Representational State Transfer (REST)

License

Contents

Introduction

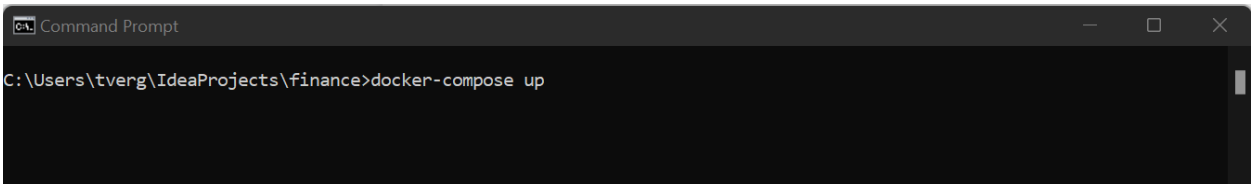
The goal of this week's lab is to implement a REST client as part of our student microservice. This will enable us to get information from the Finance microservice to find out whether a student has an outstanding balance. We will also be able to send an invoice to Finance to charge students for courses enrolled. We are going to leverage Spring Web's RestTemplate functionality to send requests and process responses in a very easy and intuitive way, with no String manipulation or JSON parsing!

The Finance Microservice

Before we start our development, let's make sure that the Finance microservice is up and running.

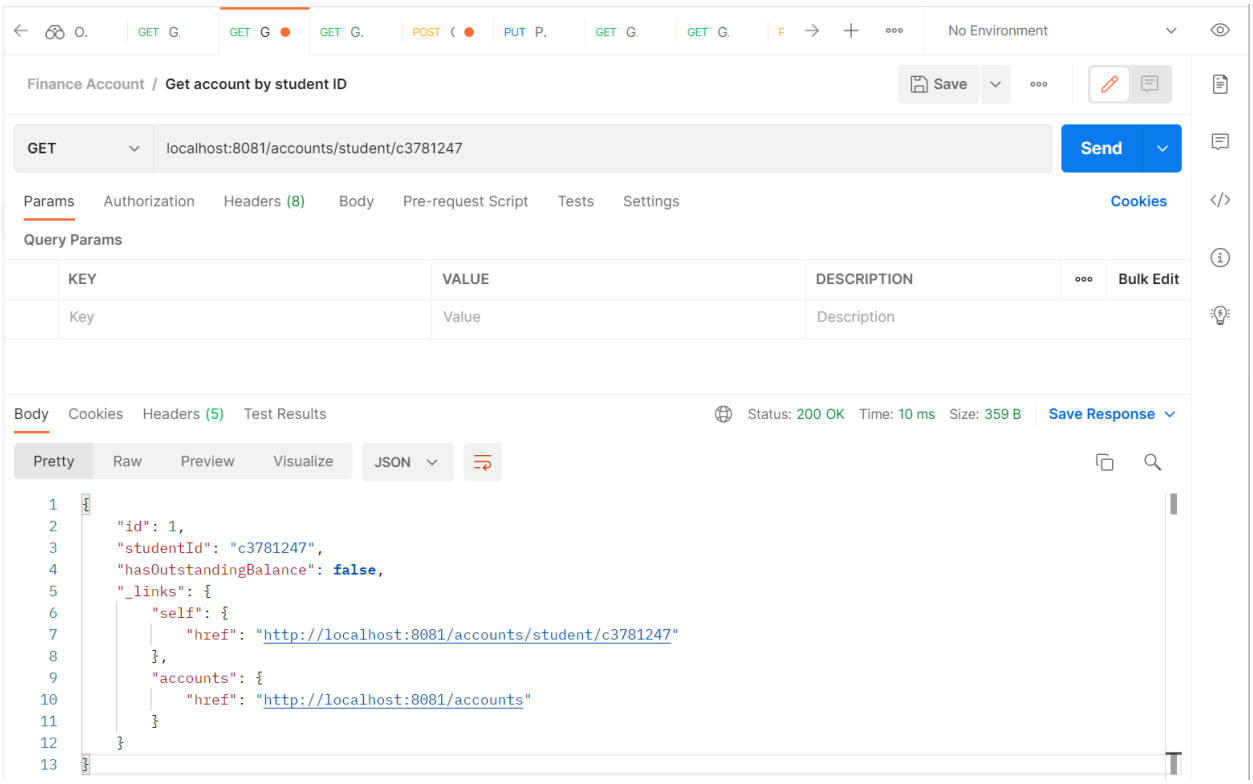
From the root of your Finance project, launch a terminal instance and run

docker-compose up



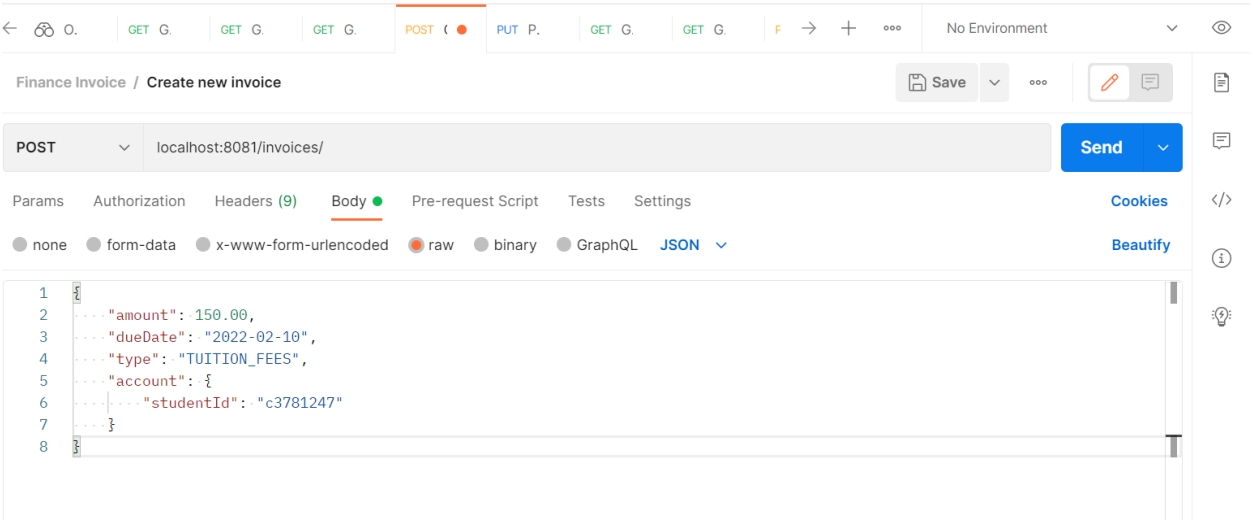
Launching the Finance microservice

Once the service is up and running, use your client of choice to send a GET request for a specific account, passing the student ID. We are going to send a request just like this programmatically, from our student service. The response will tell us whether the student has an outstanding balance.



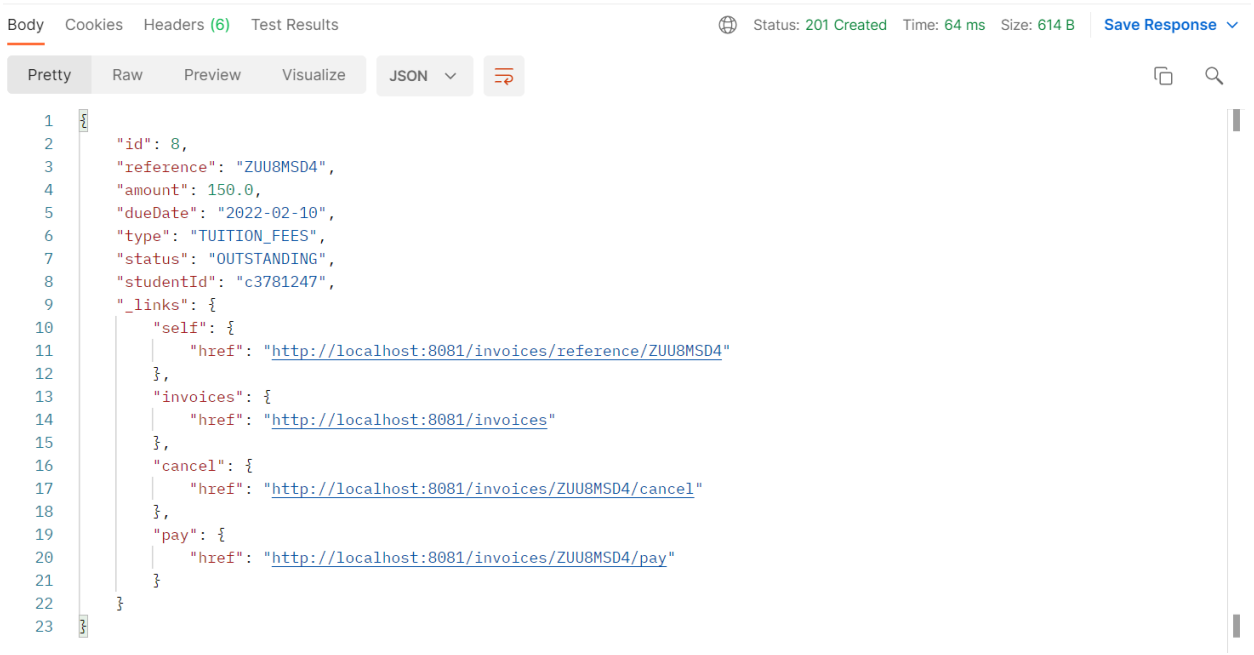
Using Postman to send a GET request to the Finance microservice

Now send a POST request to the "/invoice" endpoint to create a new invoice for tuition fees. In this lab, we will also learn how to do this programmatically so a student can be charged for courses they enrol in.



Using Postman to send a POST request to the Finance microservice

You should get a response like the one below.



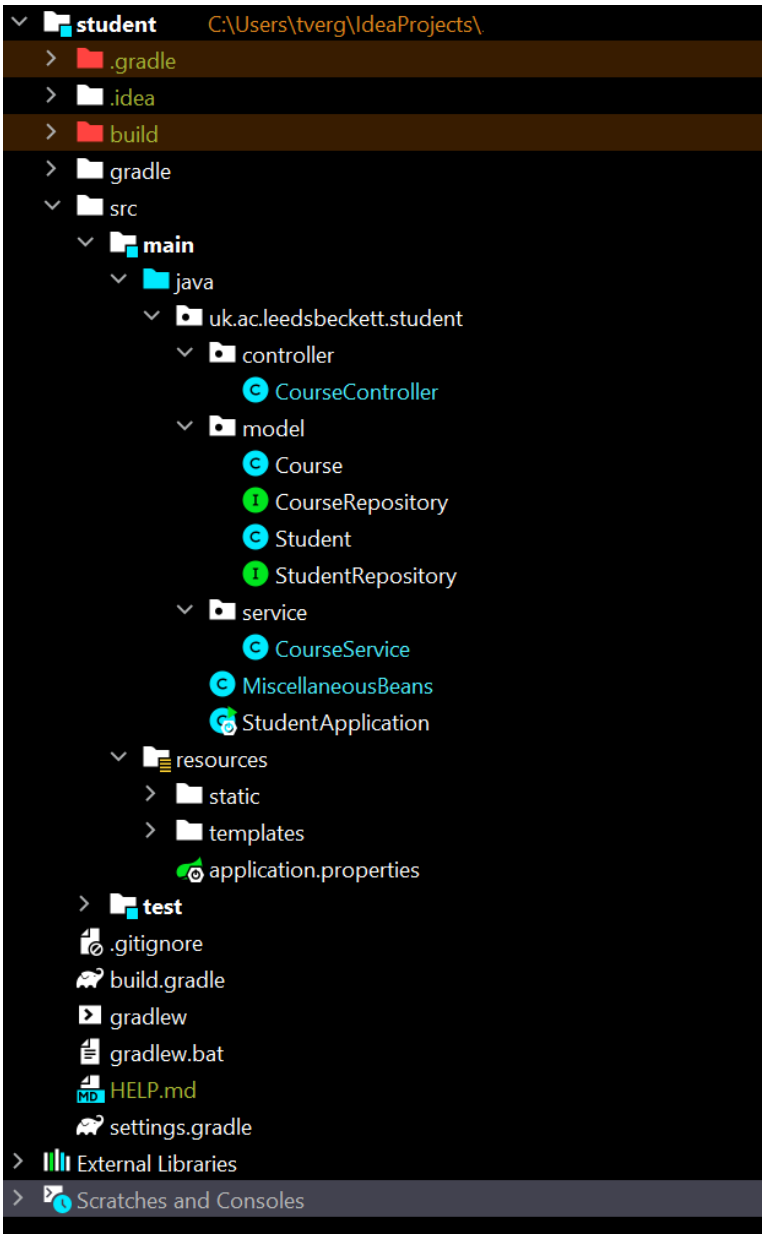
Response from the Finance service showing a successful invoice created

Answer in your log book: looking at the response, how can you tell the request was processed successfully?

In your log book: explain the purpose of the "_links" element at the end of the JSON object received.

Implementing an Integration Service

We are going to build upon the project we created in Week 3. Open your project in your IDE of choice and check that you have at least the structure shown below.



Student project from Week 3

In your service directory, create another service called IntegrationService.

Annotate it as a @Component and add a private instance of RestTemplate using constructor injection.

```java
package uk.ac.leedsbeckett.student.service;

import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

@Component
public class IntegrationService {

    private final RestTemplate restTemplate;

    public IntegrationService(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }
}

```

IntegrationService with an instance of RestTemplate provided through Spring's Dependency Injection

Your IDE may highlight the restTemplate constructor argument to say it cannot autowire, as there are no beans of type RestTemplate available. Let's make sure a bean is provided to Spring so it can find it and inject it.

Find your MiscellaneousBeans configuration class you created last week and add the following bean:

```java

    @Bean
    public RestTemplate restTemplate(RestTemplateBuilder builder) {
        return builder.build();
    }
```

Adding a RestTemplate bean to MiscellaneousBeans so Spring can inject it into our IntegrationService

You may notice that the red underlining has disappeared.

Before we write a method to call the Finance service, we need to have a strategy for dealing with the response that is going to be sent back to us. If you recall from this week's lecture, Spring's RestTemplate can map the JSON response straight into an

object for us, which is ideal as we don't want to have to parse the String manually. Let's create a domain class to be used for this purpose.

In your model directory, create a class called Account and add the following properties.

```java
package uk.ac.leedsbeckett.student.model;

import lombok.Data;

@Data
public class Account {
    private Long id;
    private String studentId;
    private boolean hasOutstandingBalance;
}

```

Account domain class used to map the response sent by the Finance service

We are now ready to implement our first REST call. Edit the IntegrationService and add a getStudentAccount() implementation, following the example below.

```java
package uk.ac.leedsbeckett.student.service;

import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;
import uk.ac.leedsbeckett.student.model.Account;

@Component
public class IntegrationService {

    private final RestTemplate restTemplate;

    public IntegrationService(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    public Account getStudentAccount(String studentId) {
        return restTemplate.getForObject( url: "http://localhost:8081/accounts/student/" + studentId, Account.class);
    }
}

```

IntegrationService showing implementation of getStudentAccount() method

Deceivingly simple?

In your log book: how do you know that this works? Write some code that calls this service to demonstrate that the implementation works as expected.

In your log book: how does this compare to using SOAP to implement the same functionality?

Additional challenge: the hardcoded URL is ugly and difficult to maintain. Can you think of a better way of handling it?

Implementing Student Enrolment into a Course

Our next challenge is to send a POST request to the Finance microservice to create an invoice upon enrolment. First things first: we need to be able to enrol a student in a course.

The student class we created on Week 4 was pretty rudimentary. We let Lombok implement a getter and a setter for the set of Courses a student is enrolled in when, in reality, we very rarely change the entire collection at once. A much better approach is to handle the enrolment course by course.

Edit your student domain model class to support enrolment in a single course (follow the example below).

You may have noticed that we have added a couple more annotations to the coursesEnrolledIn property. @EqualsandHashcode.Exclude and @ToString.Exclude instruct Lombok to exclude this field from its default implementations (which prevents persistence issues with Hibernate).

```java
@Entity
@Data
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="id")
    private Long id;
    @Column(unique = true)
    private String externalStudentId;
    private String surname;
    private String forename;
    @ManyToMany(cascade = CascadeType.ALL)
    @JoinTable(
            name = "course_student",
            joinColumns = @JoinColumn(name = "student_id"),
            inverseJoinColumns = @JoinColumn(name = "course_id"))
    @EqualsAndHashCode.Exclude
    @ToString.Exclude
    Set<Course> coursesEnrolledIn;

    public void enrolInCourse(Course course) {
        if (coursesEnrolledIn == null) {
            coursesEnrolledIn = new HashSet<>();
        }
        coursesEnrolledIn.add(course);
    }
}

```

Enhanced Student domain model class

Edit the MiscellaneousBeans class to ensure the seed data uses our new enrolment method.

You may need to resolve the circular reference between student and course at this point. One way of doing this is to add the @JsonIgnore annotation to one side of the relationship.

Course.java

```java
package uk.ac.leedsbeckett.student.model;

import com.fasterxml.jackson.annotation.JsonIgnore;
import lombok.Data;
import lombok.ToString;

import javax.persistence.*;
import java.util.Set;

@Entity
@Data
public class Course {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;
    private String description;
    private Double fee;
    @ManyToMany(mappedBy = "coursesEnrolledIn")
    @JsonIgnore
    @ToString.Exclude
    Set<Student> studentsEnrolledInCourse;
}

```

Course domain class showing @JsonIgnore annotation on line 20

Now we are ready to create an EnrolmentService, which will be used by our application to enrol students in courses. Follow the example below.

EnrolmentService.java

```java
package uk.ac.leedsbeckett.student.service;

import org.springframework.stereotype.Component;
import uk.ac.leedsbeckett.student.model.Course;
import uk.ac.leedsbeckett.student.model.Student;
import uk.ac.leedsbeckett.student.model.StudentRepository;

@Component
public class EnrolmentService {
    private final StudentRepository studentRepository;

    public EnrolmentService(StudentRepository studentRepository) {
        this.studentRepository = studentRepository;
    }

    public void enrolStudentInCourse(Student student, Course course) {
        student.enrolInCourse(course);
        studentRepository.save(student);
    }
}
```

EnrolmentService implementation

Note how the StudentRepository is added as a constructor parameter, to be injected by Spring.

All we are doing in this very simple implementation is calling the enrolInCourse() method from the student instance and getting the repository to save the record in the database.

Finally, add some seed data.

```
Student thalita = new Student();
thalita.setForename("Thalita");
thalita.setSurname("Vergilio");
thalita.setExternalStudentId("c9999999");
thalita.enrolInCourse(sesc);
thalita.enrolInCourse(rema);

Student duncan = new Student();
duncan.setForename("Duncan");
duncan.setSurname("Mullier");
duncan.setExternalStudentId("c2222222");
duncan.enrolInCourse(sesc);
duncan.enrolInCourse(ase);
```

Seed data using the new enrolInCourse() method

In your log book: what if the student was already enrolled in that particular course? Add an if statement that checks for this condition and performs an appropriate action (hint: use the repository).

Modifying the Integration Service to Post Invoices

We can now enrol a Student in a Course, but we can't yet charge them any fees. We need to be able to create invoices and send them to the Finance service. Let's do this now.

Create a new domain model class called Invoice. Follow the example below.

Invoice.java

```java
package uk.ac.leedsbeckett.student.model;

import lombok.Data;

import java.time.LocalDate;

@Data
public class Invoice {
    private Long id;
    private String reference;
    private Double amount;
    private LocalDate dueDate;
    private Type type;
    private Status status;
    private Account account;

    public enum Type {
        LIBRARY_FINE,
        TUITION_FEES
    }

    enum Status {
        OUTSTANDING,
        PAID,
        CANCELLED
    }
}
```

Invoice domain class used to map the data to send to the Finance service

In your IntegrationService, add a method that posts an invoice to the Finance service.

```java
@Component
public class IntegrationService {

    private final RestTemplate restTemplate;

    public IntegrationService(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    public Account getStudentAccount(String studentId) {
        return restTemplate.getForObject(url: "http://localhost:8081/accounts/student/" + studentId, Account.class);
    }

    public Invoice createCourseFeeInvoice(Invoice invoice) {
        return restTemplate.postForObject(url: "http://localhost:8081/invoices/", invoice, Invoice.class);
    }
}
```

Method to post a course fee invoice to the Finance service

Note how the actual REST implementation is extremely simple. One line of code is all it takes 🙄 !

The final task is to edit the EnrolmentService and make it call the createCourseFeeInvoice() method upon enrolment of a student into a course. This task is up to you to implement.

Hint: you will need to create an account as well as an invoice, since the student ID belongs to the account. Make sure you use a student ID that exists in the Finance database when testing.

Another hint: you don't need to populate all the properties of your account and invoice, just the ones that the Finance API needs. Use the Postman request from earlier to guide you.

Troubleshooting: if you mess up the Finance database by trying to insert invalid records, you may need to rebuild the containers. Stop the running containers and run "docker system prune --volumes" and "docker system prune --all" to clean everything up. Then launch the services again with "docker-compose up".

In your log book: how do you know that this code works? Demonstrate that it does, either by writing some seed data, or by using Postman, or a combination of both.

Assignment

Follow the examples we have worked through and implement sending POST requests to the Finance and Library services when a student account is created.

Conclusion

At this point you have:

- gained theoretical and practical knowledge of REST, and reflected on the advantages and disadvantages of using it when compared to SOAP;
- learned how to leverage Spring's RestTemplate functionality to send GET and POST request programmatically;
- worked independently to integrate all three services upon creation of a student account;
- progressed with your assignment project, and implemented all the integrations required.

Well done!

Writing a client to interact with a RESTful API programmatically may seem simple. Writing it well, however, requires adherence to best practices and specialised technical knowledge of the latest and most advanced tools in the industry, such as Spring Web. These skills are highly valued by employers, and will set you apart when applying for engineering jobs in the future!