To be RESTful

Developing a quality API

Summary:
In this project you will learn how to use Spring mechanisms to develop REST applications

Version: 1.00

Contents

Chapter I

Preamble

Levels of compliance of an application's `API` to the `REST` architecture as per `Richardson model` (each subsequent level is based on the previous one):

- `Level 0`. `HTTP` is used as a transport protocol. A single `URI` is used for all interactions. All required information is passed in plain `XML` text.

- `Level 1`. The `API` uses a "resource" concept. Each resource is a separate business object. Each resource has its own `URI`. All interactions are described by a single `HTTP` verb.

- `Level 2`. All interactions are described by an extended set of `HTTP` verbs: `GET` (getting an entity), `POST` (adding a new entity), `PUT` (updating an existing entity), and `DELETE`. Hence, a `CRUD` is defined for each resource.

- `Level 3`. The `API` is based on `Hypermedia` format.

Chapter II

General Rules

- Use this page as the only reference. Do not listen to any rumors and speculations about how to prepare your solution.
- You mus use the latest LTS version of Java. Make sure that compiler and interpreter of this version are installed on your machine.
- You must use GraalVM to run your code.
- You can use IDE to write and debug the source code (we recommend IntelliJ Idea).
- The code is read more often than written. Read carefully the document where code formatting rules are given. When performing each exercise, make sure you follow the generally accepted Oracle standards
- Pay attention to the permissions of your files and directories.
- To be assessed, your solution must be in your GIT repository.
- You should not leave in your directory any other file than those explicitly specified by the exercise instructions. It is recommended that you modify your .gitignore to avoid accidents.
- When you need to get precise output in your programs, it is forbidden to display a precalculated output instead of performing the exercise correctly.
- Have a question? Ask your neighbor on the right. Otherwise, try with your neighbor on the left.
- Your reference manual: mates / Internet / Google. And one more thing. There's an answer to any question you may have on Stackoverflow. Learn how to ask questions correctly.
- Read the examples carefully. They may require things that are not otherwise specified in the subject.
- Use "System.out" for output
- And may the Force be with you!
- Never leave that till tomorrow which you can do today ;)

Chapter III

Rules of the project

- The solution for each exercise is a standalone `Maven` project implemented on the basis of `Spring Boot`.
- Project structure is at a developer's discretion.
- Each project shall contain a data.sql file with a set of test data.

Chapter IV

Exercice 00: REST API

	Exercise 00
REST API	
Turn-in directory : *ex*00/	
Files to turn in : `Education Center-folder`	
Allowed functions : `n/a`	

You need to develop an `API` to manage a training center. A set of operations shall be implemented for the following domain models:

- User
    - First name
    - Last name
    - Role (Administrator, Teacher, Student)
    - Login
    - Password
- Course
    - Start date
    - End date
    - Name
    - Teachers
    - Students
    - Description
    - Lessons
- Lesson

- Start time
- End time
- Day of week
- Teacher

The task shall be completed in accordance with `REST API` requirements, for example:

- Adding a new lesson to a course with ID 42:

  `POST /courses/42/lessons`

  Request body:

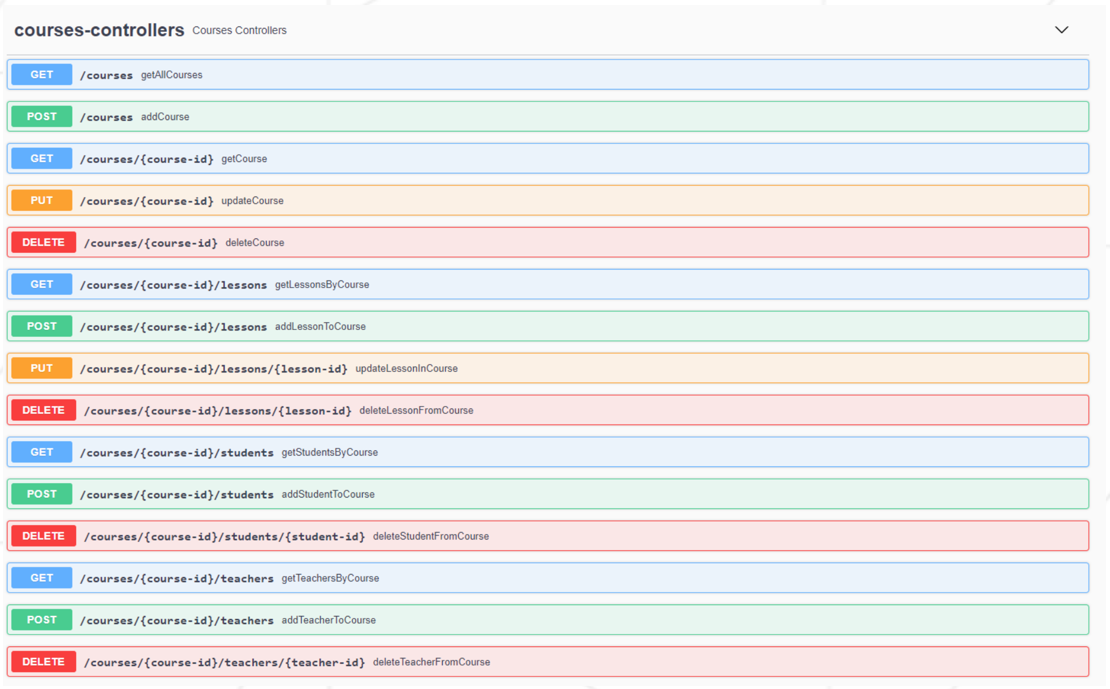
```
{
    "startTime" : "10:00",
    "finishTime" : "12:00",
    "dayOfWeek" : "Monday",
    "teacherId" : 432
}
```

  Response body:

```

"lesson": {
    "id" : 21,
    "startTime" : "10:00",
    "finishTime" : "12:00",
    "dayOfWeek" : "Monday",
    "teacher" : {
        "id" : 432,
        "firstName" : "Best",
        "lastName" : "Teacher"
    }
}
```

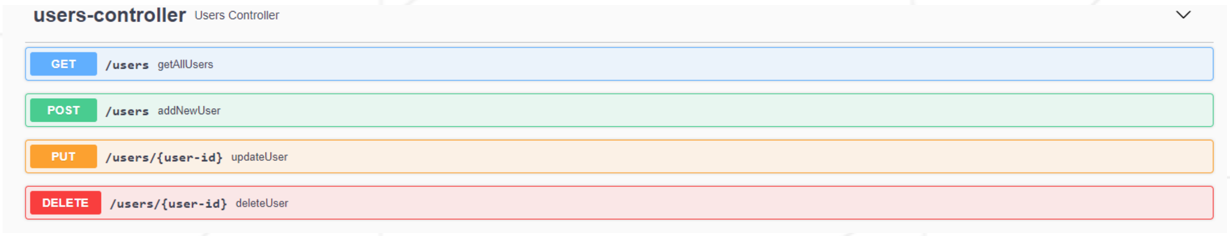
Below is a complete list of operations for the courses that need to be implemented as part of the current task:



courses-controllers Courses Controllers

GET `/courses` getAllCourses
POST `/courses` addCourse
GET `/courses/{course-id}` getCourse
PUT `/courses/{course-id}` updateCourse
DELETE `/courses/{course-id}` deleteCourse
GET `/courses/{course-id}/lessons` getLessonsByCourse
POST `/courses/{course-id}/lessons` addLessonToCourse
PUT `/courses/{course-id}/lessons/{lesson-id}` updateLessonInCourse
DELETE `/courses/{course-id}/lessons/{lesson-id}` deleteLessonFromCourse
GET `/courses/{course-id}/students` getStudentsByCourse
POST `/courses/{course-id}/students` addStudentToCourse
DELETE `/courses/{course-id}/students/{student-id}` deleteStudentFromCourse
GET `/courses/{course-id}/teachers` getTeachersByCourse
POST `/courses/{course-id}/teachers` addTeacherToCourse
DELETE `/courses/{course-id}/teachers/{teacher-id}` deleteTeacherFromCourse

Adding teachers and students to a course involves sending only a user ID in request body (as opposed to adding a lesson, where sending complete information is required). Full-scale work with User entity is performed in a separate controller:

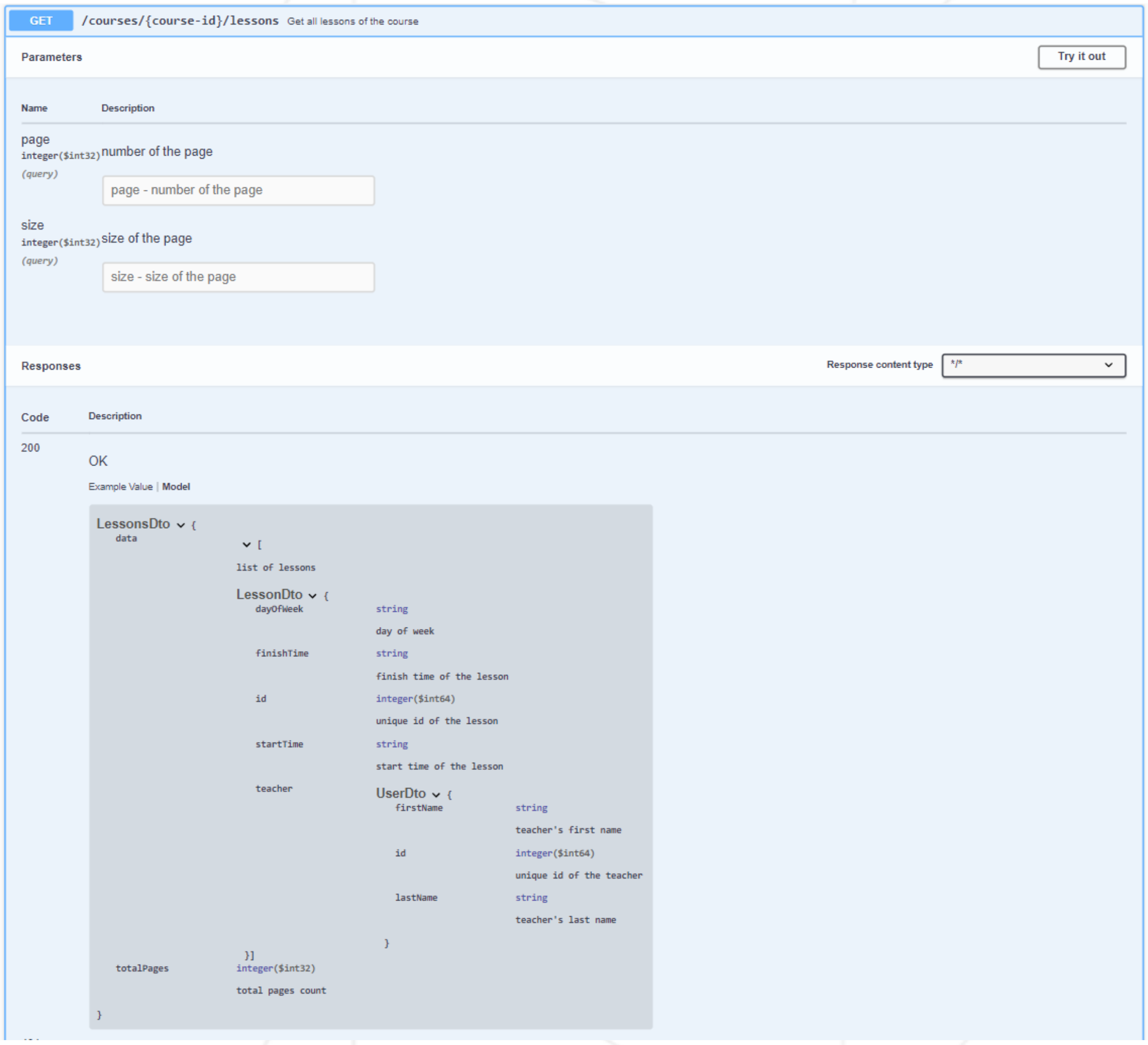


users-controller Users Controller

GET	`/users`	getAllUsers
POST	`/users`	addNewUser
PUT	`/users/{user-id}`	updateUser
DELETE	`/users/{user-id}`	deleteUser

Additional requirements:

- Each method that retrieves a collection of objects shall support pagination mechanism.
- For the convenient use of `API`, you need to integrate `Swagger` framework into your application: https://swagger.io/.
- For each method, you also need to provide documentation using `Swagger`, for example:



GET `/courses/{course-id}/lessons` Get all lessons of the course

Name	Description
page integer($int32) *(query)*	number of the page page - number of the page
size integer($int32) *(query)*	size of the page size - size of the page

Code	Description
200	OK Example Value | **Model**

```
LessonsDto ⌄ {
   data                  ⌄ [
                         list of lessons
                         LessonDto ⌄ {
                            dayOfWeek         string
                                              day of week
                            finishTime        string
                                              finish time of the lesson
                            id                integer($int64)
                                              unique id of the lesson
                            startTime         string
                                              start time of the lesson
                            teacher           UserDto ⌄ {
                                                 firstName       string
                                                                 teacher's first name
                                                 id              integer($int64)
                                                                 unique id of the teacher
                                                 lastName        string
                                                                 teacher's last name
                                              }
                          }]
   totalPages            integer($int32)
                         total pages count
}
```

You need to implement unit test for at least one GET, POST, PUT, and DELETE method using MockMvc, for example:

```
@ExtendWith(SpringExtension.class)
@AutoConfigureMockMvc
@SpringBootTest
public class CoursesTest {

    @Autowired
    private MockMvc mockMvc;

    @MockBean
    private CoursesService coursesService;

    @BeforeEach
    public void setUp() {
        when(coursesService.delete(1L)).then...;
    }

    @Test
    public void deleteCourseTest() throws Exception {
        mockMvc.perform(delete("/courses/1")).andDo(print())
                .andExpect(status().isOk())
                .andExpect(jsonPath(...));
    }
}
```

In case of incorrect operations (adding a non-existent teacher/student to the course, indicating an incorrect user role, etc.), you shall return a response with code 400 and the following content:

```
"error": {
    "status" : 400,
    "message" : "Bad request"
}
```

Spring Data REST is not allowed for this task.

Chapter V

Exercice 01: JWT

	Exercise 01
JWT	
Turn-in directory : *ex*01/	
Files to turn in : `Education Center-folder`	
Allowed functions : `n/a`	

Use `JWT` authorization to implement a mechanism for providing role-based access to resources.

Now, every request shall be accompanied by a `JWT` token with the following user information:

- User ID
- User role
- User login

Upon authorization, a user sends a `POST` request to `/signUp URL` with a login and a password. If this data is correct, a user receives a `JWT` token signed with a secret key. The key is stored in `application.properties` file of the application. If authorization data is incorrect, `403 status` shall be returned.

Each user request to `API` resources shall have an Authorization header with a custom `JWT` token. In this case, an application shall not access the database to authorize a user because all necessary information is stored in a token.

`GET` operations are available for a user with any role. `POST`, `PUT` and `DELETE` operations are only available to the center's administrator.

`To fully implement authentication using JWT, you need to implement Spring Security components: Authentication, Filter, AuthenticationProvider.`

`Spring Data REST is not allowed for this task`

Chapter VI

Exercice 03 : HATEOAS

	Exercise 03
HATEOAS	
Turn-in directory : *ex*03/	
Files to turn in : `Education Center-folder`	
Allowed functions : `n/a`	

Now, let us implement the functionality of a training center using `Spring Data REST` technology. Thus, the entire `API` for `User`, `Course`, and `Lesson` entities will be presented in `Hipermedia` format. E.g., for a `/course GET` request, the following response will be returned:

```
{
    "_embedded": {
        "courses": [
            {
                "title": "Spring Data Rest",
                "description": "Best framework",
                "state": "Published",
                "\_links": {
                    "self": {
                        "href": "http://localhost/courses/1"
                    },
                    "course": {
                        "href": "http://localhost/courses/1"
                    },
                    "lessons": {
                        "href": "http://localhost/courses/1/lessons"
                    },
                    "students": {
                        "href": "http://localhost/courses/1/students"
                    }
                }
            },
            {
                "title": "SQL",
                "description": "All about RDBMS",
                "state": "Draft",
                "_links": {
                    "self": {
                        "href": "http://localhost/courses/2"
                    },
                    "course": {
                        "href": "http://localhost/courses/2"
                    },
                    "publish": {
```

```
                    "href": "http://localhost/courses/2/publish"
                },
                "lessons": {
                    "href": "http://localhost/courses/2/lessons"
                },
                "students": {
                    "href": "http://localhost/courses/2/students"
                }
            }
        }
    ]
},
"_links": {
    "self": {
        "href": "http://localhost/courses"
    },
    "profile": {
        "href": "http://localhost/profile/courses"
    },
    "search": {
        "href": "http://localhost/courses/search"
    }
},
"page": {
    "size": 20,
    "totalElements": 2,
    "totalPages": 1,
    "number": 0
}
}
```

As you can see from the example, for Course entity you need to implement the ability to publish with `/courses/2/publish POST` request.
A course in the `DRAFT` state may be published. Once published, it is switched to `PUBLISHED` state and cannot be re-published.

`An ability to work with API through HAL Browser shall be provided.`

You also need to provide auto-generation of adoc documentation for the course publishing method based on a unit test of this method.

An example of such documentation:

Course API

Methods

Course publish

You can publish a course with status **DRAFT**

request

```
PUT /courses/1/publish HTTP/1.1
Host: localhost:8080
```

response

```
HTTP/1.1 200 OK
Vary: Origin
Vary: Access-Control-Request-Method
Vary: Access-Control-Request-Headers
Content-Type: application/hal+json
Content-Length: 93

{
  "title" : "Spring 5",
  "description" : "All about Spring",
  "state" : "PUBLISHED"
}
```

Table 1. response-fields

Path	**Type**	**Description**
`title`	`String`	Title of course
`description`	`String`	Description of course
`state`	`String`	State of course