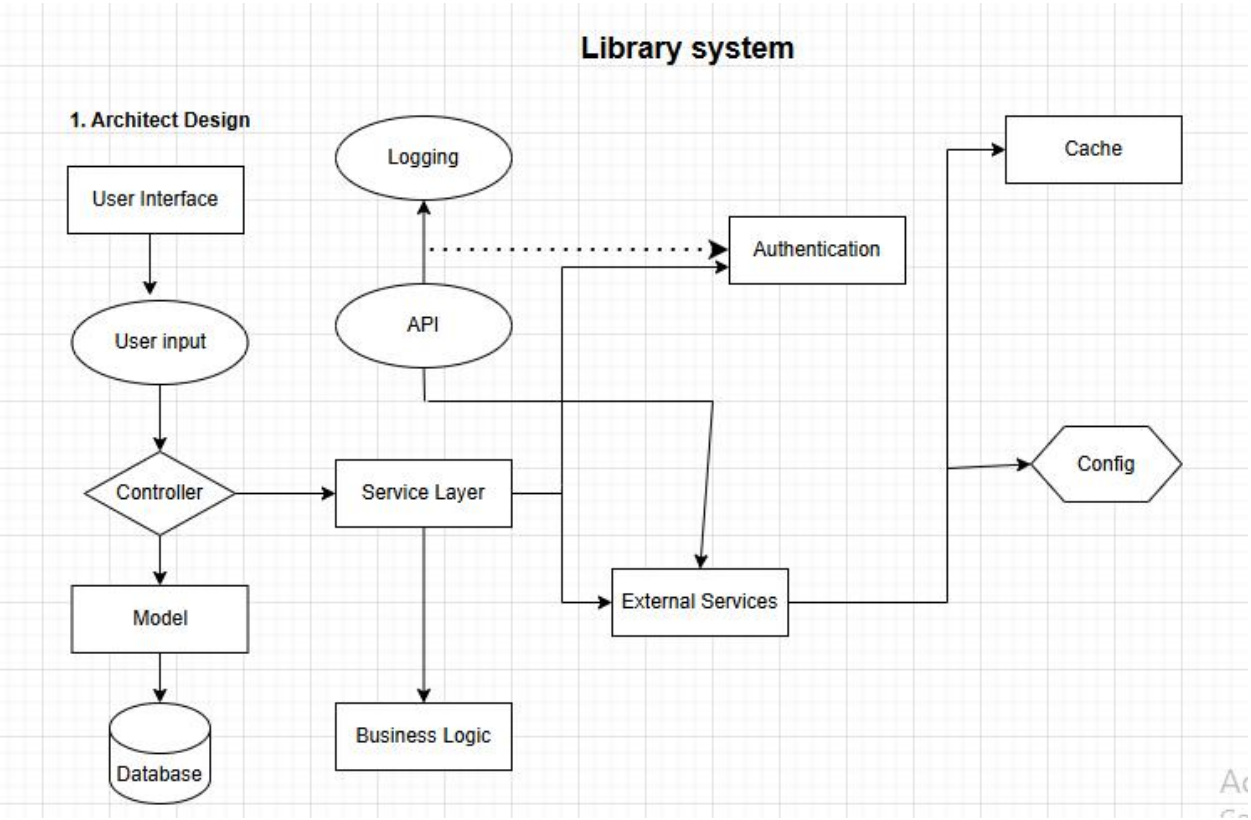
+ Catch bugs early: Find error on development, not on live prod

+ Test all scenarios: Tests check edge case

+ Prevent repeat bugs: Once a bug is fix, write a test to prevent it happen again

+ Less stress: Have good test will lessen the stress of fixing repeating bug

- Architect design:



QUESTION 2 – TESTING PRINCIPLES & TEST PROCESS

1.Define the testing scope (Test Scope)

Functions to be Tested, Risk-Based Testing

Function	Risk	Explanation
BookBorrowing POST /api/borrow	High	Core business function. Impact business rules
BookReturn POST /api/return	High	Financial impact fopr fine, important for inventory handling
MemberRegistering POST/api/members	High	Data validation. Duplicate prevention.
Authentication_Authorization	High	Security. Unauthorized access need to be prevent
PaymentHandling POST /api/fines/pay	High	Financial transactions. Payment errors affect money income
BookSearch GET /api/books/search	Normal	High usage.Need input validation. Will affect user experience

Parts not to be tested, risk-based testing

Part	Risk	Explanation
Logging & Monitoring	Low	Tested in integration tests only needed
UI	Low	Minimal business logic
Third-party library	Low	Should be tested by 3rd party

2. **Apply Testing Principles**

Principle 1: Testing Shows Presence of Defects, Not Their Absence

Explanation: Testing can prove that defects exist but cannot prove their absence. Even comprehensive testing cannot guarantee bug-free software.

Example: users can only borrow 5 books maximum, tests pass. But it exist a case where 2 users borrow the same book at the same time, and the system allows it even though only one copy exists.

Principle 2: Exhaustive Testing Is Impossible

Explanation: Testing all possible input combinations and execution paths is impossible. We must use risk analysis and prioritization.

Example: If library has 1000 books and 500 members. We cant test every combination of member + book + date. So we test with few invalid members and some unavailable books

Principle 3: Early Testing

Explanation: Defects found early in the development cycle are cheaper and easier to fix. Testing activities should start as early as possible.

Example: During review the borrow API doesnt check if a member is suspended. If this is found after deployment, suspended members could have already borrowed books incorrectly

Principle 6: Testing Is Context-Dependent

Explanation: Testing approaches depend based on the context. A library system is tested differently than a medical device or banking system.

Library System Example:

Example: For library focus is on testing business rules like borrow limits and fine calculation. No need the same level of security testing as a banking app.

3. Design test cases

- Design **10 most important test cases** for the API:
  - POST /borrow
- Each test case must include:
  - Test Case ID
  - Description

        - Input
        - Expected Result
        - Priority
    - Must include:
        - Positive cases
        - Negative cases
        - Boundary cases

Assumptions:

- Max books per member = 5

- Member must active

- Book is available

- Cant borrow same book twice

- Cant borrow if member has unpaid fines

Test Cases:

TC ID	Description	Input	Expected Result	Priority
TC1	Borrow book successfully	Valid member, valid book, member has < 5 books	Success	High
TC2	Member dont exist	Invalid memberId	Not Found	High
TC3	Book does not exist	Invalid bookId	Not Found	High
TC4	Book not available	Book already borrowed	Bad Request	High
TC5	Member reached limit (5 books)	Member already has 5 books	Bad Request	High
TC6	Member already has this book	Book borrowed by member and not yet returned	Bad Request	High
TC7	Member has unpaid fines	Member has unpaid fines	Bad Request	High
TC8	Member is suspended	Member status = suspended	Bad Request	Medium
TC9	Invalid input format	Null memberId or bookId	Bad Request	Medium
TC10	Concurrent borrow same book	Two members borrow same book at same time	One Success, One Fail	Low

4. Apply the Fundamental Test Process

- Describe the testing activities according to:
  - Test Planning
  - Test Analysis
  - Test Design
  - Test Execution

    Test Closure

Test Planning:

- Define test scope, objectives, resources, and schedule

- Select test types: unit test (BorrowService, FineService), integration test, API test, Concurrency tests, Performance tests

- Identify risks and priorities

Test Analysis:

- Analyze requirements and business rules (Max 5 books per member, Book must be available, Member must be active)

- Identify test conditions for borrow, return, reserve, fine calculation, invalid member, ID book already borrowed, member at borrowing limit, concurrent requests

- Review API specifications

Test Design:

- Prepare test cases (like the 10 borrow test cases above)

- Design test data

- Set up test environment

- Set priority

+ High: Core flows, concurrency, security

+ Medium: Edge cases, validation

+ Low: Error messages, logging

Test Execution:

- Run test cases

- Record log result

- Compare actual results with expected results

- Test regresion

Test Closure:

- Summarize test results

- Count number of defects found and fixed

- Generate test report

- Evaluate test coverage achieved

- Write report

5. Defect handling

- Write **one defect report** for the issue:

  "A user can borrow more than 5 books"

- Identify:
  - Severity
  - Priority

Defect Report: A member can borrow more than 5 books

Member can borrow more than 5 books

Description**:** The system allows a member to borrow more than the maximum limit of 5 books

Bug squence:

1. Member borrows 5 books

2. Member attempts to borrow a 6th book

3. System allows the borrow request

Expected: System should reject the request with "Borrowing limit exceeded" error

Actual: System allows the borrow and member now has 6 books

Severity: High

- Breaks core business rule

- Causes data inconsistency

Priority: High

- Must be fixed before release

- Affects main business functionality

QUESTION 3 – VERIFICATION, VALIDATION & STOP TESTING

Requirements

Evaluate the system from the perspectives of Verification, Validation, Limitations, and Decision Making.

1. Verification vs Validation

Classify the following activities:

- API code review
- Unit testing for BorrowService (.NET)
- Checking API compliance with the design documentation
- Business testing of the borrowing process with a librarian

- API code review -> **Verification**

Checks if code follows coding standards and design specifications. Does not check real user needs, only implementation correctness. verifies the code is written correctly according to architectural guidelines

- Unit testing for BorrowService (.NET) -> **Verification**

Verifies BorrowService behave correctly according to their specifications. test internal logic correctness, not if the feature meets user needs

- Checking API compliance with the design documentation -> **Verification**

Checks if API matches designed endpoints and formats. Ensures system is built according to documented specifications, ensuring the product was built according to the design.

- Business testing of the borrowing process with a librarian -> **Validation**

Perform real world borrow workflow to confirm the system supports daily operations effectively. Checks if system fits real business workflows. Librarian represents real user and confirms system solves real problem.

2. Facts, Myths & Limitations of Testing

- Classify and explain:
  - Testing can find all defects
  - It is impossible to test all scenarios
  - Testing helps reduce risk

Classify and explain:

- Testing can find all defects -> **Myth**

Testing can only show presence of defects, not their absence. Impossible to guarantee all bugs are found. Complex systems have infinite possible states and interactions. Even with 100% code coverage issues may remain undiscovered until product is push to prod

- It is impossible to test all scenarios -> **Fact**

Too many input combinations, system states, and environments exist. Exhaustive testing is not feasible in real projects.

- Testing helps reduce risk -> **Fact**

Testing is a risk mitigation activity. Testing reduces probability of critical failures in production. Improves system stability and reliability, but cannot eliminate all risks.

3. When to Stop Testing

- Propose at least **4 criteria** for stopping testing for the library backend

Criteria 1: Code Coverage Threshold Met

- Should achieve minimum 80% coverage

- Borrow, return, and fine calculation tests pass successfully

- All high-priority test cases completed

Criteria 2: All High Priority Test Cases Pass

- No open Critical or High severity bugs remain
- Minor bugs are acceptable or postponed

Criteria 3: Defect Coverage Below Threshold

- 80-90% code coverage for service layer achieved
- All main business flows are tested
- Zero critical/high severity open defects

Criteria 4: Time and Budget Constraints Met

- Project reaches testing deadline
- Testing timeline and budget exhausted, with assessment acceptable for release
- Further testing would not bring significant value

Criterion 5: No New Defects Found in Last Test Cycles

- Zero new defects discovered during the last 2 test cycles
- Regression testing dont find new bug, suggests the system has reached good enough stability
- Stable but minimally tested code is still risky