Web programming using Java technologies

Answer Key – Exam

Part I – Single-Choice Questions

1. C – GET

2. B – 201

3. B – Dependency Injection

4. B – @Bean

5. B – Cache entities and track changes

6. B – LAZY

7. C – Unit test

8. B – @Transactional

9. B – Invalid request payload

10. B – Fast and deterministic execution

Part II – Multiple-Choice Questions

11. Correct answers:

- Use meaningful HTTP status codes

- Keep APIs stateless

- Use nouns rather than verbs in endpoints

12. Correct answers:

- Beans are created and managed by the Spring container

- Dependency injection reduces tight coupling

- The context controls bean lifecycle

13. Correct answers:

- @ManyToOne relationships are typically mapped on the owning side

- Bidirectional relationships must be kept consistent in application code

14. Correct answers:

- Business rule enforcement

- Transaction demarcation

- Coordination of multiple repositories

15. Correct answers:

- Dependencies should be mocked when appropriate

- Unit tests should avoid loading the full Spring context

- Unit tests should be fast

16. Correct answers:

- Verifying repository queries

- Checking transaction behavior

- Validating JPA mappings

Part III – Short Open Questions (Expected Coverage)

17. Constructor injection preferred for immutability, explicit dependencies, and testability.

18. Centralized exception handling via @ControllerAdvice; consistency and reduced duplication.

19. Declarative transactions with @Transactional at service layer; atomicity across multiple repositories calls.

20. Cross-cutting concerns such as logging, transactions, monitoring, auditing.

21. Business logic and interactions with mocked dependencies.

22. To keep tests fast, reliable, deterministic, and isolated from infrastructure.

Part IV – Real-World Scenarios (Expected Coverage)

23. **Model solution – Expected issues (code review)**

1) Wrong HTTP method for a state-changing operation

- **Problem:** GET /borrowBook changes server state (creates a loan, updates a book). GET must be safe and cacheable.
- **Fix:** Use POST or PUT.
    - Example: POST /loans or PUT /books/{bookId}/loan.

2) Non-RESTful endpoint naming / RPC style

- **Problem:** /borrowBook is verb-based and action-oriented; REST favors resources.
- **Fix:** Model resources explicitly:
    - POST /loans with body {bookId, userId}
    - or PUT /books/{bookId}/loan (creates/updates "current loan" subresource).

3) userId passed as request parameter (API design / security smell)

- **Problem:** Using userId from request parameters enables "borrow on behalf of someone else" unless validated elsewhere; also muddles responsibility.
- **Fix:** In real systems the user comes from authentication context. If auth is out-of-scope, still prefer a request body DTO and validate.

4) Incorrect error handling: returns 200 OK on failures

- **Problem:** Catch-all Exception and returning 200 breaks HTTP semantics and client logic (clients can't reliably detect failures).
- **Fix:** Return proper status codes and consistent error bodies; use centralized exception handling via @ControllerAdvice.
    - Not found → 404
    - Business rule conflict → 409
    - Validation → 400

5) Overly broad exception handling (catching Exception)

- **Problem:** Masks programming errors and infrastructure failures; makes debugging and monitoring harder.
- **Fix:** Catch specific exceptions or allow them to propagate to a global handler.
    - Use custom exceptions: BookNotFoundException, BookNotAvailableException, etc.

6) Response type and body design are poor

- **Problem:** ResponseEntity<Object> returning a string message is not a good API contract; unstable for clients and hard to evolve.
- **Fix:** Return a proper DTO for created resource (e.g., LoanResponseDto) and/or Location header.
    - For creation: 201 Created with body containing loanId and key fields.

7) Missing transaction boundary for a multi-step write

- **Problem:** Service performs multiple writes (bookRepository.save, loanRepository.save, etc.) without a transaction → partial updates possible (book becomes unavailable but loan insert fails).
- **Fix:** Put transaction boundary at service layer:
    - @Transactional on borrowBook (service method), not controller.
    - Ensure atomicity of "mark unavailable + create loan".

8) Business rule is missing entirely: availability check removed

- **Problem:** Code always sets book unavailable and creates a loan, even if already borrowed.
- **Fix:** Validate:
    - if (!book.isAvailable()) → throw domain exception
    - Better: base truth on active loan existence rather than a boolean flag (or keep both consistent).

9) Double save of the same entity (bookRepository.save(book) twice)

- **Problem:** Redundant, noisy, potentially causes extra SQL flushes; indicates misunderstanding.
- **Fix:** Save once; rely on persistence context and dirty checking (within transaction). If explicit save is used, do it once.

10) Throwing generic RuntimeException for expected domain cases

- **Problem:** Makes mapping to HTTP statuses inconsistent; encourages 500 responses or poor error handling.
- **Fix:** Use domain exceptions and map them via @ControllerAdvice:
    - BookNotFoundException → 404

- BookNotAvailableException → 409

11) Missing validation on inputs

- **Problem:** No check for null/negative IDs; no request validation.
- **Fix:** Use request DTO + Bean Validation (@Valid, @NotNull, etc.) and return 400 on validation failures.