Databases & Web Programming

Sample Paper 2 — Mark Scheme & Model Answers

Equivalent phrasings receive full credit. Green italic notes are study tips.

Question 1 — Databases (30 marks)

1(a)(i) — Anomalies in the gym schema [4 marks]

Currently suffering: Update anomaly — instructor's phone number is stored on many rows; changing it in one place leaves the others inconsistent (2 marks).

Further anomaly: any of —

• **Deletion anomaly:** when the last booking for an instructor is deleted, the instructor's record disappears entirely.

• **Insertion anomaly:** a new instructor can't be added until they appear in a booking row. (2 marks)

1(a)(ii) — CREATE TABLE Instructor [6 marks]

```
CREATE TABLE Instructor (
    InstructorID   INT PRIMARY KEY,
    Name           VARCHAR(100) NOT NULL,
    Phone          VARCHAR(20) UNIQUE,
    Email          VARCHAR(100),
    HireDate       DATE
);
```

Marking: PK (1), NOT NULL on name (1), UNIQUE on phone (1), reasonable data types incl. INT for id and VARCHAR for text (2), correct SQL syntax / semicolon (1).

1(b) — Hotel stays to 3NF [10 marks]

Decomposition (8 marks — 2 per table) plus the dependency statements (2 marks):

```
Guest(GuestID PK, GuestName, GuestPhone)
RoomType(RoomType PK, NightlyRate)
Room(RoomNo PK, RoomType FK)
Stay(StayID PK, GuestID FK, RoomNo FK, CheckIn, Nights)
```

Driving dependencies:

• `RoomNo` → `RoomType` and `RoomType` → `NightlyRate` — transitive, so `NightlyRate` moves to a separate `RoomType` table.

• `GuestName, GuestPhone` depend on the guest, not on the stay — lift them into a `Guest` table.

■ *Accept reasonable variations (e.g. introducing a numeric surrogate* `RoomTypeID`*); core test is that update of* `NightlyRate` *for "Deluxe" now happens in one row.*

1(c)(i) — Authors with zero books [6 marks]

```
SELECT a.Name,
       COUNT(b.BookID) AS BookCount
FROM   Author a
LEFT JOIN Book b ON b.AuthorID = a.AuthorID
GROUP BY a.AuthorID, a.Name
ORDER BY BookCount DESC;
```

Marking: LEFT JOIN (2), COUNT on book id (1, not `COUNT(*)` — see (ii)), GROUP BY (1), ORDER BY DESC (1), correct selection of author name (1).

1(c)(ii) — Why not INNER JOIN? [4 marks]

INNER JOIN only returns rows where there is a match in *both* tables. An author with zero books has no row in `Book`, so the JOIN excludes them entirely — producing zero rows for that

author rather than a row with count `0`. **LEFT JOIN** keeps every row from `Author` and pads missing Book rows with NULL, which `COUNT(b.BookID)` treats as 0.

■ *This is also why we count* `b.BookID` *and not* `*`*: COUNT(*) would always count at least 1 row even when everything in the Book columns is NULL.*

Question 2 — WP Principles (30 marks)

2(a)(i) — Path Variable vs Request Parameter [6 marks]

Path variable (e.g. `2025` in `/products/2025`) is part of the URL path itself and *identifies a specific resource*. It's hierarchical, required, and changes which thing you're addressing (3 marks).

Request parameter (the bits after ?) is appended to the URL as `key=value` pairs and is used to *filter, sort or paginate* a collection. Optional by nature, and changing it doesn't change which endpoint you're hitting (3 marks).

2(a)(ii) — When to use which [4 marks]

Use a path variable when the value identifies one specific resource — e.g. `/api/v1/users/42`. Without the 42, you wouldn't know which user you're asking about (2 marks).

Use a request parameter when you're filtering or sorting a collection — e.g. `/api/v1/products?maxPrice=500&sort=price,asc`. The base endpoint stays the same; the parameters narrow the result set (2 marks).

2(b)(i) — POST on a specific id [4 marks]

405 Method Not Allowed. The path `/products/2025` addresses a single existing resource; the conventional method to *update* it is PUT or PATCH, not POST. The server advertises which methods that URL supports via the `Allow` response header. (4 marks)

■ *Some implementations return **400 Bad Request**; both are accepted with justification, but 405 is the most idiomatic and is what JHipster does by default.*

2(b)(ii) — PUT vs PATCH [4 marks]

PUT replaces the entire resource — the client must send all fields, and missing fields are treated as set-to-null. **PATCH** applies a partial update — send only the fields that changed (2 marks).

PATCH is preferable when updating one or two fields on a resource with many fields — e.g. flipping `isActive` from `true` to `false` on a user, without having to resend name, email, address etc. (2 marks)

2(c) — Spring Boot layers [12 marks]

Folder responsibilities — 2 marks each:

• **domain/** — JPA-annotated Java classes representing your entities (the data shape). Typical annotation: `@Entity` (or `@Table`, `@Id`, `@Column`).

• **repository/** — Spring Data JPA interfaces that persist domain objects. Typical annotation: `@Repository` (or `@Query` for custom JPQL).

• **service/** — optional classes that hold business logic and orchestrate repositories. Typical annotation: `@Service` (or `@Transactional`).

• **web/rest/** — REST controllers that expose endpoints over HTTP. Typical annotation: `@RestController` (or `@GetMapping`, `@PostMapping`).

Vertical slice (4 marks): a single feature (say “add a Track to an Album”) touches all four folders for that one feature — you add or modify the entity in `domain`, add a query in `repository`, add the orchestration in `service`, and add the HTTP handler in `web/rest`. You ship one complete user-visible feature at a time, rather than completing all repositories first and all controllers second.

Question 3 — WP Practice (40 marks)

3(a) — Conference JDL with ManyToMany [15 marks]

```
entity Conference {
    name String required,
    startDate LocalDate required,
    endDate LocalDate required,
    website String,
    summary TextBlob
}

entity Speaker {
    firstName String required,
    lastName String required,
    bio TextBlob
}

relationship ManyToMany {
    Conference{speakers} to Speaker{conferences}
}

relationship ManyToOne {
    Conference{organiser} to User with builtInEntity
}

paginate Conference, Speaker with pagination
service Conference with serviceClass
```

Marking: Conference entity (3), Speaker entity (3), ManyToMany with navigation on both sides (3), ManyToOne Conference→User with builtInEntity (2), `paginate` directive (2), `service ... with serviceClass` (2).

3(b)(i) — POST cURL [5 marks]

```
curl -X POST http://localhost:8080/api/v1/conferences \
     -H "Content-Type: application/json" \
     -H "Authorization: Bearer <TOKEN>" \
     -d '{
           "name": "Spring Connect 2026",
           "startDate": "2026-09-12",
           "endDate": "2026-09-14"
         }'
```

Marking: `-X POST` (1), correct collection URL (plural, no id) (1), JSON Content-Type (1), Authorization Bearer header (1), JSON body with required fields (1).

3(b)(ii) — 405 explanation [5 marks]

405 Method Not Allowed means the resource exists at that URL but the HTTP method you used is not implemented for it (2). Most likely cause: the backend has not defined a `@DeleteMapping` on the `/conferences/{id}` endpoint (or it has been disabled via `readOnly = true` on the resource) (2). The server should also return an `Allow` header listing the methods it does support (1).

3(c)(i) — Derived query method [5 marks]

```
List<Conference> findByStartDateBetweenOrderByStartDateAsc(
    LocalDate from, LocalDate to);
```

Marking: `findBy` prefix (1), `Between` keyword for date range (2), `OrderByStartDateAsc` suffix (1), correct return type and parameter types (1). Caller passes Jan 1 and Dec 31 of the year.

3(c)(ii) — JPQL @Query [5 marks]

```
@Query("SELECT c FROM Conference c "
     + "WHERE YEAR(c.startDate) = :year "
     + "ORDER BY c.startDate ASC")
List<Conference> findInYear(@Param("year") int year);
```

Marking: `@Query` annotation (1), correct JPQL syntax using entity name not table name (1), `@Param` binding (1), ORDER BY clause (1), explanation: prefer JPQL when the derived-name approach becomes too long, uses functions like `YEAR()`/joins/aggregates that don't map cleanly to method names, or when you need readable named parameters (1).

3(d) — Liquibase changelog [5 marks]

Liquibase treats schema as code: you describe the change in a `changeSet` (in XML/YAML/JSON), e.g. an `<addColumn>` on the Conference table. Each changeSet has a unique `(id, author)` tuple that Liquibase records in a `DATABASECHANGELOG` table the first time it runs (2). The `master.xml` changelog lists every per-feature change file in the order they should be applied (1). At startup, Liquibase looks at master.xml, compares against `DATABASECHANGELOG`, and applies only the new changeSets — existing data is untouched and the operation is idempotent across restarts (2).

— End of Mark Scheme —