System Requirements Specification

Index

For

Student-Course

Version 1.0

IIHT Pvt. Ltd.
fullstack@iiht.com

TABLE OF CONTENTS

Student-Course-Microservice App

System Requirements Specification

1 PROJECT ABSTRACT

The **Student-Course** Application is designed to manage academic course details and student enrollment information in a modular and scalable architecture. Built using Spring Boot and microservices principles, this application separates functionality into distinct services for courses and students. Each microservice operates with its own database and communicates with others via REST APIs and service discovery via a Eureka Naming Server. The Course service integrates with the Student service to fetch student details when retrieving course-related information, promoting separation of concerns and efficient data access.

Following is the requirement specifications:

	App
Microservices	
1	Student Microservice
2	Course Microservice
Student Microservice	
1	Create Student
2	Get Student by ID
3	Update Student
Course Microservice	
1	Create Course
2	Get Course with Student details by Course ID
3	Update Course

2 CONSTRAINTS

2.1 Student Constraints

- When fetching a student by ID, if the student ID does not exist, the `controller` should throw a `NotFoundException` with the message:

  "Student with id {id} not found"

- When updating the student, if the student ID does not exist, the service method should throw a `NotFoundException` with the message:

  "Student with id {id} not found"

2.2 Course Constraints

- When updating a course, if the course ID does not exist, the service method should throw a `NotFoundException` with the message:

  "Course with id {id} not found"

- When fetching a course with student details by ID, the service method should throw the following exceptions:
  - ➔ If the course ID does not exist, the service method should throw a `NotFoundException` with the message:

    "Course with id {id} not found"
  - ➔ If the associated student record is **`null`**, the service method should throw a `NotFoundException` with the message:

    "Student with id {studentId} not found"
  - ➔ If the Student microservice returns 404 Not Found, the service method should catch `HttpClientErrorException.NotFound` and throws a `NotFoundException` with the message:

    "Student with id {studentId} not found"

2.3 Common Constraints

- For all rest endpoints receiving @RequestBody, validation check must be done and must throw custom exception if data is invalid
- All the business validations must be implemented in dto classes only.
- All the database operations must be implemented on entity object only
- Do not change, add, remove any existing methods in service layer
- In Repository interfaces, custom methods can be added as per requirements.

3 Database Operations

1. Student

- Class must be treated as an entity.
- Id must be of type id and generated by IDENTITY technique.
- `name` should not be blank and must be between 3 and 255 characters.
  - ➔ **Message if invalid:**
    - ➢ If blank: `"Name is required"`
    - ➢ If size is invalid: `"Name must be between 3 and 255 characters"`
- `email` should not be blank and must follow a valid email format.
  - ➔ **Message if invalid:**
    - ➢ If blank: `"Email is required"`
    - ➢ If size is invalid: `"Email should be valid"`
- `program` should not be blank and must be between 2 and 100 characters.
  - ➔ **Message if invalid:**
    - ➢ If blank: `"Program is required"`
    - ➢ If size is invalid: `"Program must be between 2 and 100 characters"`

2. Course

- Class must be treated as an entity.
- Id must be of type id and generated by IDENTITY technique.
- `name` should not be blank and must be between 3 and 255 characters.
  - ➔ **Message if invalid:**
    - ➢ If blank: `"Course name is required"`
    - ➢ If size is invalid: `"Course name must be between 3 and 255 characters"`
- `code` should not be blank and must be between 2 and 20 characters. It should be **unique** in the database and be **non-nullable.**
  - ➔ **Message if invalid:**
    - ➢ If blank: `"Course code is required"`

        - ➢ If size is invalid: `"Course code must be between 2 and 20 characters"`
- `studentId` must not be null.
    - ➔ **Message if invalid:**
        - ➢ If blank: `"Student ID must not be null"`

4 System Requirements

4.1 EUREKA-NAMING-SERVER

This is a discovery server for all the registered microservices. Following implementations are expected to be done:

a. Configure the Eureka server to run on port: 8761.
b. Configure the Eureka server to deregister itself as Eureka client.
c. Add appropriate annotation to Enable this module to run as Eureka Server.

You can launch the admin panel of Eureka server in the browser preview option.

4.2 STUDENT-MICROSERVICE

The student microservice manages student-related operations. In this microservice, you have to write the logic for StudentService.java and StudentController.java classes. Following implementations are expected to be done:

a. Configure this service to run on port: 8081.

4.3 COURSE-MICROSERVICE

The course microservice manages course-related operations and communicates with Student microservice via RESTTemplate. In this microservice, you have to write the logic for CourseService.java and CourseController.java. Following implementations are expected to be done:

a. Configure this service to run on port: 8082.
b. Configure a RESTTemplate to fetch Student details from Student microservice by Student ID.

5 Template Code Structure

5.1 COURSE

1 Package: com.course

Resources

CourseServiceApplication (Class)	This is the Spring Boot starter class of the application.	Already Implemented

2 Package: com.course.repo

Resources

Class/Interface	**Description**	**Status**
CourseRepository (interface)	● Repository interface exposing CRUD functionality for Course entity. ● You can go ahead and add any custom methods as per requirements.	Already Implemented

3 Package: com.course.service

Class/Interface	**Description**	**Status**
CourseService (class)	● Contains template method implementation. ● Need to provide implementation for managing courses related functionalities. ● Do not modify, add or delete any method signature.	To be implemented.

4 PACKAGE: COM.COURSE.CONTROLLER

Resources

Class/Interface	**Description**	**Status**
CourseController (Class)	● Controller class to expose all rest-endpoints for course related activities. ● May also contain local exception handler methods.	To be implemented

5 PACKAGE: COM.COURSE.DTO

Resources

Class/Interface	**Description**	**Status**
CourseResponse (Class)	Used to wrap department details along with associated student details. Acts as a response DTO for combined data.	Already implemented.
StudentDTO (Class)	DTO representing an student with fields (ID, name, email, program). Used in CourseResponse.	Already implemented.

6 PACKAGE: COM.COURSE.ENTITY

Resources

Class/Interface	**Description**	**Status**
Course (Class)	● This class is partially implemented. ● Annotate this class with proper annotation to declare it as an entity class with **id** as primary key. ● Generate the **id** using the IDENTITY strategy	Partially implemented.

7 PACKAGE: COM.COURSE.EXCEPTION

Resources

Class/Interface	**Description**	**Status**
NotFoundException (Class)	● Custom Exception to be thrown when trying to fetch, update or delete the course info which does not exist. ● Need to create Exception Handler for same wherever needed (local or global)	Already implemented.
ErrorResponse (Class)	● RestControllerAdvice Class for defining global exception handlers.	Already implemented.

	• Contains Exception Handler for **InvalidDataException** class. • Use this as a reference for creating exception handler for other custom exception classes	
RestExceptionHandler (Class)	• RestControllerAdvice Class for defining rest exception handlers. • Contains Exception Handler for **NotFoundException** class. • Use this as a reference for creating exception handler for other custom exception classes	Already implemented.

5.2 STUDENT

1 PACKAGE: COM.STUDENT

Resources

StudentServiceApplication (Class)	This is the Spring Boot starter class of the application.	Already Implemented

2 PACKAGE: COM.STUDENT.REPO

Resources

Class/Interface	**Description**	**Status**
StudentRepository (interface)	• Repository interface exposing CRUD functionality for Student entity. • You can go ahead and add any custom methods as per requirements.	Already Implemented

3 PACKAGE: COM.STUDENT.SERVICE

Class/Interface	**Description**	**Status**
StudentService (class)	• Contains template method implementation. • Need to provide implementation for managing student related functionalities. • Do not modify, add or delete any method signature.	To be implemented.

4 PACKAGE: COM.STUDENT.CONTROLLER

Resources

Class/Interface	**Description**	**Status**
StudentController (Class)	• Controller class to expose all rest-endpoints for student related activities. • May also contain local exception handler methods.	To be implemented

5 PACKAGE: COM.STUDENT.ENTITY

Resources

Class/Interface	**Description**	**Status**
Student (Class)	● This class is partially implemented. ● Annotate this class with proper annotation to declare it as an entity class with **id** as primary key. ● Generate the **id** using the IDENTITY strategy	Partially implemented.

6 PACKAGE: COM.STUDENT.EXCEPTION

Resources

Class/Interface	**Description**	**Status**
NotFoundException (Class)	● Custom Exception to be thrown when trying to fetch, update or delete the student info which does not exist. ● Need to create Exception Handler for same wherever needed (local or global)	Already implemented.
ErrorResponse (Class)	● RestControllerAdvice Class for defining global exception handlers.	Already implemented.

	• Contains Exception Handler for **InvalidDataException** class. • Use this as a reference for creating exception handler for other custom exception classes	
RestExceptionHandler (Class)	• RestControllerAdvice Class for defining rest exception handlers. • Contains Exception Handler for **NotFoundException** class. • Use this as a reference for creating exception handler for other custom exception classes	Already implemented.

6 METHOD DESCRIPTIONS

1. Service Class - Method Descriptions

A. CourseService – Method Descriptions

- Declare dependencies for `CourseRepository` and `RestTemplate` using **constructor injection**.

Method	**Task**	**Implementation Details**
`private final CourseRepository repo`	Inject repository dependency	- Injected via constructor - Provides access to course DB operations

`private final RestTemplate restTemplate`	Inject RestTemplate dependency	- Injected via constructor - Used to call Student microservice using REST
`private String studentServiceBase`	Holds the base URL of the Student microservice	- Injected using the `@Value` annotation from application properties or environment variables - Defaults to `http://student-service` if the property `student.service.url` is not found - Used to construct REST URLs for making cross-service calls to fetch student details

Method	**Task**	**Implementation Details**
`save()`	To save a new course	- Calls `repo.save(c)` - Returns the saved `Course` object
`get()`	To retrieve a course by ID	- Calls `repo.findById(id)` - Returns `Optional<Course>`
`update()`	To update course details by ID	- Checks existence using `repo.existsById(id)` - If not found, throws `NotFoundException` with message: `"Course with id " + id + " not found"` - Sets the ID manually using `updated.setId(id)` - Calls `repo.save(updated)` and returns updated course
`getCourseWithStudent()`	To get course and associated student details	- Retrieves course by ID using `repo.findById(id)` - If not found, throws `NotFoundException` with message: `"Course with id " + id + " not found"` - Constructs URL to fetch student using injected `studentServiceBase`

		- Calls `restTemplate.getForEntity(...)` to fetch student info - If response body is `null`, throws `NotFoundException` with message: `"Student with id " + id + " not found"` - If REST call throws `HttpClientErrorException.NotFound`, also throws same `NotFoundException` with message: `"Student with id " + id + " not found"` - Combines both into `CourseResponse` and returns it

B. EmployeeService – Method Descriptions

- Declare dependencies for `StudentRepository` using **constructor injection**.

Method	**Task**	**Implementation Details**
`private final StudentRepository repo;`	Inject repository dependency	- Final field for dependency injection - Injected via constructor

Method	**Task**	**Implementation Details**
`createStudent()`	To save a new student	- Calls `repo.save(s)` - Returns the saved `Student` object
`getStudent()`	To retrieve a student by ID	- Calls `repo.findById(id)` - Returns `Optional<Student>`
`updateStudent()`	To update existing student details by ID	- Calls `repo.findById(id)` and throws `NotFoundException` if student not found with the message: “Student with id " + id + " not found" - Updates `name`, `email`, and `program` - Saves and returns the updated student using `repo.save(existing)`

2. Controller Class - Method Descriptions

A. CourseController – Method Descriptions

- Declare a private variable named `service` of type `CourseService` and inject it via constructor-based dependency injection.

Method	Task	Implementation Details
`private final CourseService service;`	Declares the service to handle course operations	- Final field - Injected via constructor

Method	Task	Implementation Details
`create()`	To create a new course	- **Request type**: `POST`, **URL**: `/api/courses` - Accepts `Course` entity from request body - Calls `service.save(c)` - Returns created course wrapped in `ResponseEntity.ok()`
`get()`	To fetch course and its student details by ID	- **Request type**: `GET`, **URL**: `/api/courses/{id}` - Uses `@PathVariable` to get `id` - Calls `service.getCourseWithStudent(id)` - Returns `CourseResponse` object wrapped in `ResponseEntity.ok()`
`updateDepartment()`	To update an existing course by ID	- **Request type**: `PUT`, **URL**: `/api/courses/{id}` - Accepts `Course` from request body - Calls `service.update(id, c)` - Returns updated course wrapped in `ResponseEntity.ok()`

B. StudentController – Method Descriptions

- Declare a private variable named `service` of type `StudentService` and inject it via constructor-based dependency injection.

Method	Task	Implementation Details
`private final StudentService service`	Declares the service to handle student operations	- Final field - Injected via constructor

Method	Task	Implementation Details
`create()`	To create a new student	- **Request type**: `POST`, **URL**: `/api/students` - Accepts `Student` entity from request body - Calls `service.createStudent(s)` - Returns created student wrapped in `ResponseEntity.ok()`
`get()`	To fetch a student by ID	- **Request type**: `GET`, **URL**: `/api/students/{id}` - Uses `@PathVariable` to get `id` - Calls `service.getStudent(id)` - If student is present, returns it wrapped in `ResponseEntity.ok()` - If not, throws `NotFoundException` with message: `"Student with id " + id + " not found"`
`update()`	To update an existing student by ID	- **Request type**: `PUT`, **URL**: `/api/students/{id}` - Accepts `Student` from request body - Calls `service.updateStudent(id, s)` - Returns updated student wrapped in `ResponseEntity.ok()`

7 Microservices Communication

Communication among the microservices needs to be achieved by using **RestTemplate**. A RestTemplate bean is configured within the application, but you are required to implement the communication logic in the `CourseService.java` class to interact with the Student microservice.

- You are specifically required to configure the `RestTemplate` to fetch **Student details by Student ID** from the **Student Microservice**.
- The expected endpoint to communicate with is: http://student-service/api/students/{id}

8 Rest Endpoints

Rest Endpoints to be exposed in the controller along with method details for the same to be created

a. StudentController

URL Exposed		Purpose
1. /api/students		Creates a new Student
Http Method	POST **The student data to be created must be received in the controller using @RequestBody.**	
Parameter 1	`Student` Entity	
Return	`Student` Entity	
2. /api/students/{id}		Retrieves Student details by ID
Http Method	GET	
Path Variable	Long id	
Return	`Student` Entity	
3. /api/students/{id}		Updates Student details by ID
Http Method	PUT **The updated student data must be received**	

	using @RequestBody.	
Path Variable	Long id	
Parameter	`Student` Entity	
Return	`Student` Entity	

b. COURSECONTROLLER

URL Exposed		Purpose
1. /api/courses		Creates a new Course
Http Method	POST **The course data must be received using @RequestBody.**	
Parameter 1	`Course` Entity	
Return	`Course` Entity	
2. /api/courses/{id}		Retrieves Course and associated Student details by Course ID
Http Method	GET	
Path Variable	Long id	
Return	`CourseResponse` DTO (includes `Course + Student`)	
3. /api/courses/{id}		Updates Course details by ID
Http Method	PUT **The updated course data must be received using @RequestBody.**	
Path Variable	Long id	
Parameter	`Course` Entity	
Return	`Course` Entity	

9 Sequence to Execute

The sequence has to be followed for step 10 for every microservice are given below:

- eureka-naming-server
- course-micro-service
- student-micro-service

**Strictly follow the above sequence to follow.

10 Execution Steps to Follow

1. **All actions like build, compile, running application, running test cases will be through Command Terminal.**
2. **To open the command terminal the test takers need to go to the Application menu (Three horizontal lines at left top) -> Terminal -> New Terminal.**
3. **cd into your backend project folder**
4. **To build your project use command:**

   mvn clean package -Dmaven.test.skip

5. **To launch your application, move into the target folder (cd target). Run the following command to run the application:**

   java -jar <your application jar file name>

6. **This editor Auto Saves the code.**
7. **These are time bound assessments the timer would stop if you logout and while logging in back using the same credentials the timer would resume from the same time it was stopped from the previous logout.**
8. **To test any Restful application, the last option on the left panel of IDE, you can find ThunderClient, which is the lightweight equivalent of POSTMAN.**
9. **To test any UI based application the second last option on the left panel of IDE, you can find Browser Preview, where you can launch the application.**

10. Default credentials for MySQL:

    a. Username: root
    b. Password: pass@word1

11. To login to mysql instance: Open new terminal and use following command:

    a. sudo systemctl enable mysql
    b. sudo systemctl start mysql

       NOTE: After typing any of the above commands you might encounter any warnings.

       >> Please note that this warning is expected and can be disregarded. Proceed to the next step.

    c. mysql -u root -p

       The last command will ask for password which is ‘pass@word1’

12. Mandatory: Before final submission run the following command:

    mvn test