420-N34_LA Java Web Programming
Prof. Haikel Hichri, Email: hhichri@champlaincollege.qc.ca

Term Project

Milestone 1 : backend in Spring boot

Luca Rarau, Loukmane Bessam

Overview

Design and implement a RESTful Web Service application using a clean 3-layer architecture (Controller → Service → Repository) with DTOs and an in-memory H2 database. You will publish your backend and preload it with sample data. You must choose one of these topics and define at least two core resources with a relationship:

- Library Management System: Book, Author, Borrower
- Hospital Appointment System: Doctor, Patient, Appointment
- Event Management System: Event, Attendee
- University Department System: Department, Professor
- Cinema Booking System: Movie, Screening
- Fragrance Online Shop: Product, Customer, Order
- Handyman Booking System: Handyman, Customer, Job

Note: You are free to add a third resource (e.g., a join or transactional entity like Appointment/Order/Job) if it improves realism, but the minimum is two.

Learning Objectives

- Develop a front-end application using ReactJS (will be used in a later milestone)
- Develop RESTful web services using Spring Boot
- Integrate a React app with a RESTful web service
- Design full-stack applications using ReactJS, Spring Boot, and an SQL database
- Integrate RESTful web services with an SQL database
- Use an HTTP client to test web services (e.g., Postman, Insomnia, curl)

Instructions

- Teams: 2–3 students.
- Read the evaluation rubric before you begin.
- Use Git for version control. One repository per team with clear commit history and a README.

Submission Requirements

1. Due date is indicated on Moodle
2. Zip your project and submit it to Moodle.

3. You will present your work in class during a peer review by another student.
4. You will also be graded on how well you peer review your partner.

Part 1: Design

Conceive a three-layer system, determine its data model, and plan the UI.

Deliverable: A short design report (PDF) including the items below.

1. Introduction

- One paragraph that explains the system's purpose, primary users, and high-level features.

The project focuses on developing a University Department System using a RESTful web service architecture, where the core resources are Department and Professor. The system's primary purpose is to manage and provide access to information about academic departments and the professors affiliated with them. The primary users would be university staff and administrators who need to maintain an up-to-date registry of departmental structure and faculty assignments. High-level features will include the ability to create, read, update, and delete (CRUD) information for both Departments and Professors. Furthermore, the system will support a one-to-many relationship where a Department has multiple Professors. This relationship will be exposed through aggregated endpoints, allowing users to, for example, retrieve a list of all professors belonging to a specific department. This provides a clean, layered backend for a future front-end application.

2. Design Section

- Resources
  - Define at least two resources.

  Department Resource

  Professor Resource

  - Provide UML class diagrams for each entity (fields, types, constraints).

Department
- id: Long << Primary Key >>
- name: String << Not Null, Unique >>
- code: String <<Not Null, Unique>>
- yearEstablished: Integer << @Min (1800), @Max(2100)>>
- professors: List<Professor> << OneToMany, Lazy >>

Professor
- id: Long <<Primary Key >>

- firstName: String << Not Null>>
- lastName: String << Not Null >>
- email: String << Not Null, Unique, Email >>
- title: String (e.g., Fulltime Professor)
- department: Department<< ManyToOne, Eager >>

- Include DTOs: request and response DTOs for each resource.

DepartmentRequestDTO	**DepartmentResponseDTO**	**DepartmentSummaryDTO**
- name: String << @NotBlank >>	**- id**: Long	**- id**: Long
- code: String << @NotBlank >>	**- name**: String	**- name**: String
- yearEstablished: Integer << @PastOrPresent >>	**- code**: String	**- code**: String
	- yearEstablished: Integer	
	- professorCount: Integer (e.g., calculated)	

ProfessorRequestDTO	**ProfessorResponseDTO**	**ProfessorSummaryDTO**
- firstName: String << @NotBlank>>	**- id**: Long	**- id**: Long
- lastName: String<<@NotBlank >>	**- firstName**: String	**- firstName**: String
- email: String << @Email>>	**- lastName**: String	**- lastName**: String
- title: String	**- email**: String	**- title**: String
- departmentId: Long<<Not Null >>	**- title**: String	
	- departmentId: Long	
	- department: DepartmentSummary	

- Example note: For social systems, consider Post includes images or short videos.

- Relationships
  - Specify relationships (e.g., one-to-many, many-to-many via join entity). o Provide an pERD/database schema diagram (tables, PK/FK, cardinalities).

Table: DEPARTMENT		**Table: PROFESSOR**	
PK	**id**: Long	PK	**id**: Long
	name: String		**firstName**: String
	code: String		**lastName**: String
	yearEstablished: Integer		**email**:String
			title: String
		FK	**department_id**: Long({REFERENCES DEPARTMENT(id))
Cardinality	1	**Cardinality**	N

- Clarify cascade and fetch strategies (e.g., CascadeType.PERSIST on one-to-many, LAZY collections).

JPAConcept	**Configuration**	**Explanation**	
CascadeStrategy		**CascadeType.PERSIST**[2]	If a new `Department` is saved, any **new** Professor objects attached to its list are also automatically saved. **CascadeType.REMOVE is generally avoided** on the one-to-many side as it would delete all professors if a department is deleted, which may not be the desired business rule.
FetchStr		**FetchType.LAZY**[3]	The collection of `Professor` entities is **not** loaded from the database until the collection is explicitly accessed (e.g., to build the DepartmentWithProfessorsResponseDTO. This is the best practice for collections to prevent performance issues (over-fetching).

ategy			
Mapping	**MappedBy department**	Used to indicate that the relationship is managed by the `department` field in the Professor entity.	

- Endpoints
    - List all REST endpoints by resource. o Required mappings for each resource: GET (all), GET (one by id), POST, PUT, DELETE. o For aggregated/relationship endpoints, provide GETs (e.g., GET /authors/{id}/books). o Follow REST naming conventions, plural nouns, and resource-oriented paths.
    - Include example requests/responses (JSON) and expected HTTP status codes.

.Get All Departments:Method:GET**Path:** `/api/departments` **Description:** Retrieves a list of all departments.

Expected Status: 200 OK

Example Response (JSON): A list of `DepartmentResponseDTO` objects.

JSON
```
[
  { "id": 1, "name": "Computer Science", "code": "CS", "professorCount": 15 },
  { "id": 2, "name": "History", "code": "HIST", "professorCount": 8 }
]
```

2. Get Department by ID.Method: GET **Path:** `/api/departments/{id}` **Description:** Retrieves a single department by its ID. **Expected Status:** 200 OK(if found) or 404 Not Found (if missing)

Example Response (JSON): A single `DepartmentResponseDTO`.

JSON
```
{ "id": 1, "name": "Computer Science", "code": "CS", "yearEstablished": 1995,
"professorCount": 15 }
```

3. Create New DepartmentMethod: POST**Path:** `/api/departments` **Description:** Creates a new department.

Expected Status: 201 Createdwith a `Location` header pointing to the new resource. or 400 Bad Request (if validation fails)

Example Request (JSON): A `DepartmentRequestDTO`.

JSON
```
{ "name": "Physics", "code": "PHYS", "yearEstablished": 2000 }
```

4. Update Existing Department.Method:PUT **Path:** `/api/departments/{id}`**Description:** Updates the department with the given ID (idempotent operation). **Expected Status:** 200 OK(if successful) or 404 Not Found(if missing)

Example Request (JSON): A `DepartmentRequestDTO`.

JSON
```
{ "name": "Physics & Astronomy", "code": "PHYS", "yearEstablished": 2000 }
```

5. Delete Department Method: DELETE **Path:** `/api/departments/{id}` **Description:** Deletes the department with the given ID. **Expected Status:**204 No Content(if successful) or 404 Not Found(if missing)

- **Example Response:** Empty body.

6. Aggregated Endpoint: Get Professors by Department

- Professor responses automatically include department information through the embedded DepartmentSummary field, so a separate /api/professors/{id}/department endpoint is no longer needed.
- **Description:** Retrieves a list of all professors belonging to the specified department.
- **Expected Status:** 200 OK (if department found) or 404 Not Found (if department missing)
- **Example Response (JSON):** A `DepartmentWithProfessorsResponseDTO`.

JSON
```
{
  "id": 1,
  "name": "Computer Science",
  "professors": [
    { "id": 101, "firstName": "Alice", "lastName": "Smith" },
    { "id": 102, "firstName": "Bob", "lastName": "Jones" }
  ]
}
```

Professor Endpoints

- The **Professor** resource is the “many” side of the relationship.
- **Get All Professors**
- **Method:** GET
- **Path:** /api/professors
- **Description:** Retrieves a list of all professors.
- **Expected Status:** 200 OK
- **Example Response (JSON):**
- [
- { "id": 101, "firstName": "Alice", "lastName": "Smith", "email": "a.smith@uni.ca" },
- { "id": 102, "firstName": "Bob", "lastName": "Jones", "email": "b.jones@uni.ca" }
-]
- **Get Professor by ID**
- **Method:** GET
- **Path:** /api/professors/{id}
- **Description:** Retrieves a single professor by their ID.
- **Expected Status:** 200 OK (if found) or 404 Not Found (if missing)
- **Example Response (JSON):**
- { "id": 101, "firstName": "Alice", "lastName": "Smith", "email": "a.smith@uni.ca", "title": "Full Professor" }
- **Create New Professor**
- **Method:** POST
- **Path:** /api/professors

- **Description:** Creates a new professor and assigns them to a department using the required departmentId.
- **Expected Status:** 201 Created (with Location header) or 400 Bad Request (if validation fails or departmentId is invalid)
- **Example Request (JSON):**
- { "firstName": "Carol", "lastName": "Davis", "email": "c.davis@uni.ca", "title": "Lecturer", "departmentId": 2 }
- **Update Existing Professor**
- **Method:** PUT
- **Path:** /api/professors/{id}
- **Description:** Updates the professor with the given ID.
- **Expected Status:** 200 OK (if successful) or 404 Not Found (if missing)
- **Example Request (JSON):**
- { "firstName": "Carol", "lastName": "Davis", "email": "c.davis@uni.ca", "title": "Assistant Professor", "departmentId": 2 }
- **Delete Professor**
- **Method:** DELETE
- **Path:** /api/professors/{id}
- **Description:** Deletes the professor with the given ID.
- **Expected Status:** 204 No Content (if successful) or 404 Not Found (if missing)
- **Example Response:** Empty body.
- **Aggregated Endpoint: Get Department by Professor**
- **Method:** GET
- **Path:** /api/professors/{id}/department
- **Description:** Retrieves the department details for the specified professor.
- **Expected Status:** 200 OK (if professor found) or 404 Not Found (if professor missing)
- **Example Response (JSON):**
- {
- "id": 101,
- "firstName": "Alice",
- "lastName": "Smith",
- "department": { "id": 1, "name": "Computer Science", "code": "CS" }
- }
- ---
- **Validation**
- **Department Resource**
  For the DepartmentRequestDTO, the **name** field requires the @NotBlank annotation, and any input that is null or only whitespace should trigger a **400 Bad Request** with the message "name must not be blank".
  Similarly, the **code** field also requires @NotBlank.
  The yearEstablished field must be validated using @Min(1800) and @Max(2100) to ensure the year is between 1800 and 2100, returning 400 Bad Request if invalid..
- **Professor Resource**
  For the ProfessorRequestDTO:
- firstName and lastName must have @NotBlank; empty values trigger **400 Bad Request**.
- email must include both @Email and @NotBlank; invalid or missing formats return **400 Bad Request** with "email must be a well-formed email address".
- departmentId must have @NotNull and @Positive to ensure it links to a valid department.
  All validation failures are handled globally by a @ControllerAdvice that returns **400 Bad Request** with standardized error messages.
- ---
- **Wireframes (for later front-end)**
- Provide 2–3 key wireframes:
- List view
- Detail view

- Create/Edit form
  (Use tools such as Figma, Pencil, or Balsamiq.)
- ---
- **Non-Functional Considerations**
- **Error Handling Approach**
  The system uses a global error handler with @ControllerAdvice to keep responses consistent.
- **Missing Resources:**
  A custom NotFoundException is thrown from the Service layer when a Department or Professor doesn't exist.
  The @ControllerAdvice maps it to **HTTP 404 Not Found** with a standardized JSON error body.
- **Invalid Input:**
  Validation failures (@NotBlank, @Email, etc.) throw MethodArgumentNotValidException.
  The handler maps it to **HTTP 400 Bad Request**, including field-specific error details.
- ---
- **Naming and Packaging Conventions**
- **Root package:** com.yourorg.yourapp
- **Layers:**
- Presentation layer
- Business Logic Layer
- Data Access Layer
- Mapper layer
- DTOs
- Exception handling
- **Class naming:**
- Entities → PascalCase (e.g., Book)
- DTOs → RequestDTO / ResponseDTO suffix
- Controllers, Services, Repositories → suffixed accordingly (DepartmentController, ProfessorService, etc.)
- ---
- **Testing Approach**
- **Unit Tests:** Focus on Service layer logic (data transformation and NotFound handling).
- **Integration Tests:** Ensure Controller endpoints return correct HTTP codes (200, 201, 404).
- **Manual Tests:** Verify CRUD operations and error handling using Postman.
- .

Submission for Part 1: PDF with the above content, plus a link to your Git repo (initial structure allowed).

https://github.com/LKBSM/UniversityDepartmentSystem

Part 2: Implementing the Backend

Implement the web service and database for the resources and relationships defined in Part 1.

Deliverables:

- Source code (Spring Boot project) in your team Git repository.
- A README with:
  - Project description o How to run locally o API endpoints table o Sample curl/Postman requests o Deployment URL

- Credentials (if any)
- A short demo video (3–6 minutes) showing:
  - Running the application o Executing key endpoints (CRUD + relationship + aggregated DTOs) o H2 console view of data o Successful deploy

Technical Requirements (R1–R13)

R1. 3-Layer architecture: Controller → Service → Repository
- Controllers: REST endpoints using ResponseEntity.
- Services: business logic; handle NotFound and validation mapping; transactional boundaries where needed.
- Repositories: Spring Data JPA interfaces.

R2. At least two resources
- Example: Author and Book; Doctor and Patient; Department and Professor.

R3. Implement the relationship between the two resources
- Prefer one-to-many or many-to-one for simplicity. Many-to-many must use a join table (or explicit join entity).

R4. Follow Java/Spring naming conventions
- Packages: com.yourorg.yourapp
- Entities: PascalCase (Book), fields: camelCase (publishedDate), DTOs: Suffix with Request/ResponseDTO, Controllers: Suffix with Controller, Services: Suffix with Service, Repositories: Suffix with Repository.

R5. Use H2 in-memory DB with proper configuration
- application.properties: H2 console enabled, datasource URL, Hibernate DDL auto update/create-drop, show SQL optional.
- Separate tables for each entity; correct PK/FK.

R6. Implement GET all, GET one, POST, PUT, DELETE for both resources Controllers accept RequestDTOs and return ResponseDTOs.
- PUT should be idempotent; PATCH is optional.

R7. Two aggregated response DTOs based on the one-to-many relationship
- Example: AuthorWithBooksResponseDTO (Author + List)
- And BookWithAuthorResponseDTO (Book + AuthorSummaryDTO)

R8. Use both request and response DTOs
- RequestDTOs exclude server-generated fields (id, timestamps).
- ResponseDTOs hide sensitive/internal fields.

R9. Use ResponseEntity
- Return ResponseEntity from controllers.

R10. Return appropriate HTTP status codes
- 201 Created on successful POST, with Location header.
- 204 No Content on successful DELETE.
- 200 OK for successful GET/PUT.
- 404 Not Found for missing resources.
- 400 Bad Request for validation errors.

R11. Implement NotFoundException for both resources
- Custom exception + @ControllerAdvice to map to 404 with a standardized error body.

R12. Publish the app on a web hosting service
- Deploy your Spring Boot app (e.g., Render/Railway/Heroku alternatives). Provide a public base URL in README.

R13. Seed at least 10 records in each table
- Use data.sql or a CommandLineRunner bean to insert data on startup.

Suggested Project Structure (Packages)

- com.chamsoft.app
  - config (H2 config, CORS if needed) o Data access Layer
  - Business Logic Layer o presentation layer
    - Controllers
    - request
    - response o exception o Mapper layer (manual mappers or MapStruct if allowed) o bootstrap (data seeding via CommandLineRunner)

Example Endpoint Design (for a one-to-many)

Assume Author (one) → Book (many).

- Authors o GET /api/authors o GET /api/authors/{id} o POST /api/authors o PUT /api/authors/{id} o DELETE /api/authors/{id}
  - GET /api/authors/{id}/books (aggregated)
- Books o GET /api/books o GET /api/books/{id} o POST /api/books o PUT /api/books/{id} o DELETE /api/books/{id}
  - GET /api/books/{id}/author (aggregated)

HTTP examples:

- POST /api/authors → 201 Created, Loca on: /api/authors/{id}
- DELETE /api/books/{id} → 204 No Content
- GET missing id → 404 with { " mestamp": ..., "status": 404, "error": "Not Found", "message": "Book 123 not found", "path": "/api/books/123" }

DTO Guidelines

- AuthorRequestDTO: name, bio
- AuthorResponseDTO: id, name, bio
- BookRequestDTO: title, isbn, authorId
- BookResponseDTO: id, title, isbn Aggregated:
  - AuthorWithBooksResponseDTO: author fields + List o BookWithAuthorResponseDTO: book fields + AuthorSummaryDTO
- Summary DTOs keep nested data concise (id, name/title).

Validation and Error Handling

- Use javax.validation annotations on RequestDTOs: @NotBlank, @Size, @Email, @Positive, etc.
- Add @Valid on controller method parameters.
- Global exception handler using @RestControllerAdvice:

- Handle NotFoundException → 404 o Handle MethodArgumentNotValidException → 400 with field errors o Optionally handle DataIntegrityViolationException → 409

H2 Configuration Example application.properties:

- spring.datasource.url=jdbc:h2:mem:demo;H2DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE
- spring.datasource.driverClassName=org.h2.Driver
- spring.jpa.hibernate.ddl-auto=create-drop
- spring.jpa.show-sql=true
- spring.h2.console.enabled=true
- spring.h2.console.path=/h2-console
- Seeding: a @Bean CommandLineRunner that saves 10+ records per table through repositories

---

Testing Your API

- Use Postman/Insomnia collections or curl scripts
- Include a small Postman collection JSON in the repo (optional but recommended) □ Verify:
  - CRUD for both resources o Aggregated endpoints o Correct status codes o Error responses for missing ids and invalid payloads

---

Deployment

- Deploy your Spring Boot JAR on a hosting provider (e.g., Render or Railway)
- Ensure CORS is enabled if you will later access from React
- Provide the public URL in README and in your demo video
- If H2 memory resets on redeploy, ensure seeding covers your demo

---

Submission Checklist

- GitHub repo link with:

- Design report PDF (Part 1) o Complete Spring Boot project o README (how to run locally, endpoints, sample requests, deployment URL) o Postman collection (optional)
- Running deployment URL (R12)
- H2 seeded with at least 10 records per table (R13)

Evaluation Rubric (Total: 25 points)

- Design Report (Part 1) — 5 pts
  - Clear system description — 1 o Resources, UML, DTOs, ERD — 2 o Endpoints list with REST conventions — 1 o Wireframes and validation plan — 1
- Backend Architecture & Code Quality — 6 pts
  - Proper 3-layer separation — 2 o Naming, packaging, DTO usage — 2 o Mappers and service logic clarity — 2
- Data & Persistence — 4 pts
  - H2 configuration and schema correctness — 2 o Seeding 10+ per table and relationship set up — 2
- API Functionality — 6 pts o CRUD for both resources — 2 o Relationship endpoints and two aggregated DTOs — 2 o ResponseEntity usage and correct HTTP codes — 2
- Error Handling & Validation — 2 pts o NotFoundException with @ControllerAdvice — 1 o Request validation and 400 responses — 1
- Deployment & Documentation — 2 pts
  - Live deployment with working endpoints — 1 o README and demo video clarity — 1

Optional stretch (extra credit up to +2):

- Basic tests (service/controller)
- Swagger/OpenAPI documentation
- CI workflow (e.g., GitHub Actions build)

Starter Task Suggestions (Week Plan)

- Day 1-2: Pick domain, complete Part 1 (UML/ERD/endpoints/wireframes/validation)
- Day 3–4: Implement entities, repositories, DTOs, mappers, services
- Day 5: Implement controllers, exception handling, validation
- Day 6: Seed data, test with Postman, polish README
- Day 7: Deploy, record demo video, final checks

Peer Review Checklist

Criteria	Resource 1	Resource 2	Comments
☐ 3-layer pattern used			
☐ Spring Boot and Java naming conventions used			
☐ Fill the app with data: at least 3 users and 5-10 posts for each user.			
☐ Get All correctly implemented			
o Uses correct REST endpoint naming (write it in the comments)			
o Returns a list of ResponseDTO (check code)			
☐ Get by resource id correctly implemented			
o Uses correct REST endpoint naming (write it in the comments)			
o Returns a ResponseDTO (check code)			
o Using a non-existing resource Id gives a 404 Not Found HTTP status (check Postman)			
☐ Add a resource correctly implemented			
o Uses correct REST endpoint naming (write it in the comments)			
o Takes a Request Body that contains a RequestDTO (check code)			
o Returns a ResponseDTO (check code)			
o Resource has been added to database (check h2-console)			
☐ Update a resource correctly implemented			
o Uses correct REST endpoint naming (write it in the comments)			
o Takes a Request Body that contains a RequestDTO (check code)			
o Takes a Path Variable that contains the Resource Id (check code)			
o Returns a ResponseDTO (check code)			
o Resource has been updated in the database (check h2console)			
o Using a non-existing resource Id gives a 404 Not Found HTTP status (check Postman)			
☐ Delete by resource id correctly implemented			
o Uses correct REST endpoint naming (write it in the comments)			
o Takes a Path Variable that contains a Resource Id (check code)			
o Returns Void (check code)			

o Resource has been deleted from database (check h2console)			
o Using a non-existing resource Id gives a 404 Not Found HTTP status (check Postman)			
One to Many Relationship	Checklist (Y/N)	Comments	
☐ One aggregated GET endpoint has been implemented for the one-to-many relationship			
o Uses correct REST endpoint naming (write it in the comments)			
o Returns a list of ResponseDTO (check code)			
o The ResponseDTO contains information about the resources in the relationship.(check Postman)			