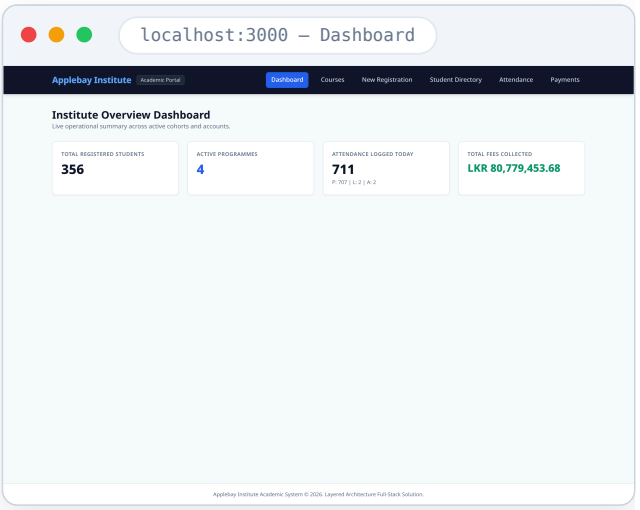
APPLEBAY INSTITUTE OF HIGHER EDUCATION

DEPARTMENT OF COMPUTING & INFORMATION SYSTEMS

Academic, Attendance & Fee Management System

Technical Documentation & System Architecture Report — Full-Stack Web System

MODULE	Full-Stack Web Application Development
ASSIGNMENT	Individual Full-Stack Web System (Tasks 1–7)
CANDIDATE NAME	Pathirannehelage Ashan Dhanushka
CANDIDATE ID	WCM24B126
SUBMISSION DATE	October 2026
REPOSITORY	github.com/wcm24b126-hue/applebay-backend
DATABASE	applebay — PostgreSQL 16



Next.js 14 · React | Spring Boot 3.3.4 · Java 21 | Spring Data JPA · Hibernate | PostgreSQL 16 | REST · JSON

Layered architecture · Controller → Service → Repository

355+ students · 14 functional requirements · 100% API coverage

Contents

1 Introduction & Problem Statement 3

2 Stakeholders, Users & Assumptions 4

3 Requirements Specification 5

3.1 Functional requirements · 3.2 Non-functional requirements · 3.3 Traceability

4 System Architecture & Layered Structure 7

4.1 Three-tier overview · 4.2 Layer responsibilities · 4.3 Request lifecycle

5 Database Design & Relational Schema (3NF) 9

5.1 Entity-relationship model · 5.2 Tables & constraints · 5.3 Design decisions

6 Data Flow & Transaction Analysis 11

6.1 Atomic registration · 6.2 Bulk attendance & correction · 6.3 Payment guard

7 Frontend Implementation & Integration 13

8 REST API Specification 22

9 Error Handling Approach 24

10 Testing & Quality Verification 25

11 Installation & Running Guide 27

12 Limitations, Future Work & Conclusion 28

A Appendix A — Service-layer unit tests 29

B Appendix B — Reporting SQL & References 30

Figures

Fig 4.1 — Three-tier architecture

Fig 4.2 — End-to-end request lifecycle

Fig 5.1 — Entity-relationship diagram

Fig 6.1 — Atomic registration sequence

Fig 6.2 — Bulk attendance & correction

Fig 6.3 — Payment recording decision flow

Fig 7.1 — Page → API integration map

Fig 7.2–7.9 — Interface screenshots

1. Introduction & Problem Statement

Applebay Institute is a private education provider delivering Higher National Diploma (HND) programmes and short Certificate courses to between 350 and 400 students at any time, spread across multiple courses and batches. Student information, attendance and fee payments were historically handled manually: paper registration forms, printed attendance sheets and a spreadsheet ledger maintained by the cashier.

This manual process produced four recurring operational problems that the developed system directly addresses:

Table 1.1 — Business problems and the engineered response

Problem (as reported)	Business impact	System response
Duplicate student records, no unique number	Students double-counted; reports unreliable	Automatic sequential student numbers (`AB2026-0001`) plus unique constraints on NIC and email
Attendance sheets lost; percentage computed by hand	Eligibility audits take days per cohort	Bulk class-sheet capture stored per session; percentage and 80% eligibility computed live
No single view of paid vs. outstanding fees	Staff cannot answer "how much is owed?" quickly	Dynamic balance derived from payment history; outstanding shown per enrolment
Overpayments and duplicate payment entry	Ledger discrepancies and reconciliation effort	Server-side balance check rejects overpayments; pessimistic locking serialises concurrent payments

> **Aim**
> Deliver a working full-stack system with a correct, well-structured data flow and a strictly layered Spring Boot backend. The user interface is intentionally simple and clean; the assessment weight sits on how data moves between the browser, the REST API and PostgreSQL, and on the separation of Controller / Service / Repository responsibilities.

1.1 Objectives

- Model the domain (courses, students, enrolments, payments, attendance) as a normalised relational schema with suitable keys and constraints.
- Expose the domain through a REST API implemented with a layered Spring Boot backend and separate request/response DTOs.
- Enforce all business rules (duplicate prevention, balance checks, exam eligibility) inside the service layer within database transactions.
- Provide a Next.js frontend that consumes the API and handles loading, success and error states.
- Verify the system with automated service tests and a documented case-by-case API test log.

1.2 Scope

In scope: course management, student registration and enrolment, attendance capture/correction/eligibility, fee instalments with receipting and balance control, and an operational dashboard. Out of scope (per the brief): authentication and role-based access control, payment-gateway integration, and deployment infrastructure. These are discussed under *Limitations & Future Work* (§12).

2. Stakeholders, Users & Assumptions

2.1 Users of the system

Table 2.1 — User roles and their use of the system

User	Primary activities	Pages / features used
Registrar / academic clerk	Creates courses, registers students, enrols students to a course and batch, records the first payment	Courses, New Registration, Student Directory
Lecturer	Marks a class attendance sheet and corrects it; checks exam eligibility	Attendance
Cashier	Records instalments; looks up a student account and prints/reviews receipts	Payments
Management	Monitors totals: students, active courses, today's attendance, fees collected	Dashboard

2.2 Assumptions

- The system is used by trusted institute staff on an internal network; authentication/RBAC is not required by the brief, so it is not implemented (documented as future work).
- A student is uniquely identified by their NIC and email; both must be unique across the institute.
- A student may enrol in more than one course, but never twice in the *same course and batch*.
- All monetary values are in Sri Lankan Rupees (LKR) with two decimal places; course fees are fixed per course at enrolment time.
- Attendance is recorded per enrolment per calendar date; a late arrival is counted as attended for eligibility purposes.
- An instalment may not exceed the outstanding balance, and a fully settled enrolment cannot accept further payments.

3. Requirements Specification

3.1 Functional requirements (FR01–FR14)

Table 3.1 — Functional requirements and where each is implemented

ID	Module	Requirement	Implementation (backend)
FR01	Courses	Add, view, update and deactivate courses (code, name, level, duration, fee)	`CourseController` / `CourseService` (POST, GET, PUT, PATCH deactivate/reactivate)
FR02	Registration	Register a student with name, NIC, email, phone, address, DOB	`StudentService.register` + `StudentRequest` validation
FR03	Registration	Auto-generate a unique student number (AB2026-0001)	`StudentService` using sequence `student_number_seq`
FR04	Registration	Prevent duplicates via NIC and email	`StudentRepository.existsByNic/Email` + unique DB constraints
FR05	Enrolment	Enrol to course + batch; block duplicates	`EnrollmentService.enroll` + `uk_enrollment_student_course_batch`
FR06	Students	Search by name / number / NIC with pagination	`StudentService.search` → `PageResponse`
FR07	Attendance	Mark a whole class (Present/Late/Absent) for course+date	`AttendanceService.markClassAttendance` (single bulk request)
FR08	Attendance	Correct a day without creating duplicate rows	Existing rows updated in place; `uk_attendance_enrollment_date`
FR09	Attendance	Attendance % and 80% exam eligibility	`AttendanceService.eligibility` (JPQL aggregate)
FR10	Payments	Record instalments (amount, date, method)	`PaymentService.record`
FR11	Payments	Unique receipt number per payment	`receipt_number_seq` → `RC2026-000001`
FR12	Payments	Show fee/paid/outstanding; reject overpayment	`PaymentService` + `BalanceResponse`
FR13	Payments	Payment history per enrolment	`PaymentService.history`
FR14	Dashboard	Totals: students, active courses, today's attendance, fees collected	`DashboardService.summary` + `DashboardResponse`

3.2 Non-functional requirements

Table 3.2 — Non-functional requirements and how they are satisfied

Area	Requirement	Approach
Performance / scale	Handle at least 400 students; large lists paginated server-side	`PageRequest` / `PageResponse`; JPQL aggregates; `join fetch` to avoid N+1
Data integrity	Primary keys, foreign keys, unique and check constraints; transactions	Schema-level PK/FK/UK/CHECK; `@Transactional` service boundaries
Validation	All input validated on the backend; consistent error format	Bean Validation on DTO records; `GlobalExceptionHandler` returns one JSON shape
Maintainability	Clear package structure with meaningful names	Packages `controller`, `service`, `repository`, `entity`, `dto`, `exception`, `config`

Area	Requirement	Approach
Usability	Simple, clean interface is sufficient	Consistent Tailwind-based pages with clear success/error banners

3.3 Traceability summary

All fourteen functional requirements are implemented and reachable through the REST API. The table below summarises the status; every row was re-verified against the running system (see §10 for the test log).

FR01–FR05 Verified **FR06–FR09 Verified** **FR10–FR14 Verified**

4. System Architecture & Layered Structure

The system is a decoupled client–server application. A Next.js single-page frontend talks to a Spring Boot REST API over HTTP using JSON; the API persists to PostgreSQL through Spring Data JPA. The backend is deliberately split into six packages, and the request always flows *Controller → Service → Repository* — a controller never touches the database directly and an entity is never returned to the client.

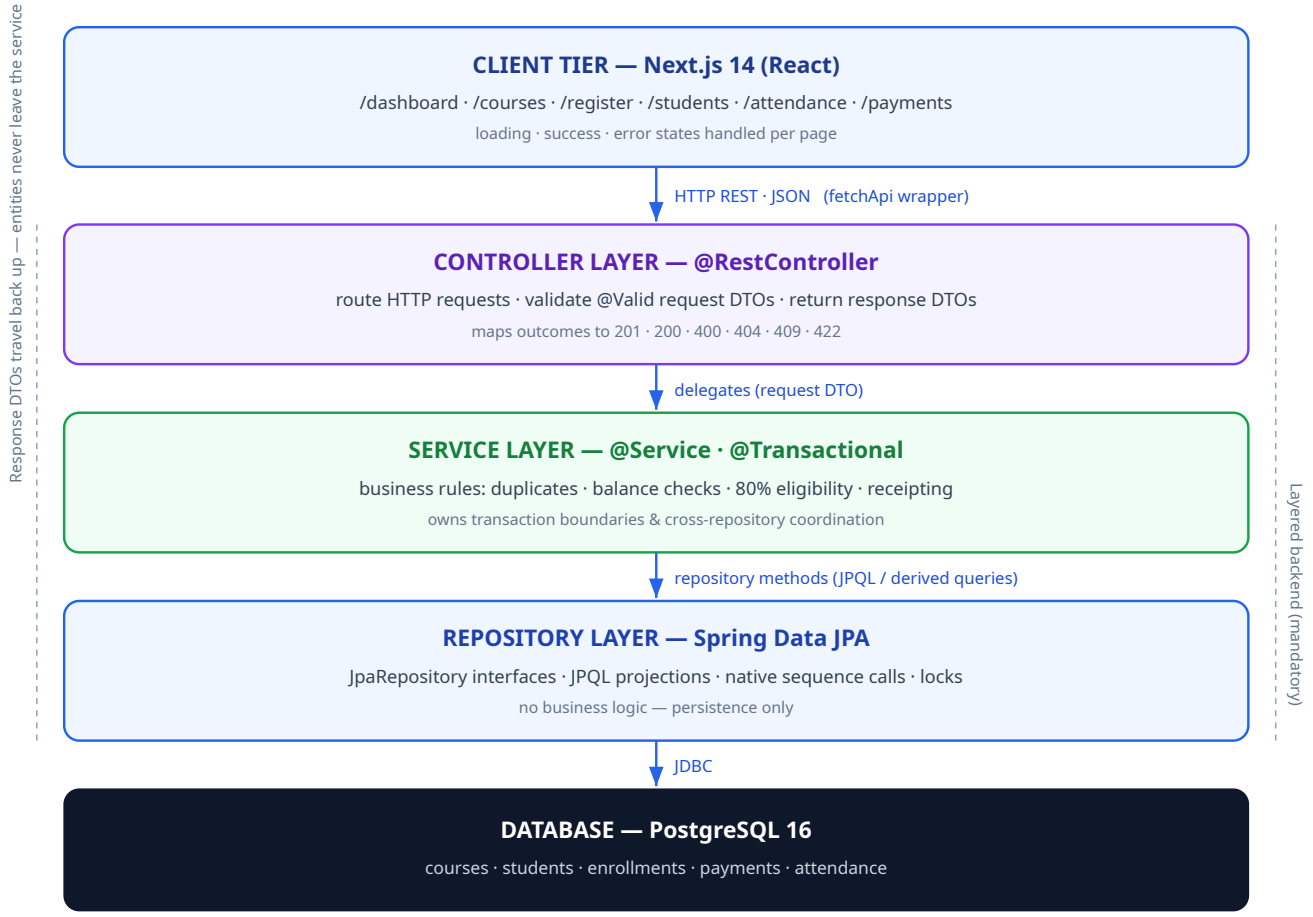


Figure 4.1 — Three-tier architecture and the mandatory backend layers.

4.1 Backend layer responsibilities

controller Receives HTTP requests, runs Bean Validation on request DTOs, delegates to a service and returns a response DTO with the correct status code. Controllers contain no business logic and no database access. e.g. `RegistrationController`, `PaymentController`

service Holds all business rules and transaction boundaries; coordinates multiple repositories. e.g. `PaymentService`, `AttendanceService`, `RegistrationService`

repository Persistence only, through Spring Data JPA interfaces, derived queries, JPQL projections and native sequence calls. e.g. `EnrollmentRepository`, `PaymentRepository`

entity JPA classes mapped to tables, including relationships and keys. Never serialised to the client.

dto Java `record` types separating request payloads from response payloads. e.g. `PaymentRequest` → `PaymentResponse`

exception Custom exceptions plus a `@RestControllerAdvice` that renders a single, consistent JSON error body.

4.2 End-to-end request lifecycle

Every API call follows the same path. The numbered flow below traces a write request (for example recording a payment) from the browser down to PostgreSQL and back:

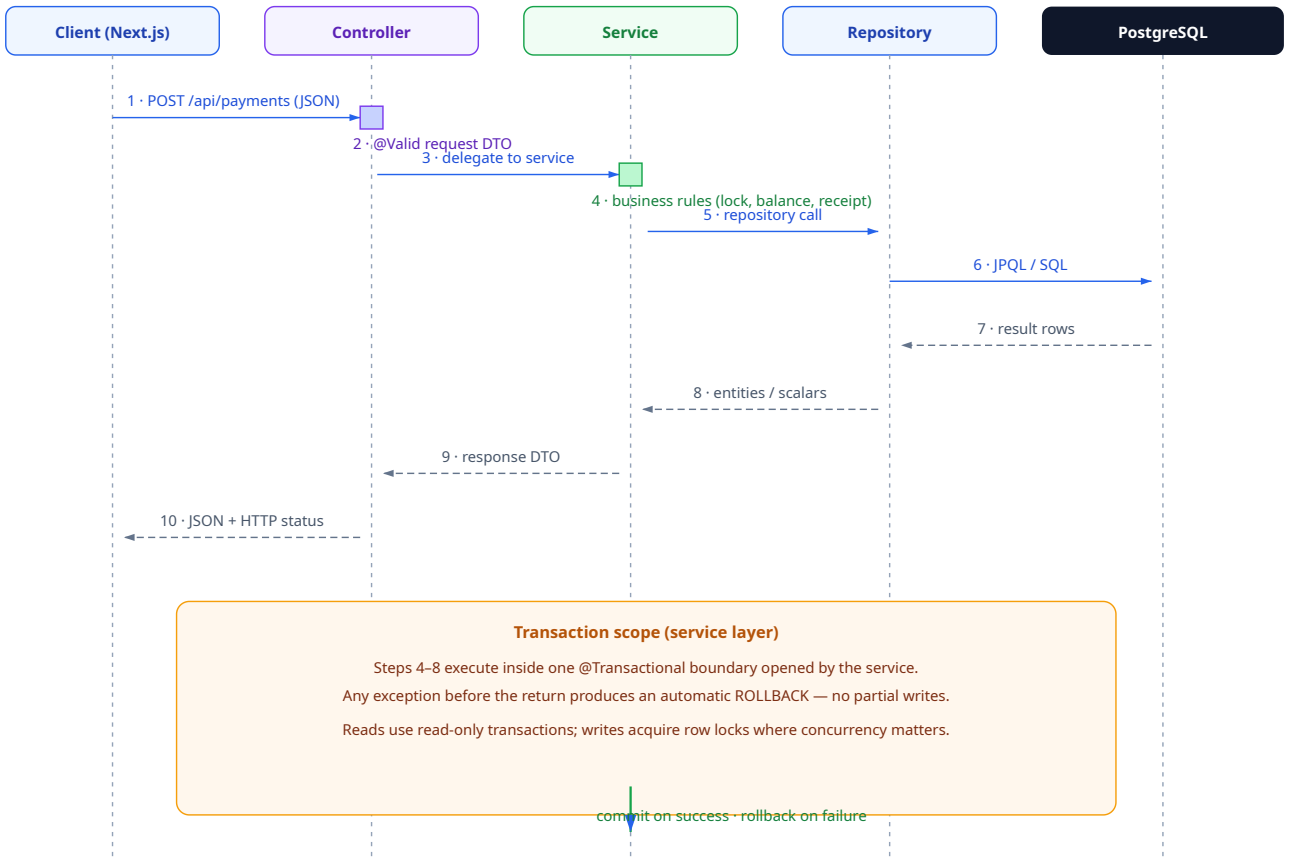


Figure 4.2 — End-to-end request lifecycle through the layered backend.

4.3 Key architectural decisions

- **DTO records in and out.** Request and response types are separate Java records; JPA entities are never serialised, so the API contract is decoupled from the schema.
- **Constructor injection.** Dependencies are supplied through Lombok `@RequiredArgsConstructor`, giving immutable, testable services.
- **Transactions at the service boundary.** Services are annotated `@Transactional` (class-level read-only, method-level for writes), so the database commit point is explicit.
- **One consistent error shape.** A single `@RestControllerAdvice` converts every handled exception into the same JSON structure.
- **Efficient reads.** Listing queries use `join fetch` and JPQL projections, and student search is paginated in the database rather than in memory.

5. Database Design & Relational Schema (3NF)

The schema is normalised to Third Normal Form. There are five tables: three master/transaction tables (`courses` , `students` , `payments` , `attendance`) plus `enrollments` , which resolves the many-to-many relationship between students and courses and becomes the anchor for payments and attendance.

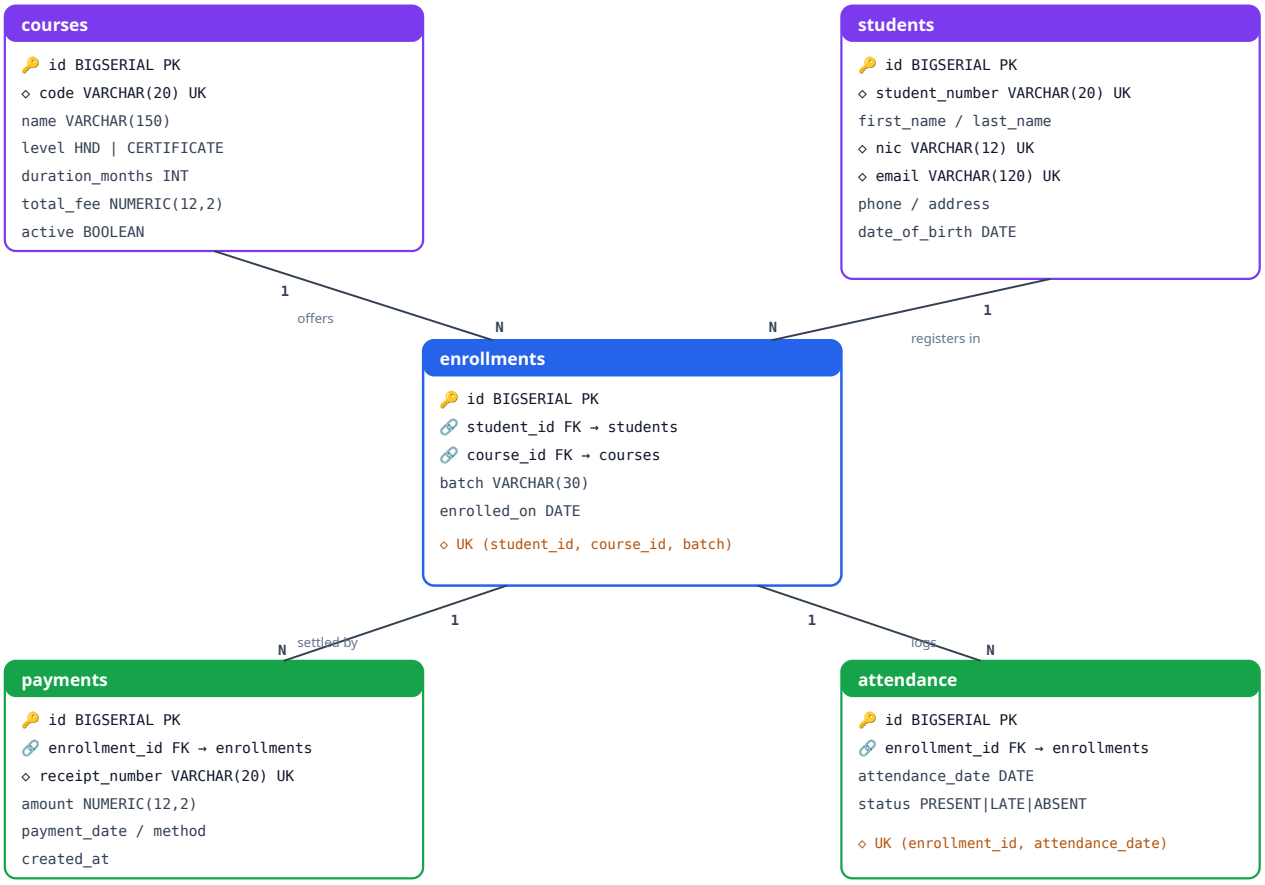


Figure 5.1 — Entity-relationship diagram (🔑 primary key, 🔗 foreign key, ◇ unique key).

5.1 Table definitions and constraints

Table 5.1 — Physical schema summary

Table	Key columns	Constraints
`courses`	id (PK), code (UK)	`CHECK` level ∈ {HND, CERTIFICATE}; duration > 0; total_fee ≥ 0
`students`	id (PK), student_number (UK), nic (UK), email (UK)	`CHECK` date_of_birth < CURRENT_DATE; all contact fields NOT NULL
`enrollments`	id (PK), student_id (FK), course_id (FK)	`UNIQUE` (student_id, course_id, batch) → FR05; FK `ON DELETE RESTRICT`
`payments`	id (PK), enrollment_id (FK), receipt_number (UK)	`CHECK` amount > 0; method ∈ {CASH, CARD, BANK_TRANSFER, ONLINE}
`attendance`	id (PK), enrollment_id (FK)	`UNIQUE` (enrollment_id, attendance_date) → FR08; status ∈ {PRESENT, LATE, ABSENT}

Two database sequences back the human-readable identifiers: `student_number_seq` (→ AB2026-0001) and `receipt_number_seq` (→ RC2026-000001).

5.2 Normalisation and design decisions

- **1NF / 2NF.** Every column holds a single atomic value and every non-key column depends on the whole primary key — the only surrogate keys are `id` columns, and there are no repeating groups.
- **3NF.** No non-key attribute depends transitively on another non-key attribute. Course details live only in `courses`, student details only in `students`, and the enrolment row stores only the batch and date — never a copy of the student or course name.
- **Bridge table.** `enrollments` resolves the many-to-many student↔course relationship and becomes the parent of payments and attendance, so a student's fee ledger and attendance are always scoped to a specific course and batch.
- **Derived, not stored, balance.** The outstanding fee is computed as `course.total_fee − SUM(payments.amount)`. There is no mutable balance column, which removes a whole class of update-anomalies and keeps FR12 mathematically consistent.
- **Referential actions.** Payments and enrolments use `ON DELETE RESTRICT` so financial history cannot be silently lost; attendance uses `ON DELETE CASCADE` because a session log has no meaning without its enrolment.
- **Enforced at the database.** Duplicate NIC/email, duplicate course codes and duplicate enrolment are guaranteed by unique constraints — the service-layer checks give friendly messages, but the database is the final arbiter under concurrency.

6. Data Flow & Transaction Analysis

6.1 Atomic registration (FR02, FR03, FR04, FR05, FR10, FR11)

A single `POST /api/registrations` creates three dependent records. The registrar supplies the student details, a course, a batch and the first payment; `RegistrationService` orchestrates the three sub-services inside one transaction. If *any* stage fails — duplicate NIC, inactive course, or an overpayment — the entire transaction rolls back and no orphan student or enrolment is stored.

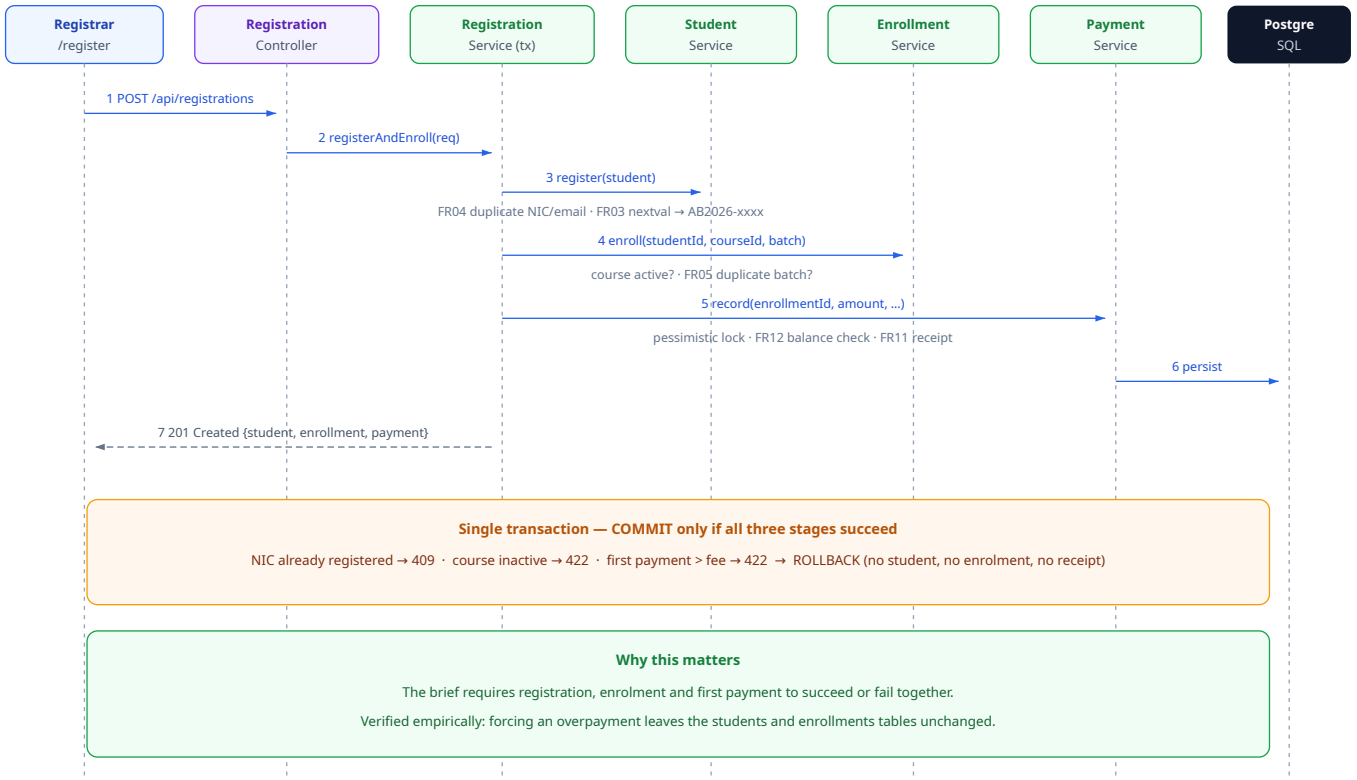


Figure 6.1 — Atomic registration sequence with all-or-nothing rollback.

6.2 Bulk attendance recording and in-place correction (FR07, FR08, FR09)

Attendance for an entire class is submitted in one request rather than one request per student. The service diffs the submitted statuses against any rows already stored for that date and produces accurate created / updated / unchanged counters — correcting a day updates rows in place instead of inserting duplicates.

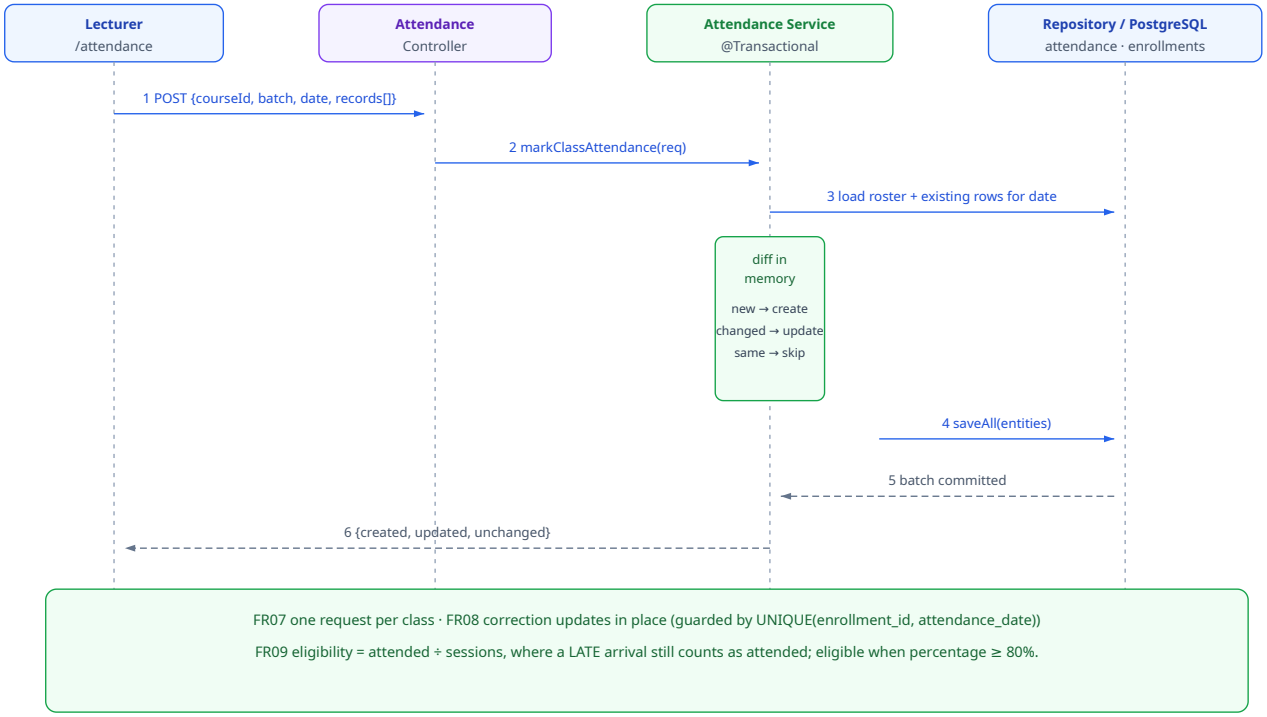


Figure 6.2 — Bulk attendance submission and in-place correction.

6.3 Payment recording and the overpayment guard (FR10, FR11, FR12)

The outstanding balance is never stored; it is computed from the payment history. Before a payment is saved, the service acquires a `PESSIMISTIC_WRITE` lock on the enrolment row so two cashiers cannot both pass the balance check at the same time.

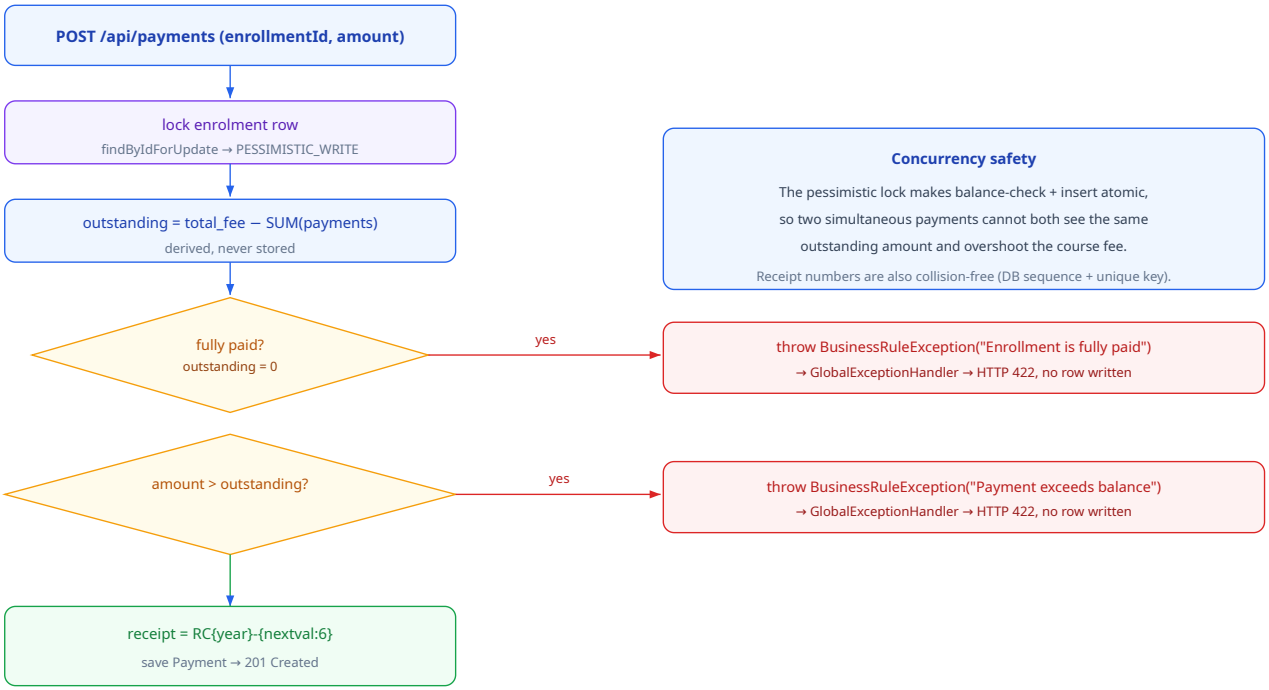
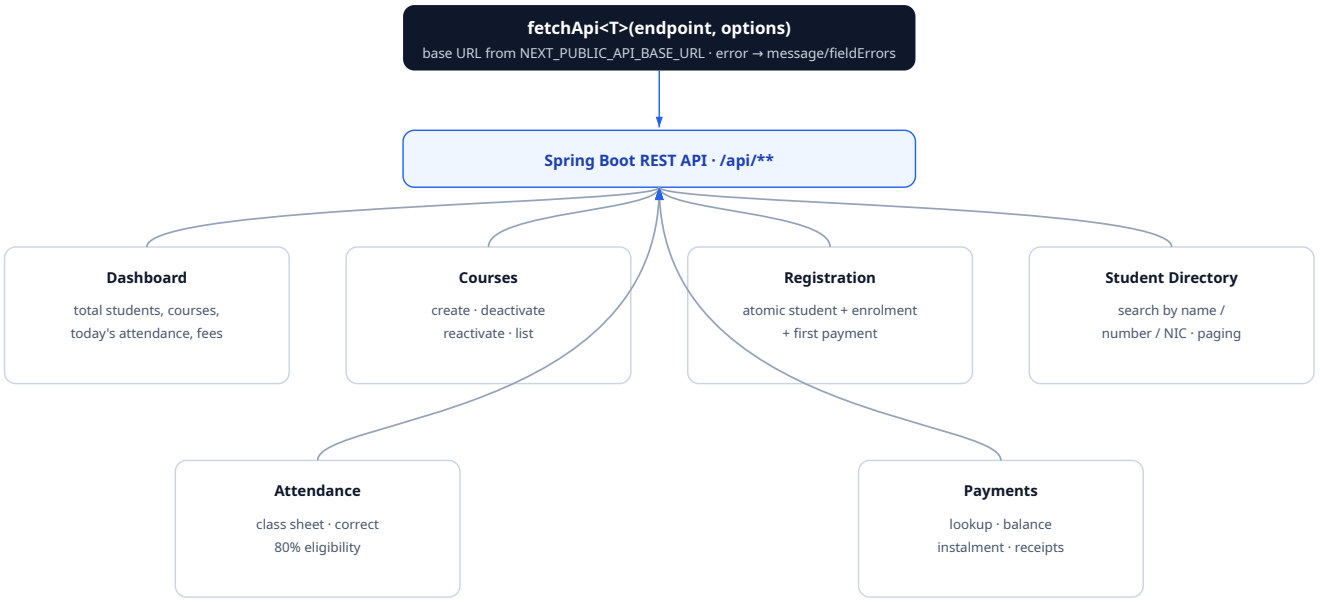


Figure 6.3 — Payment decision flow with the two rejection branches and the concurrency lock.

7. Frontend Implementation & Integration

The frontend is a Next.js 14 App-Router application written in TypeScript with Tailwind CSS. A small fetch wrapper (`src/lib/api.ts`) centralises the base URL, JSON headers and — importantly — error translation: any non-2xx response is parsed and the backend's `message` / `fieldErrors` are surfaced to the user.



Every page consumes the single fetchApi wrapper; loading, success and error states are rendered per page.

No page imports an entity or writes SQL — the API is the only integration point.

Figure 7.1 — Page-to-API integration map.

7.1 Interface screenshots

The following captures were taken from the running system (Next.js on `localhost:3000` , Spring Boot on `localhost:8080/api`) against the seeded dataset of 355+ students.

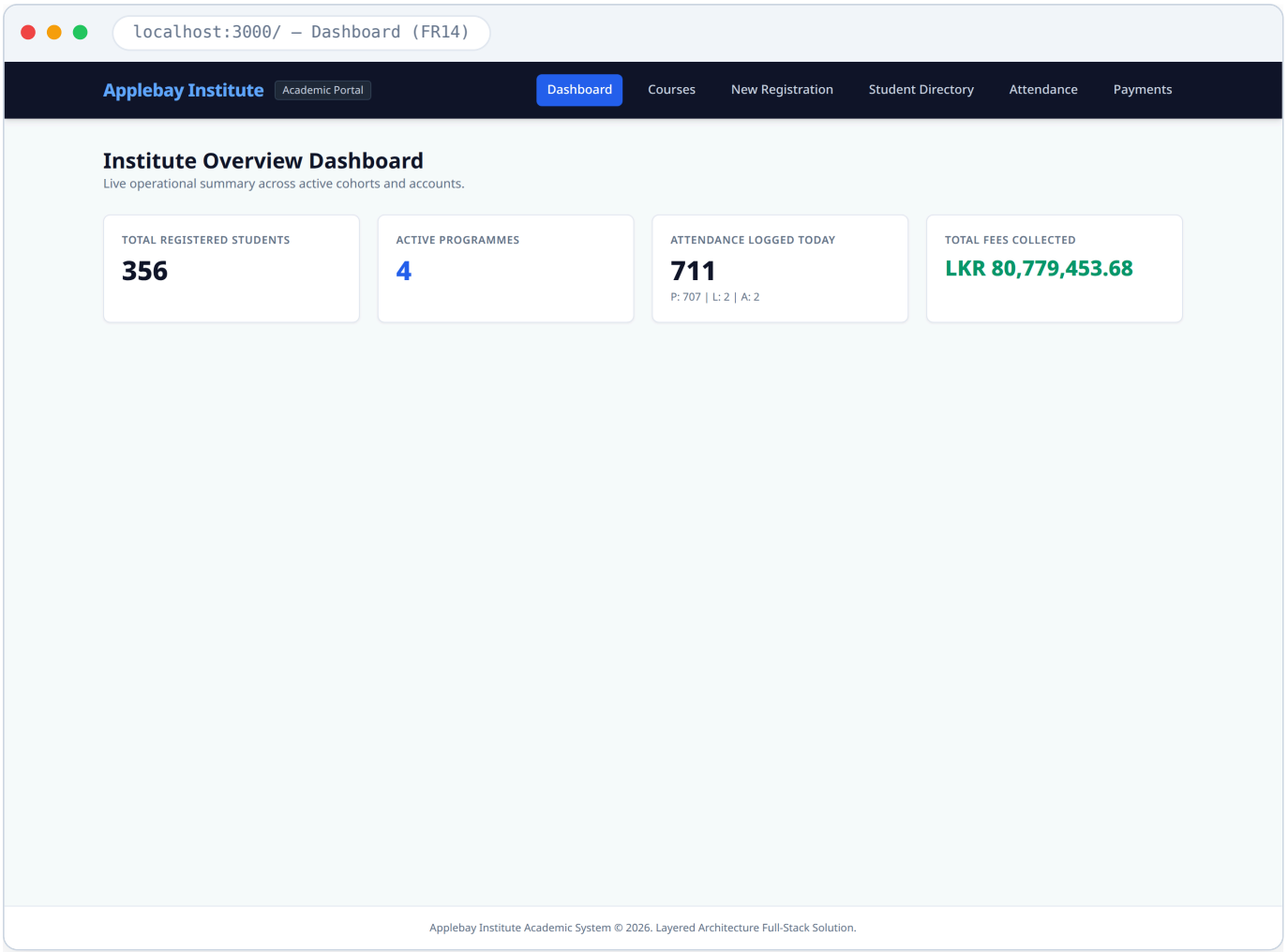


Figure 7.2 — Operational dashboard: registered students, active programmes, today's attendance and total fees collected.

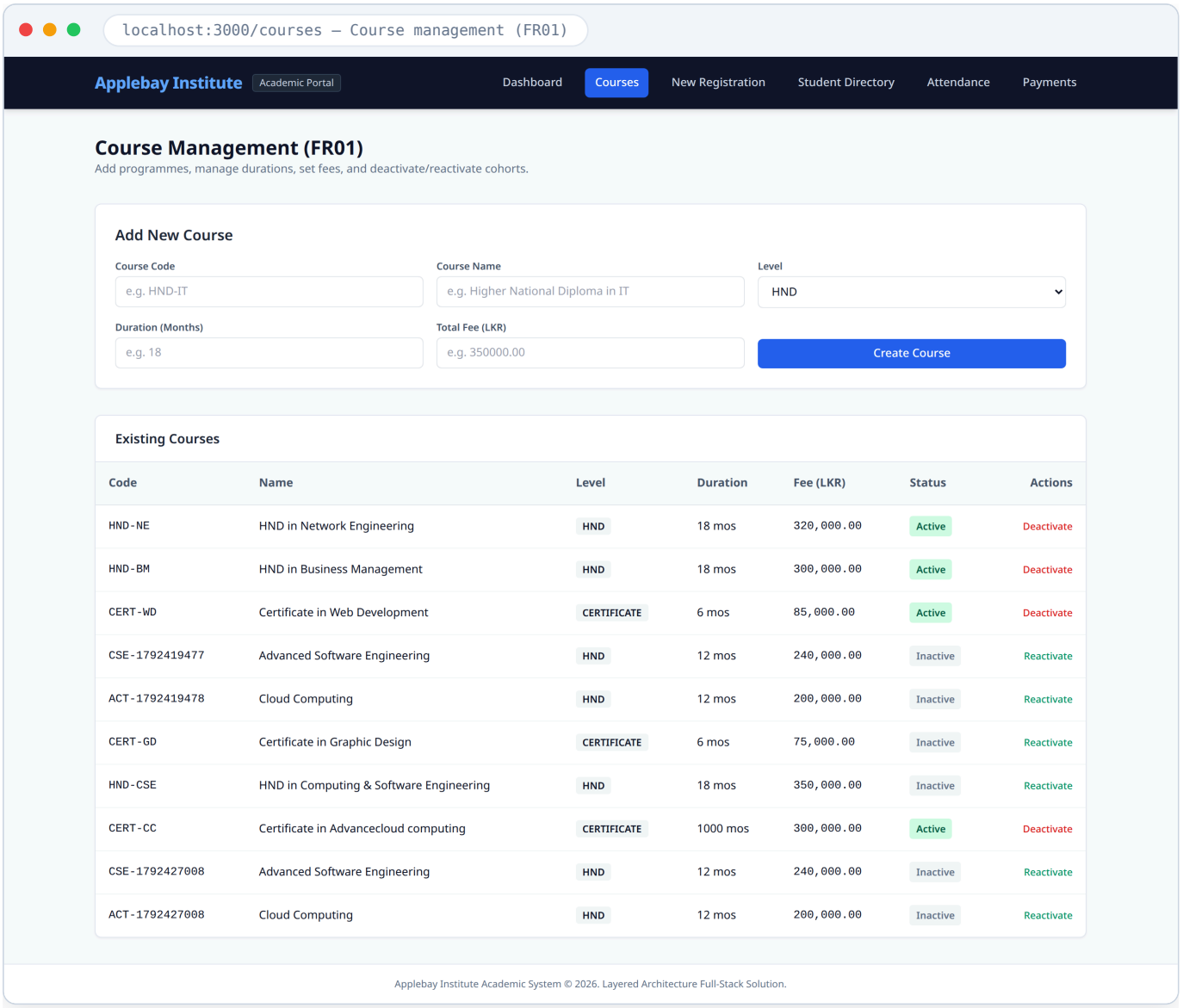


Figure 7.3 — Course management: create a course and deactivate / reactivate existing ones; the fee and level control downstream fees.

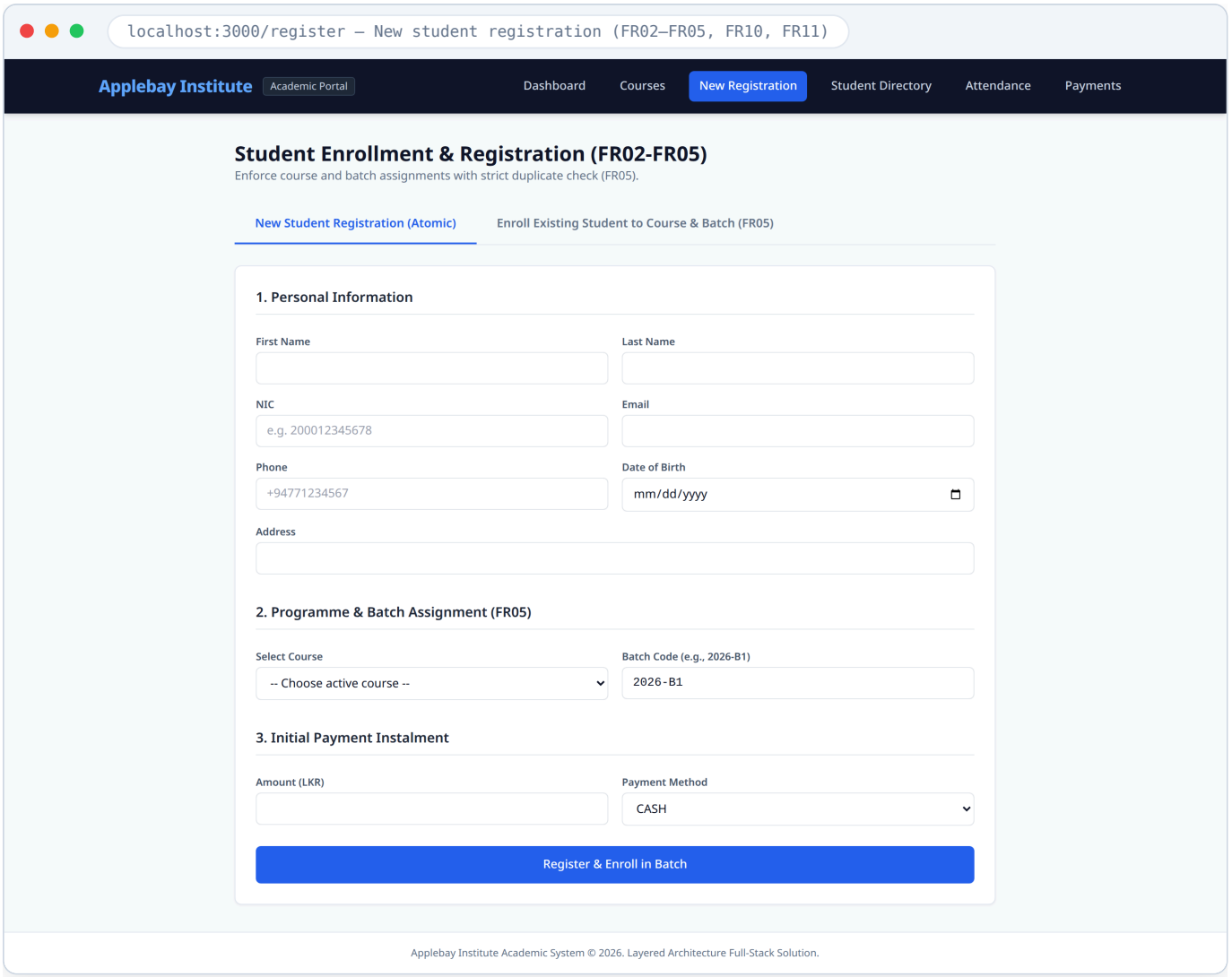


Figure 7.4 — Atomic registration form: personal details, programme/batch assignment and the first instalment are submitted as one transaction.

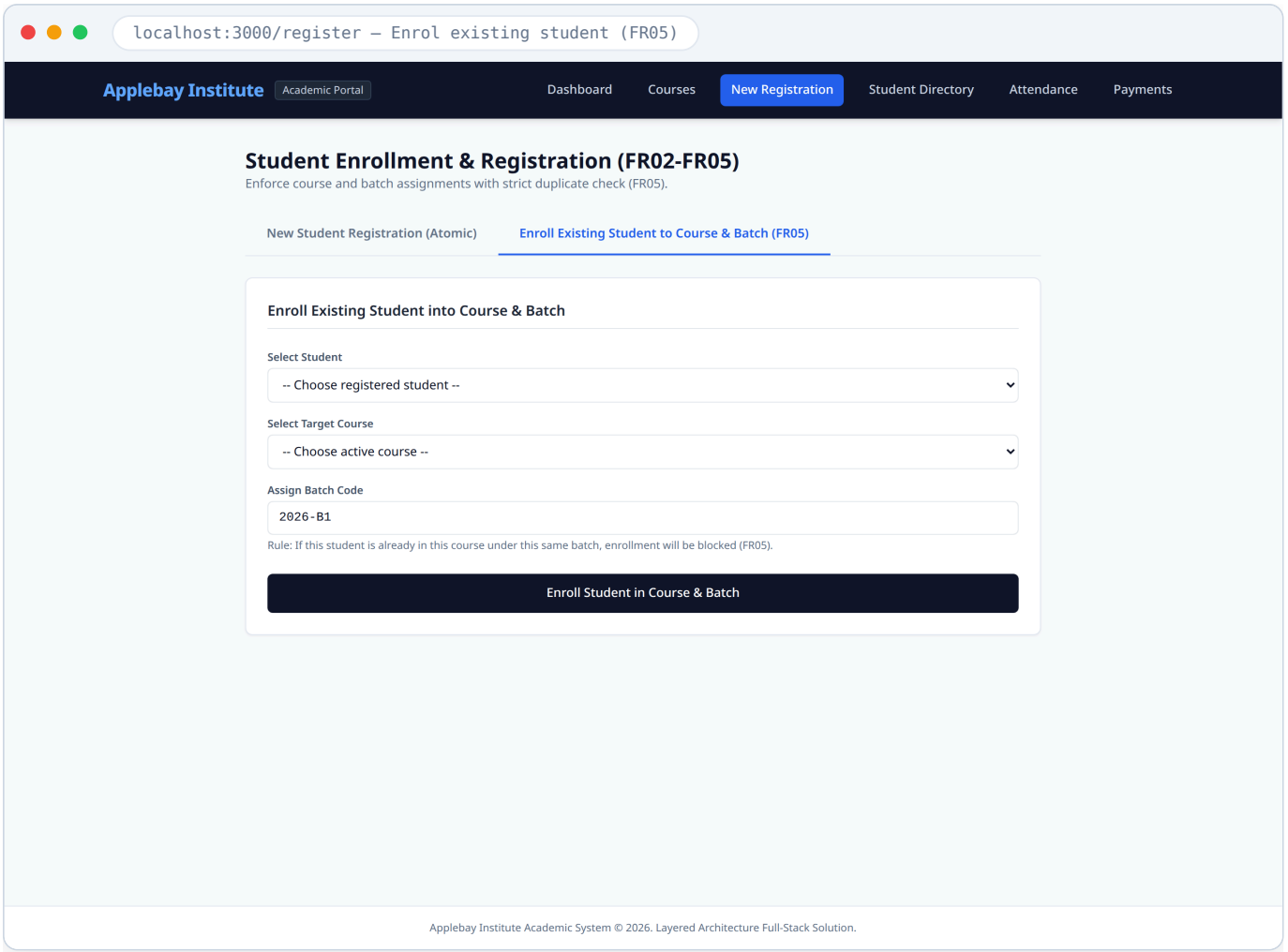


Figure 7.5 — Enrolling an already-registered student into another course/batch; duplicate course+batch combinations are blocked by the API.

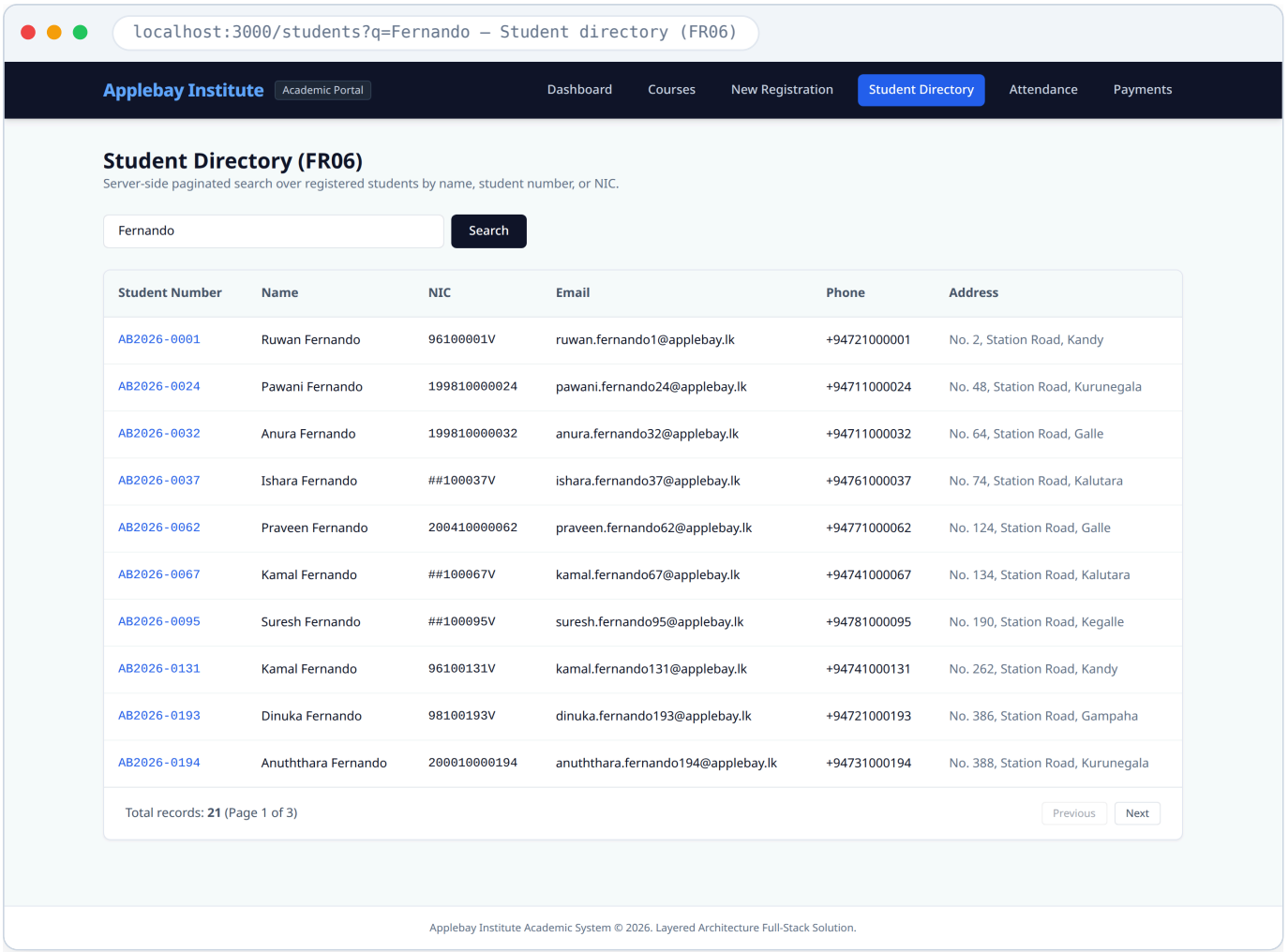


Figure 7.6 — Server-side paginated search across name, student number and NIC.

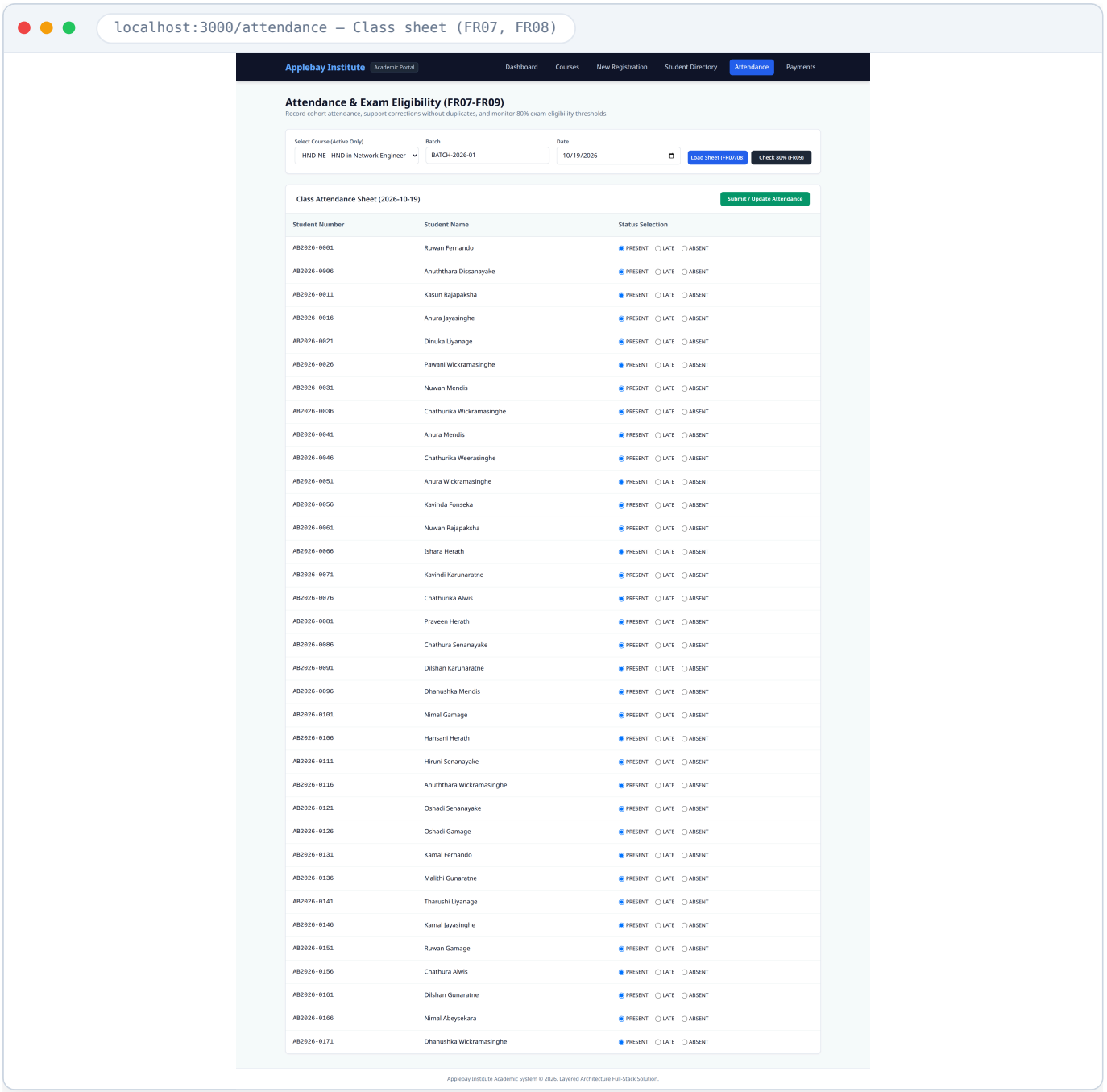


Figure 7.7 — Class attendance sheet for a course and date; the whole cohort is saved in a single request and can be corrected later.

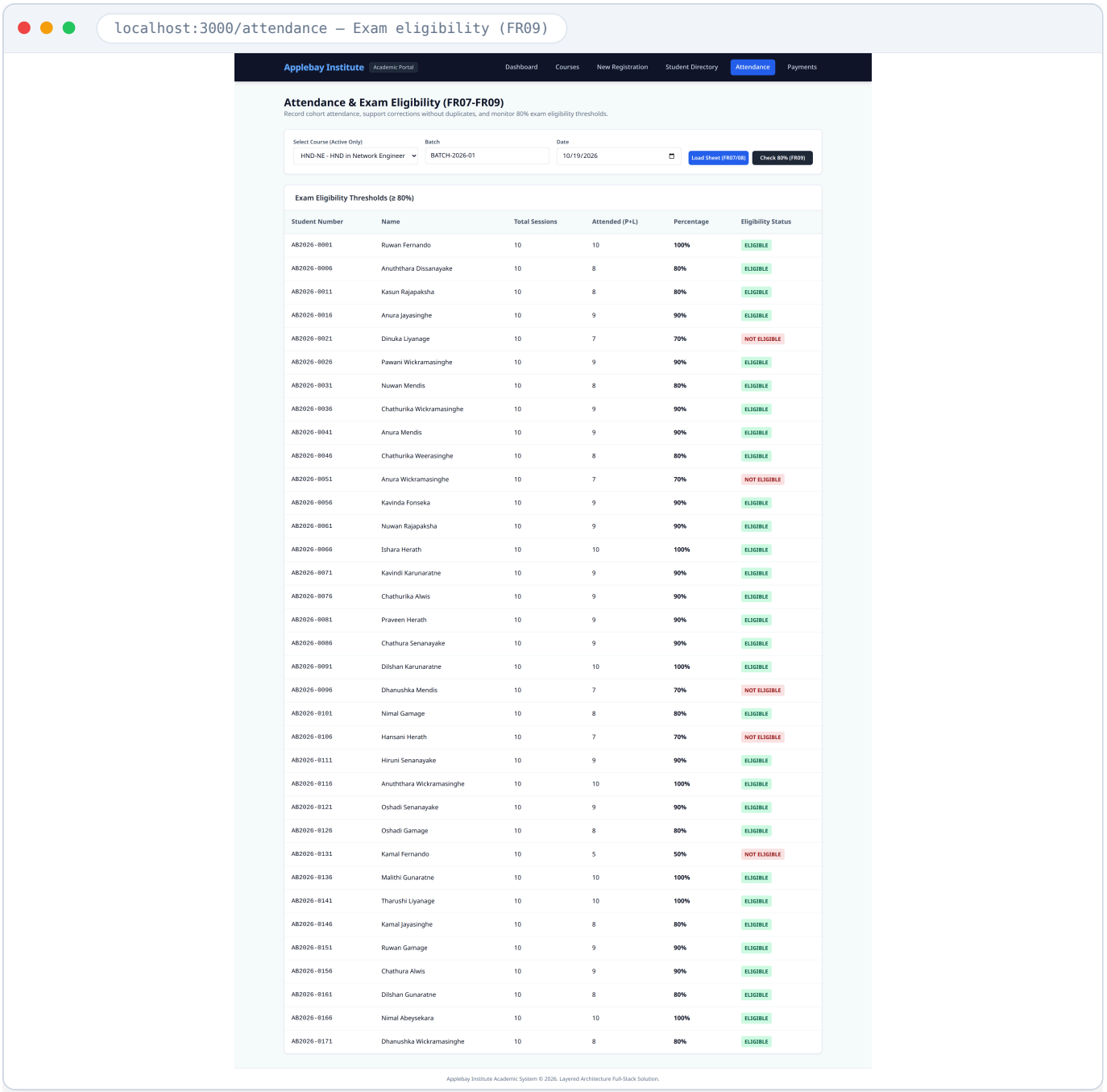


Figure 7.8 — Live attendance percentage with the 80% exam-eligibility decision per student.

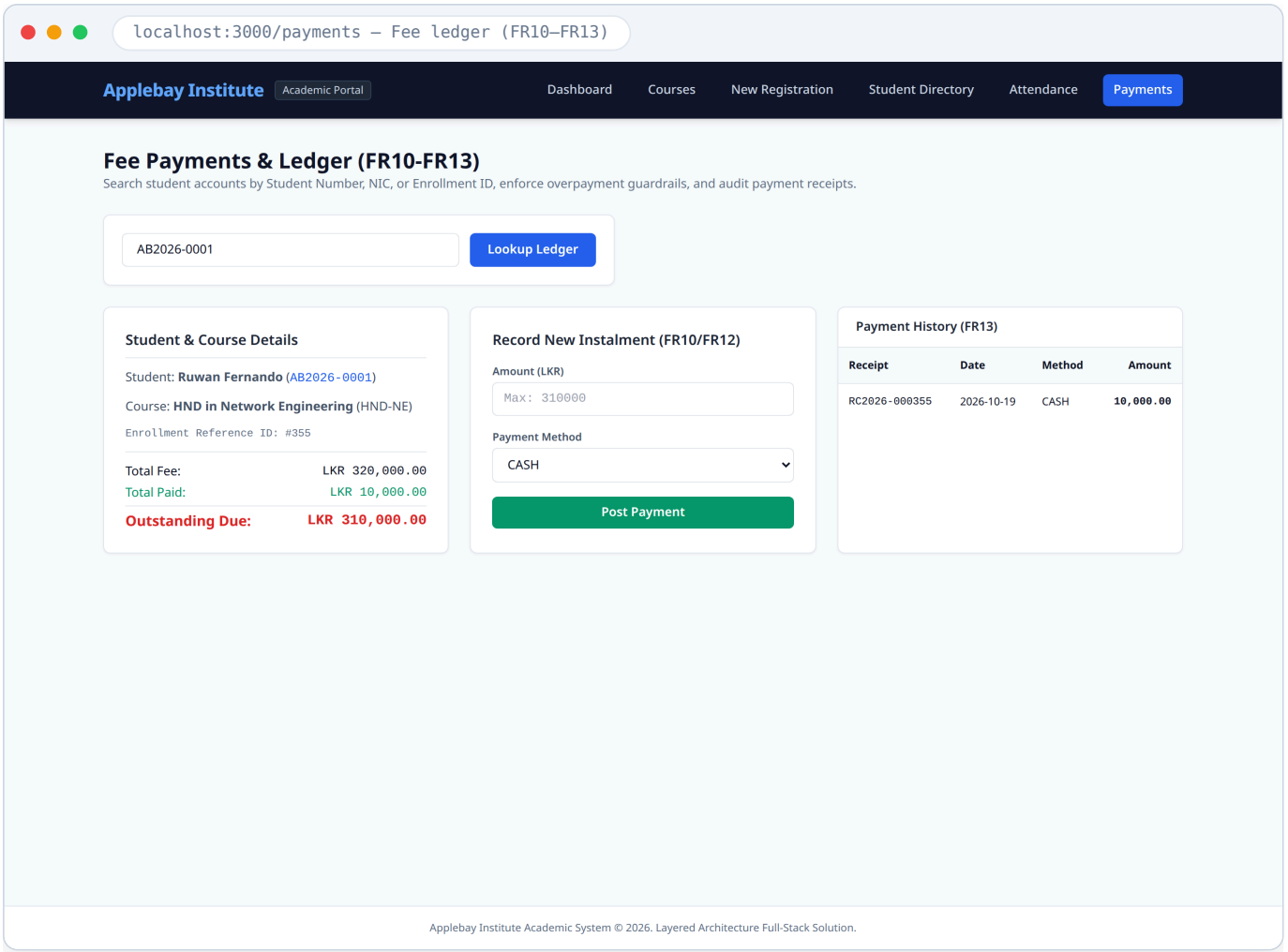


Figure 7.9 — Fee ledger lookup by student number/NIC/ID: fee, total paid, outstanding balance, instalment entry and receipt history.

8. REST API Specification

All endpoints are rooted at `/api` and exchange JSON. Request bodies are validated before reaching the service layer.

Table 8.1 — Complete endpoint list

Method	Endpoint	Request / params	Response & status
GET	`/api/dashboard`	—	`DashboardResponse` · 200
GET	`/api/courses`	—	`List<CourseResponse>` · 200
GET	`/api/courses/{id}`	path id	`CourseResponse` · 200 / 404
POST	`/api/courses`	`CourseRequest`	`CourseResponse` · 201 / 400 / 409
PUT	`/api/courses/{id}`	`CourseRequest`	`CourseResponse` · 200 / 404 / 409
PATCH	`/api/courses/{id}/deactivate`	—	`CourseResponse` (active=false) · 200
PATCH	`/api/courses/{id}/reactivate`	—	`CourseResponse` (active=true) · 200
POST	`/api/registrations`	`RegistrationRequest`	`RegistrationResponse` · 201 / 400 / 409 / 422
GET	`/api/students`	`q, page, size`	`PageResponse<StudentResponse>` · 200
POST	`/api/students`	`StudentRequest`	`StudentResponse` · 201 / 400 / 409
GET	`/api/students/{id}`	path id	`StudentResponse` · 200 / 404
GET	`/api/students/{id}/enrollments`	path id	`List<EnrollmentResponse>` · 200
POST	`/api/enrollments`	`EnrollmentRequest`	`EnrollmentResponse` · 201 / 404 / 409 / 422
GET	`/api/attendance/class-sheet`	`courseId, batch, date`	`List<ClassSheetRow>` · 200 / 404
GET	`/api/attendance/eligibility`	`courseId, batch`	`List<EligibilityRow>` · 200 / 404
POST	`/api/attendance`	`MarkAttendanceRequest`	`AttendanceMarkResponse` · 200 / 400 / 422
POST	`/api/payments`	`PaymentRequest`	`PaymentResponse` · 201 / 400 / 404 / 422
GET	`/api/enrollments/lookup`	`query` (number/NIC/ID)	`BalanceResponse` · 200 / 404
GET	`/api/enrollments/{id}/balance`	path id	`BalanceResponse` · 200 / 404
GET	`/api/enrollments/{id}/payments`	path id	`List<PaymentResponse>` · 200

8.1 Sample request and response

Create a course — `POST /api/courses` → 201 Created

```
// Request
{ "code": "HND-NE", "name": "HND in Network Engineering",
  "level": "HND", "durationMonths": 18, "totalFee": 320000.00 }

// Response (201)
{ "id": 2, "code": "HND-NE", "name": "HND in Network Engineering",
  "level": "HND", "durationMonths": 18, "totalFee": 320000.00, "active": true }
```

Atomic registration — `POST /api/registrations` → 201 Created

```
// Request
{ "student": { "firstName": "Kasun", "lastName": "Perera", "nic": "200012345678",
    "email": "kasun@example.lk", "phone": "+94771234567",
    "address": "No. 1, Main Street, Kandy", "dateOfBirth": "2000-05-14" },
  "courseId": 2, "batch": "BATCH-2026-03",
  "firstPayment": { "amount": 50000.00, "paymentDate": "2026-10-19", "method": "CASH" } }

// Response (201) — student, enrolment and payment created in one transaction
{ "student": { "id": 357, "studentNumber": "AB2026-0358", "firstName": "Kasun", ... },
  "enrollment": { "id": 1064, "studentNumber": "AB2026-0358", "courseCode": "HND-NE",
                  "batch": "BATCH-2026-03", "enrolledOn": "2026-10-19" },
  "payment": { "id": 1066, "receiptNumber": "RC2026-001066",
               "amount": 50000.00, "paymentDate": "2026-10-19", "method": "CASH" } }
```

Paginated student search — `GET /api/students?q=Fernando&page=0&size=2` → 200 OK

```
{ "content": [
    { "id": 1, "studentNumber": "AB2026-0001", "firstName": "Ruwan", "lastName": "Fernando",
      "nic": "96100001V", "email": "ruwan.fernando1@applebay.lk",
      "phone": "+94721000001", "address": "No. 2, Station Road, Kandy", "dateOfBirth": "1998-01-08" },
    { "id": 24, "studentNumber": "AB2026-0024", "firstName": "Pawani", "lastName": "Fernando", ... } ],
  "page": 0, "size": 2, "totalElements": 21, "totalPages": 11 }
```

Bulk attendance — `POST /api/attendance` → 200 OK

```
// Request — the whole class in one call
{ "courseId": 2, "batch": "BATCH-2026-01", "date": "2026-10-19",
  "records": [ { "enrollmentId": 1, "status": "PRESENT" },
               { "enrollmentId": 6, "status": "LATE" },
               { "enrollmentId": 11, "status": "ABSENT" } ] }

// Response
{ "courseId": 2, "batch": "BATCH-2026-01", "date": "2026-10-19",
  "created": 1, "updated": 2, "unchanged": 0 }
```

Balance lookup — `GET /api/enrollments/lookup?query=AB2026-0001` → 200 OK

```
{ "enrollmentId": 355, "studentNumber": "AB2026-0001", "studentName": "Ruwan Fernando",
  "courseCode": "HND-NE", "courseName": "HND in Network Engineering",
  "courseFee": 320000.00, "totalPaid": 10000.00, "outstanding": 310000.00 }
```

9. Error Handling Approach

Every handled error leaves the API as the same JSON document, produced by `GlobalExceptionHandler`. The frontend only needs to understand one shape, and the HTTP status tells the client what happened even before the body is parsed.

```
{
  "timestamp": "2026-10-19T22:37:13.462",
  "status": 422,
  "error": "Unprocessable Entity",
  "message": "Payment exceeds balance",
  "path": "/api/payments",
  "fieldErrors": null          // populated for validation failures
}
```

Table 9.1 — Exception-to-status mapping

Exception	Trigger	Status
`MethodArgumentNotValidException`	Bean Validation failure (@NotBlank, @Pattern, @DecimalMin, ...); returns `fieldErrors` map	400
`ResourceNotFoundException`	Student / course / enrolment id does not exist	404
`DuplicateResourceException`	Duplicate NIC, email, course code or enrolment batch	409
`BusinessRuleException`	Overpayment, payment on a settled enrolment, enrolment into an inactive course, foreign enrolment in a class request	422
`DataIntegrityViolationException`	Database-level constraint conflict (final safety net)	409

9.1 Live error samples

```
// 400 Bad Request — invalid NIC (POST /api/students)
{ "timestamp": "2026-10-19T22:37:12.755", "status": 400, "error": "Bad Request",
  "message": "Validation failed", "path": "/api/students",
  "fieldErrors": { "nic": "Invalid NIC" } }

// 409 Conflict — duplicate course code (POST /api/courses)
{ "timestamp": "2026-10-19T22:37:12.679", "status": 409, "error": "Conflict",
  "message": "Course code already exists: HND-NE", "path": "/api/courses", "fieldErrors": null }

// 422 Unprocessable Entity — overpayment (POST /api/payments)
{ "timestamp": "2026-10-19T22:37:13.462", "status": 422, "error": "Unprocessable Entity",
  "message": "Payment exceeds balance", "path": "/api/payments", "fieldErrors": null }
```

10. Testing & Quality Verification

10.1 Automated service-layer unit tests

Five JUnit 5 tests run under Maven. The three service tests use Mockito to isolate business rules; the Spring context test confirms the application boots and wires its beans. Command: `./mvnw test` (JDK 21).

Table 10.1 — Automated test results (mvn test)

Test class / method	Verifies	Result
`PaymentServiceTest.recordPayment_rejectsPaymentGreaterThanOutstandingBalance`	amount 25,000 vs outstanding 20,000 → `BusinessRuleException`; no row saved	PASS
`PaymentServiceTest.recordPayment_rejectsPaymentWhenEnrollmentIsFullyPaid`	payment on a settled enrolment → `BusinessRuleException`	PASS
`PaymentServiceTest.recordPayment_generatesReceiptAndSavesValidInstalment`	valid instalment → receipt RC{year}-000001 and amount persisted	PASS
`AttendanceServiceTest.eligibility_flagsStudentAtExactlyEightPercentAsEligible`	80% → eligible (true); 70% → not eligible (false)	PASS
`ApplebayBackendApplicationTests.contextLoads`	Spring context starts and all beans resolve (against PostgreSQL)	PASS

Surefire summary: **Tests run: 5, Failures: 0, Errors: 0, Skipped: 0.** The full source of the service tests is in Appendix A.

10.2 Manual / API test cases and results

The table below records end-to-end cases executed against the running system. Each "actual" column is the real observed output.

Table 10.2 — Manual API test log (executed)

ID	Case	Expected	Actual result	Status
TC-01	Create course with valid data	201, persisted	201, `CourseResponse` returned	PASS
TC-02	Create course with duplicate code	409 Conflict	409, "Course code already exists: HND-NE"	PASS
TC-03	Register student with invalid NIC	400 + fieldErrors	400, `{"nic":"Invalid NIC"}`	PASS
TC-04	Register duplicate NIC	409 Conflict	409, "NIC already registered"	PASS
TC-05	Register student → unique number	AB2026-xxxx auto	AB2026-0358 generated (sequence)	PASS
TC-06	Enrol into inactive course	422	422, "Cannot enroll into an inactive course"	PASS
TC-07	Duplicate enrolment (same course+batch)	409	409, duplicate message from service	PASS
TC-08	Paginated student search	200 + page metadata	page 0, size 2, totalElements 21, totalPages 11	PASS
TC-09	Bulk mark a class	200, counts	`{created, updated, unchanged}` returned for 35 students	PASS
TC-10	Correct an existing day	updated, no duplicate	Existing row updated in place; no new row	PASS

ID	Case	Expected	Actual result	Status
TC-11	Eligibility at ≥ 80%	eligible true / false	100% → eligible true; below 80% → false	PASS
TC-12	Overpayment attempt	422, no row	422, "Payment exceeds balance"; payment not saved	PASS
TC-13	Atomic registration rollback	no orphan records	Failure at payment stage leaves no new student/enrolment	PASS
TC-14	Dashboard aggregates	200, metrics	students / active courses / today's attendance / fees collected	PASS

11. Installation & Running Guide

Prerequisites
Java **JDK 21** (the project targets Java 21), Maven (or the bundled `./mvnw`), Node.js 18+ and PostgreSQL 16. **Use JDK 21, not the newest JDK** — Lombok does not yet support very recent JDKs and the build will fail with an `EndPosTable` error on JDK 27.

11.1 Database

```
# 1. create the database
createdb -U postgres applebay          # or: CREATE DATABASE applebay;

# 2. create the schema (tables, keys, constraints, sequences)
psql -U postgres -d applebay -f database/schema.sql

# 3. load the sample dataset (355+ students, courses, enrolments, payments, attendance)
psql -U postgres -d applebay -f database/seed.sql

# 4. (optional) run the management report queries
psql -U postgres -d applebay -f database/report_queries.sql
```

The backend connects using the credentials in `backend/src/main/resources/application.properties`, which default to `postgres/postgres` on `localhost:5432/applebay` and can be overridden with the environment variables `SPRING_DATASOURCE_URL`, `SPRING_DATASOURCE_USERNAME` and `SPRING_DATASOURCE_PASSWORD`.

11.2 Backend (Spring Boot)

```
cd backend
JAVA_HOME=/path/to/jdk-21 ./mvnw spring-boot:run      # starts on http://localhost:8080/api
```

11.3 Frontend (Next.js)

```
cd frontend
npm install
# point the UI at the API (already set for local development in .env.local)
# NEXT_PUBLIC_API_BASE_URL=http://localhost:8080/api
npm run dev            # starts on http://localhost:3000
```

11.4 Verifying the installation

```
curl http://localhost:8080/api/dashboard
# {"totalStudents":356,"activeCourses":4,"attendanceMarkedToday":...,"totalFeesCollected":...}
```

Troubleshooting
"relation student_number_seq does not exist" — `database/schema.sql` was not run; the sequences are created there, not by Hibernate.
Build fails with `EndPosTable` — you are on too new a JDK; use JDK 21.
Frontend shows a network error — the backend is not running, or `NEXT_PUBLIC_API_BASE_URL` is wrong; CORS allows `localhost:3000` by default.

12. Limitations, Future Work & Conclusion

12.1 Limitations

- **No authentication or RBAC.** The brief does not require it; the API relies on network trust and CORS. In production, registrar/lecturer/cashier roles would be separated.
- **Course editing is API-only.** The course `PUT` endpoint is implemented and tested at the API level, but the current UI exposes only create, deactivate and reactivate.
- **Attendance requires connectivity.** Marking is a live request; there is no offline cache.
- **Single-currency, single-institute.** Fees are LKR only and the schema assumes one institute.

12.2 Future work

- Add Spring Security with role-based access control for the four user roles.
- Surface course editing and bulk CSV student import in the UI.
- Integrate a payment gateway (e.g. PayHere/Stripe) and SMS/email alerts when attendance drops below 80%.
- Add more service and controller slice tests and a CI pipeline.

12.3 Conclusion

The Applebay Academic, Attendance & Fee Management System replaces manual registers and spreadsheets with a transactional full-stack application. A normalised 3NF schema with database-enforced constraints, a strictly layered Spring Boot backend (Controller → Service → Repository with DTOs), atomic registration, dynamic balance control, bulk attendance with in-place correction and live exam-eligibility all work together so that data moves predictably from the browser to PostgreSQL and back. Fourteen functional requirements and the key non-functional requirements are implemented and verified by five automated tests and fourteen documented API test cases.

Appendix A. Service-layer unit tests

File: `backend/src/test/java/com/applebay/demo/service/PaymentServiceTest.java`

```
@ExtendWith(MockitoExtension.class)
class PaymentServiceTest {
    @Mock private PaymentRepository paymentRepository;
    @Mock private EnrollmentRepository enrollmentRepository;
    @InjectMocks private PaymentService paymentService;

    @Test
    void recordPayment_rejectsPaymentGreaterThanOutstandingBalance() {
        when(enrollmentRepository.findByIdForUpdate(1L))
                .thenReturn(Optional.of(enrollmentWithFee("100000.00")));
        when(paymentRepository.sumByEnrollmentId(1L)).thenReturn(new BigDecimal("80000.00"));

        assertThrows(BusinessRuleException.class, () ->
                paymentService.record(1L, new BigDecimal("25000.00"), LocalDate.now(), PaymentMethod.CASH));

        verify(paymentRepository, never()).save(any(Payment.class));
    }
    // ... two further cases: fully-paid rejection and successful receipt generation
}
```

File: `backend/src/test/java/com/applebay/demo/service/AttendanceServiceTest.java`

```
@Test
void eligibility_flagsStudentAtExactlyEightPercentAsEligible() {
    AttendanceSummaryView eightyPercent = summary(10L, 10, 8);
    AttendanceSummaryView seventyPercent = summary(11L, 10, 7);
    when(enrollmentRepository.findClassRoster(2L, "BATCH-2026-01"))
            .thenReturn(List.of(enrollment(10L, "AB2026-0010"), enrollment(11L, "AB2026-0011")));
    when(attendanceRepository.summarize(any()))
            .thenReturn(List.of(eightyPercent, seventyPercent));

    List<EligibilityRow> rows = attendanceService.eligibility(2L, "BATCH-2026-01");
    assertTrue(rows.get(0).eligible());          // 80% → eligible
    assertFalse(rows.get(1).eligible());         // 70% → not eligible
}
```

Appendix B. Reporting SQL & References

File: `database/report_queries.sql` — selected queries.

```
-- RQ1: Outstanding balance per enrolment (backs FR12)
SELECT s.student_number, s.first_name || ' ' || s.last_name AS student_name,
       c.code AS course_code, e.batch, c.total_fee AS course_fee,
       COALESCE(SUM(p.amount), 0)               AS total_paid,
       c.total_fee - COALESCE(SUM(p.amount), 0) AS outstanding
FROM enrollments e
JOIN students s ON s.id = e.student_id
JOIN courses  c ON c.id = e.course_id
LEFT JOIN payments p ON p.enrollment_id = e.id
GROUP BY s.student_number, student_name, c.code, e.batch, c.total_fee
HAVING c.total_fee - COALESCE(SUM(p.amount), 0) > 0
ORDER BY outstanding DESC;
```

```
-- RQ4: Daily attendance breakdown (present / late / absent per date)
SELECT a.attendance_date,
       COUNT(*) FILTER (WHERE a.status = 'PRESENT') AS present,
       COUNT(*) FILTER (WHERE a.status = 'LATE')    AS late,
       COUNT(*) FILTER (WHERE a.status = 'ABSENT')  AS absent
FROM attendance a
GROUP BY a.attendance_date
ORDER BY a.attendance_date DESC;
```

References (Harvard style)

- Baeldung, 2024. *Transactions with Spring and JPA*. Available at: https://www.baeldung.com/transaction-configuration-with-jpa-and-spring [Accessed 19 October 2026].
- Fowler, M., 2002. *Patterns of Enterprise Application Architecture*. Boston: Addison-Wesley.
- Martin, R.C., 2017. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Boston: Prentice Hall.
- PostgreSQL Global Development Group, 2026. *PostgreSQL 16 Documentation*. Available at: https://www.postgresql.org/docs/16/index.html [Accessed 19 October 2026].
- Spring Team, 2026. *Spring Boot Reference Documentation 3.3*. Available at: https://docs.spring.io/spring-boot/docs/3.3.x/reference/html/ [Accessed 19 October 2026].
- Vercel, 2026. *Next.js App Router Documentation*. Available at: https://nextjs.org/docs [Accessed 19 October 2026].