VIETNAM NATIONAL UNIVERSITY OF HOCHIMINH CITY
THE INTERNATIONAL UNIVERSITY
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

A SYSTEM FOR STUDENT LEARNING PROGRESS MANAGEMENT WITH EVENT DRIVEN MEDIATOR PATTERN

By
LY BAO THOAI

A thesis submitted to the School of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
Bachelor of Information Technology/Computer Science/Computer Engineering

Ho Chi Minh City, Vietnam
Year

A SYSTEM FOR STUDENT LEARNING PROGRESS MANAGEMENT WITH EVENT DRIVEN MEDIATOR PATTERN

APPROVED BY:

________________________________ ,
(*Type Committee names beneath lines*)

____
(*Typed Committee name here*)

____
(*Typed Committee name here*)

____
(*Typed Committee name here*)

____
(*Typed Committee name here*)

THESIS COMMITTEE
(Whichever applies)

ACKNOWLEGMENTS

I want to express my sincere gratitude to Dr. Tran Thanh Tung, whom I cannot thank enough. It was under his experienced and passionate guidance along the way that I could begin and complete this work. I am also grateful for the help from Nguyen Phan Hoang Tu, an earlier student, who has given me support in both technical and informational regarding his undergraduate thesis. And I want to thank the faculty of the School of Computer Science of the International University for giving me the opportunity to study under professional conditions. Finally, acknowledgement is also spread to my reading and examination committee for dedicating to this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Event-driven architecture is becoming more relavent in microservice projects. What has been done prior is merely replacing the synchronous restful requests with topics. However, as the system scales, the mediator topology is more suitable for handling complex service-to-event wiring networks. Thus, the main goal of this thesis is to apply the *mediator pattern* into *mananging student learning progress*, that is, monitoring different stages of the students in many areas in the university. It is done by using Kafka as a messaging broker, with Nodejs running as the back-end services and Reactjs for front-end web application. Together, the project has brought about flexibility, resilience and estensibility of the whole system. Experiments to evalutate the mediator topology has also been carried out with an increase in time delay result, meaning this pattern should not be used anywhere but only on systems that benefit from this. Apart from it, from this thesis, other rooms for future research can be possible and they are needed to enhance the ease of development or to create related apps.

CHAPTER 1

INTRODUCTION

1.1. Motivation

The mediator pattern is a great way to manage complex flows of the services in event-driven paradigms. Besides, the current need for our system to manage university students' studying progresses matches with the benefits that the design delivers.

When microservices became a popular architecture for developing an application with independent pieces, event-driven is a way to abstract the interactivities between them. By using topics and queues, the communications are more responsive and fault tolerance, that was what we have achieved so far in the previous projects. Despite the flexibility that the event-driven approach has brought, we also need a way to orchestrate the processes on a higher level. For instance, an ability to control and monitor which event goes to which service or what topics should it forward the messages next. From researching, *mediator* is one of the design patterns in event-driven that are perfect for such task, that is managing complex flow logic between services especially in which there are procedural steps need to be executed in order. This works by implementing another service whose responsibility is only listening and forwarding messages. Additionally, it can have its own storage of the states of a particular progress and execute accordingly.

On the other side, at the moment, the students in our university here are lacking a central system to manage all of the learning progresses since all of the information are currently in different apps like Edusoftweb, Blackboard, Outlook, and IUOSS. Those progresses are also viewed as different small steps, for example, the semester progress of the students includes registering courses, confirmation, payment, examination and waiting for results. In event-driven, it is troublesome to manage the execution orders of services (student service, course service, payment service, …) but by applying the mediator pattern, we can simply delegate all of the complex flow logics to the mediator sitting in the middle. Another kind of progress is the general progress, or the overall status of the students. It has steps like application, entry English exam, thesis, certificates and graduation; in which, parallel execution also required. Considering the step certificates, before moving to the final step, the mediator has to wait for not one but until all necessary conditions are met (all three certificates: English, political education, military), only then it can signal the processes to the next graduation step. This is not possible or at least complicated to implement without a mediator.

In addition, as the design pattern concerns with a large system, it is required to have a way to manage all of the services. In other words, we need to display all of the services on the screen with their statuses as well as which category they belong to. Another problem arises inside big systems is error handling, since there are a lot of places it could happen. So, an error handling pattern is also applied to this project, it has capability of resolving common defined errors or pushing them to the dashboard if the error is not known.

It is not only with the ability to handle complex flows of the system, but also managing services and error handlings. Therefore, the proposed patterns are suitable for this project. Thus, it is a good idea to build the system using the discussed methodologies.

1.2. Problem Statement

In this thesis, I want to apply the mediator pattern and related event-driven methods including services managing and error handling into the system for student learning progress management in addition with a user interface implementation to visualize the entire process.

1.3. Scope and Objectives

The objectives of the thesis include:

- Apply mediator pattern into tracking student learning progress
- Apply error handling pattern and a system to manage services.
- Implement a web app to display the data and relavent system info.

The scope of this thesis does not include finishing a complete system that can be used outright but rather a realization of the methodology to a practical scenario. For which, we have selected the scenario of managing the progress of the students in steps.

1.4. Structure of thesis

This is the summarized list of chapters to help following the report:

Chapter	Title	Content
1	Introduction	A brief explanation about motivation, problem, scope and objectives as well as assumptions for this thesis
2	Liturature review	Overview of the current technology and method utilized in this project, additionally reviewing a related thesis
3	Methodology	An architectural view of the solutions applied with design diagrams and explanations
4	Implementation and results	Detailed view (coding) for individual part of the system including setting up event driven queue, back-end and front-end for the implementation and some demo flows for the results
5	Evaluation and discussion	Assessing the performance of the project, discussing about the benefits and drawbacks of the solutions compared to other designs.
6	Conclusion and future work	Values of thesis and suggestion on further related development ideas

Table 1: Structure of thesis

CHAPTER 2

LITURATURE REVIEW/RELATED WORK

2.1. Background

To accurately understand and appreciate the effort put into this project, it is essential to look over some of the important definition about event-driven architecture as well as the patterns that lie behind it.

2.1.1. Event-driven architecture

The event-driven architecture is a loosy connected and extensible way of communication between microservices[1] in a system, although it is not really a new concept.
Traditionally, the simple way to communicate between services are restful requests, where the sender will wait until the request is responded (from A to B) and synchronous.

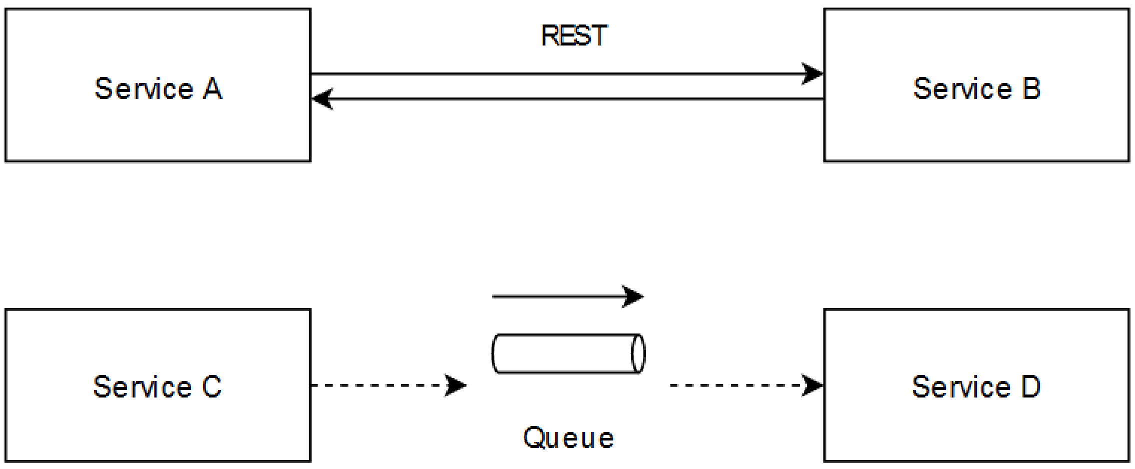


Figure 1: Restful versus event-driven communication

From C to D, however, leverages the queue to temporary stores the requests until they are read. This is event-driven type of communication and asynchronous.

The advantages of event-driven over rest are decouplization between services, higher responsives for users, scaling potential, and fault tolerance[2].

2.1.2. Mediator

Despite that all services are separated using events, each service is still depended on the topic that it publishes or subscribes, especially in a complex system with a long sequence that requires multiple services talk to each other. Each service will then have to be aware of all the existing topics out there. To solve this problem, Neal Ford and Mark Richards in their book *Fundamentals of Software Architecture*[3] discussed about the *Mediator Topology* in Chapter 14. The pattern includes a mediator, which stays in the middle and does the job of connecting all services and leading the flow:

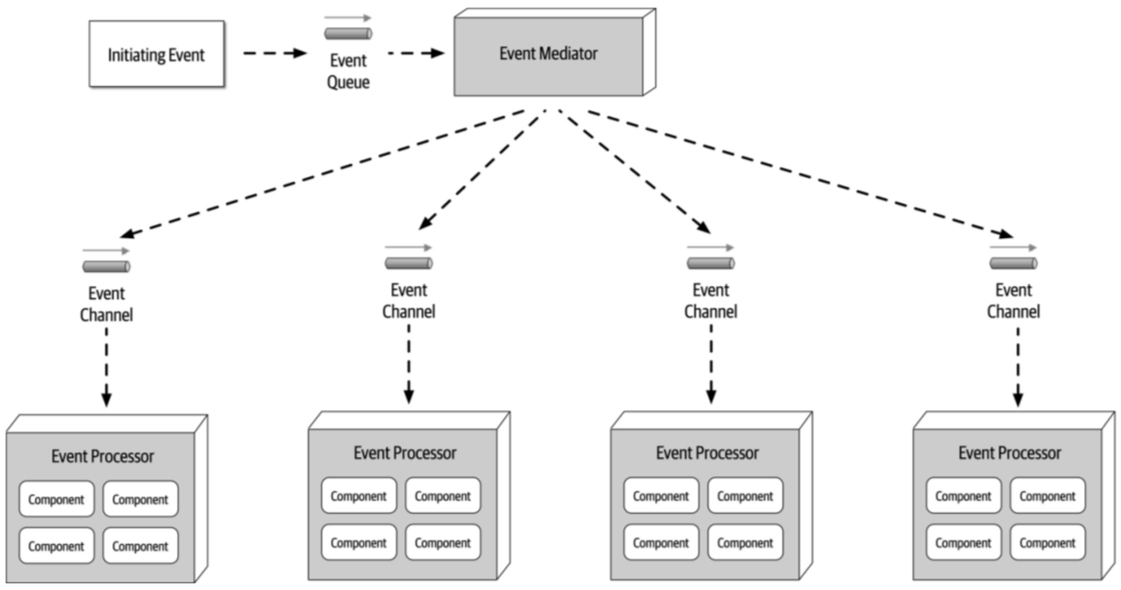


Figure 2: Mediator topology

Therefore, the event processors only subscribe/publish to their own channels, leading to lower couplings between them. Furthermore, the mediator can easily tweak the flow by changing the code inside without affecting other processors.

2.1.3. Error handling

In event-driven architecture, as more and more events being produced and consumed, errors are not uncommon. Therefore, a pattern for error handling is also introduced in the same book, which consists of a set of three event processors: the producer, consumer and workflow processor.

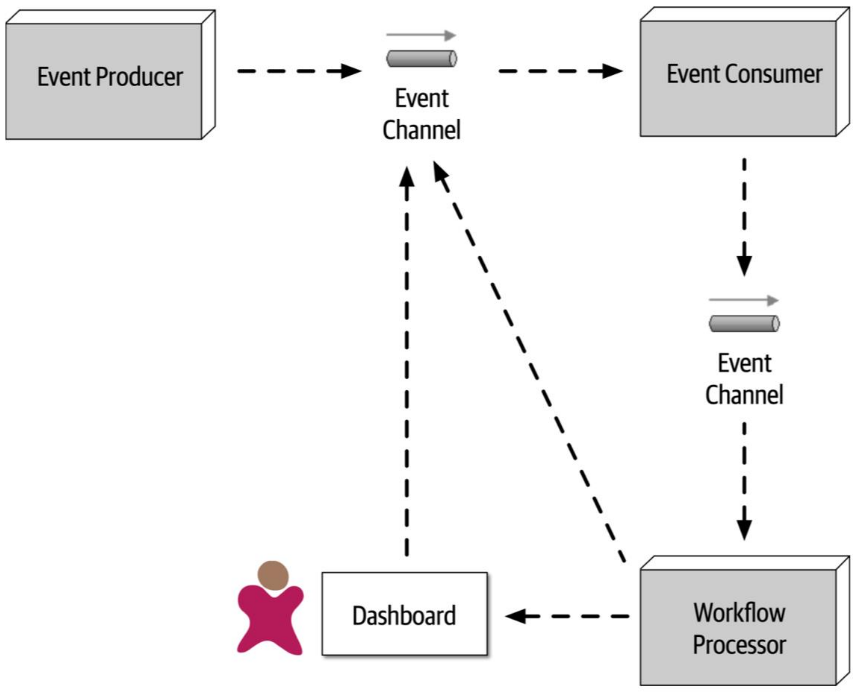


Figure 3: Error handling in event-driven architcture

In the beginning, after consuming event from the producer, the event consumer detected problems in the event, it will forward the message to workflow processor. That processor can then handle the error and send it back to the event channel at the start or display them to the dashboard and have it fixed manually.

2.2. Liturature review

Reviewing Nguyen Phan Hoang Tu's thesis, *System Integration for Student Academic Affairs Management with Event-driven Architecture*[4]. The system was built under event-driven methodology with Apache Kafka is the medium for microservice communications. There are student, course, mail, retention, academic schedule and event central services implemented using Spring Boot along with API Gateway, Netflix Eureka and Spring Cloud Config Server.

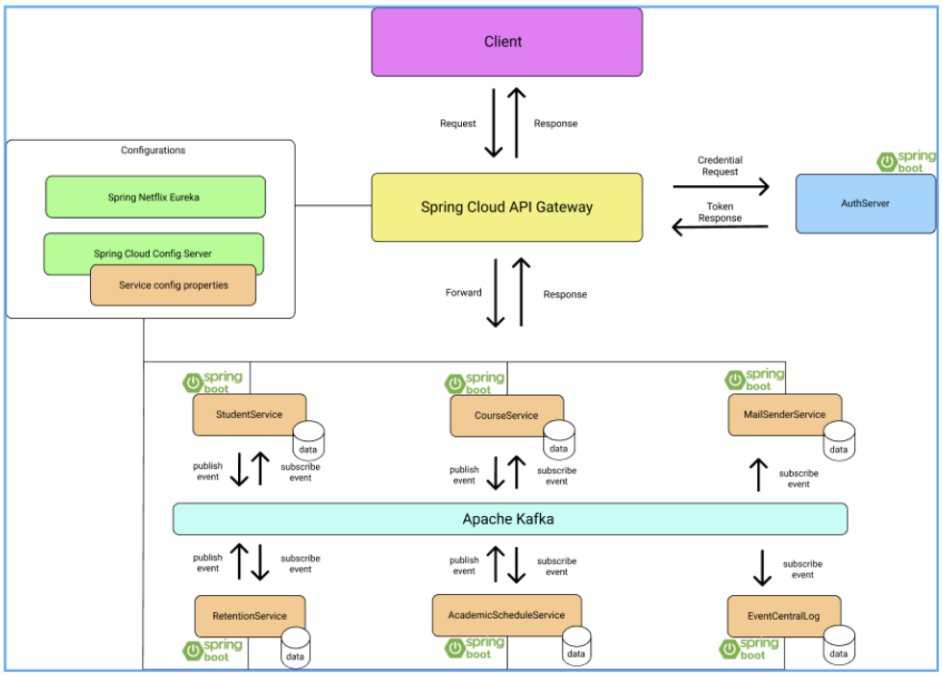


Figure 4: Hoang Tu's thesis

This has brought many advancements to the old microservice integration using only restful requests. The pros of using events are decoupling between services, better scalability, fault tolerance and responsiveness.

For the drawbacks, although Kafka has separated those services from interacting directly to one another, they are still depending on the kind of topics that they want to publish in order to continue the progress. In addition, it is still hard to manage the entire flow of a scenario from end to end. Those disadvantages will be the focus of my thesis to come over them.

CHAPTER 3

METHODOLOGY

This chapter will provide the necessary requirements to be met as well as the designs. This thesis *does not* aim to finish the complete system but all of the current work is for demonstration purpose, which serves as a skeleton for managing event flows using mediator and related event-driven solutions that can be apply not only to student progress managing system but also to any similar field.

3.1. System requirements

Patterns is used to enhance the event-driven system, hence there are many requirements to be addressed in order to show their values. This section will list out the most important requirements follow with a use case diagram.

3.1.1. Requirements

This table contains the five functional requirements for this system:

Requirement	**Description**
Event-driven based system	All services in system should only use event-driven to communicate to one another to ensure decoupling except for the web service that uses restful api to retrieve info from mediators
Managable event flow	All event firing orders for a particular progress should belong to a single mediator, hence the ease for modifying the flow logic
User interface	A web application for displaying important information about mediators in the system, services as well as their data is required for administration
Extensible system	System is constructed so that developers can connect to new mediators along with their associated services without problem
Managable services	It is required to have a way to manage all mediators, services, and service types states including error handling and all together, displayable on the user interface

Table 2: List of requirements

3.1.2. Use case diagram

From those requirements, we can sketch out a use case diagram that will satisfy the objectives of the project like this:

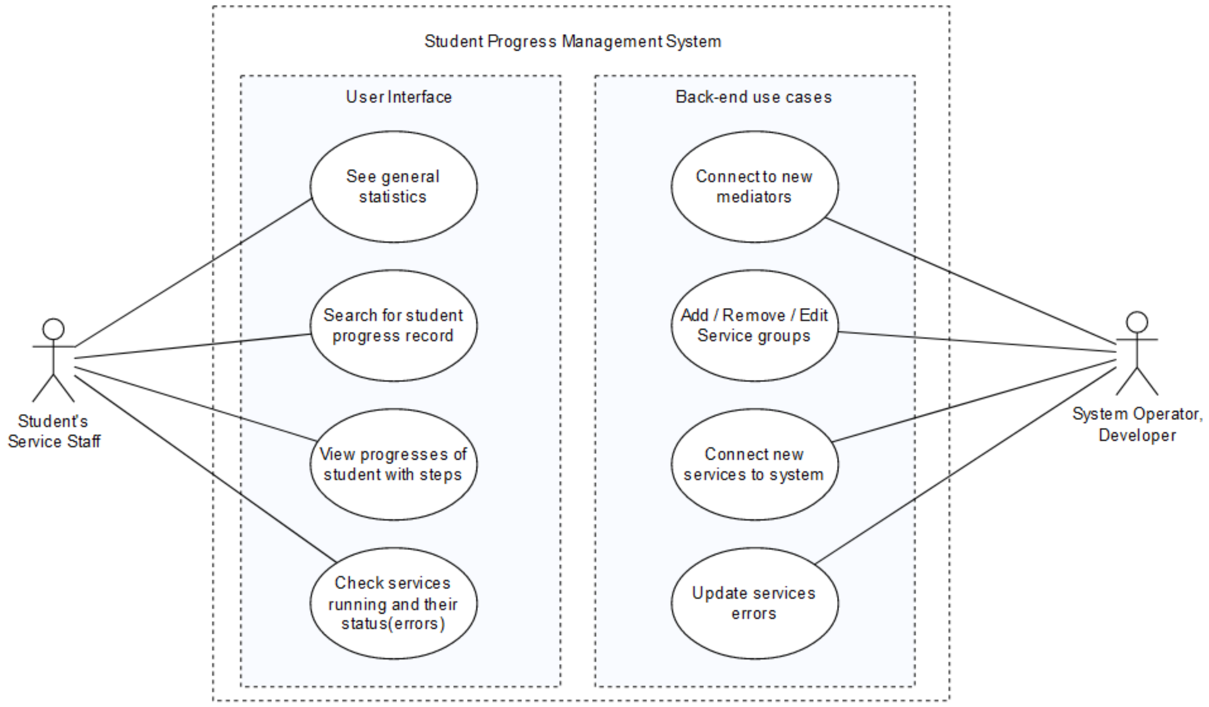

Student Progress Management System
User Interface
See general statistics
Search for student progress record
View progresses of student with steps
Check services running and their status(errors)
Back-end use cases
Connect to new mediators
Add / Remove / Edit Service groups
Connect new services to system
Update services errors
Student's Service Staff
System Operator, Developer

Figure 5: Use case diagram

Two main actors of the system are student's service staff, who constantly checking the system and support when problems or requests arrive, and developers will also benefit from this, who continue to work on the system.

We split into user interface and back-end use cases. Let's go through the first one. The information displayed on the screen will be useful for monitoring student progresses and related system information with four points:

Use case	Description
See general statistics	General statistics are the numbers of students grouped by their progresses and student years; they will be displayed on the dashboard of the web under the form of pie charts for each of progress category. Additionally, metadata about each mediator are also included such as steps descriptions
Search for student progress record	There will be a place that user can view a list of students with their progresses with respect to a specific mediator (progress category), and can be filtered using student ID, name, year, and progress status.
View progresses of student with steps	User can click on a student record and see their steps in checkmarks that indicate which step is completed. When user clicks on a step, it will show further information about the completion percentage and list of smaller items of the step to be finished.
Check services running and their status (errors)	There will be another page that shows a list of mediators in the system and a list of services that can be filtered using the relation to mediators. The services are divided into groups, the services in a group serve the same purpose (for instance, the course service group can have 10 running course services). In addition, the status of the service groups and individual services is also displayed like if they are working or not, and the errors produced by these processors. User can view details by clicking

	on the service group they want to see more.

Table 3: User interface use cases

Besides the use cases for student's service staff on the user interface, we also have many use cases for the backend development side:

Use case	Description
Connect to new mediators	The system is made so that the developers can connect to a new mediator under the assumption that it provides sufficient set of APIs and correct data format.
Add/Remove/Edit service groups	The service groups can be altered to redefine different responsibilities of different groups of services. Actions involve adding, removing groups, editing name or description.
Connect new services to the system	System operators or developers can easily connect to a new service to an existing group by only using events.
Get and update services' errors	Developers can get list of errors and their statuses of whether they are handled. It is also possible to mark the errors as fixed or completely remove the errors.

Table 4: Back-end use cases

3.2. Design

To better understand each component of the designs, we look at a broader view first, then go through each of the specific parts of the whole. This is a simple diagram that shows how the system operates:

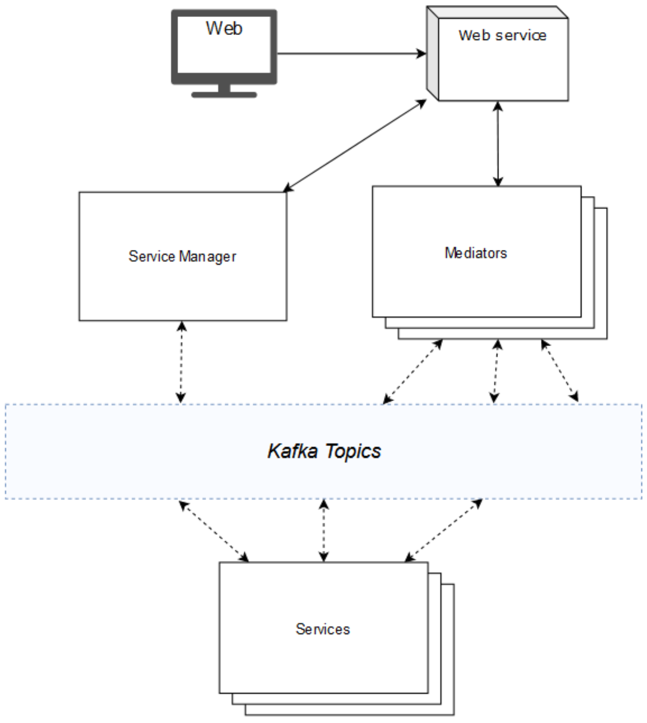


Figure 6: System diagram

First, the user interface is connected to a web service that has the job of gathering all information using rest api calls to the service manager and the mediators. The service manager handles the statuses of which service are running as well as errors. To the right, that rectangle composed of many mediators, each has a purpose of managing progress belongs to its own area. In addition, at the bottom of the diagram are all the remaining services that has their own purpose. And finally, Apache Kafka[5] will be the medium that they all connected by subscribing and publishing messages.

In the following pages, we will go to the details of how each of the components work, including mediators, service managing and how they handle errors.

3.2.1. Mediator design

Semester flow without mediator:

Before mediators are introduced, let's have alook at the flow of a student in a semester progress using broker topology, or a basic implementation without a mediator:

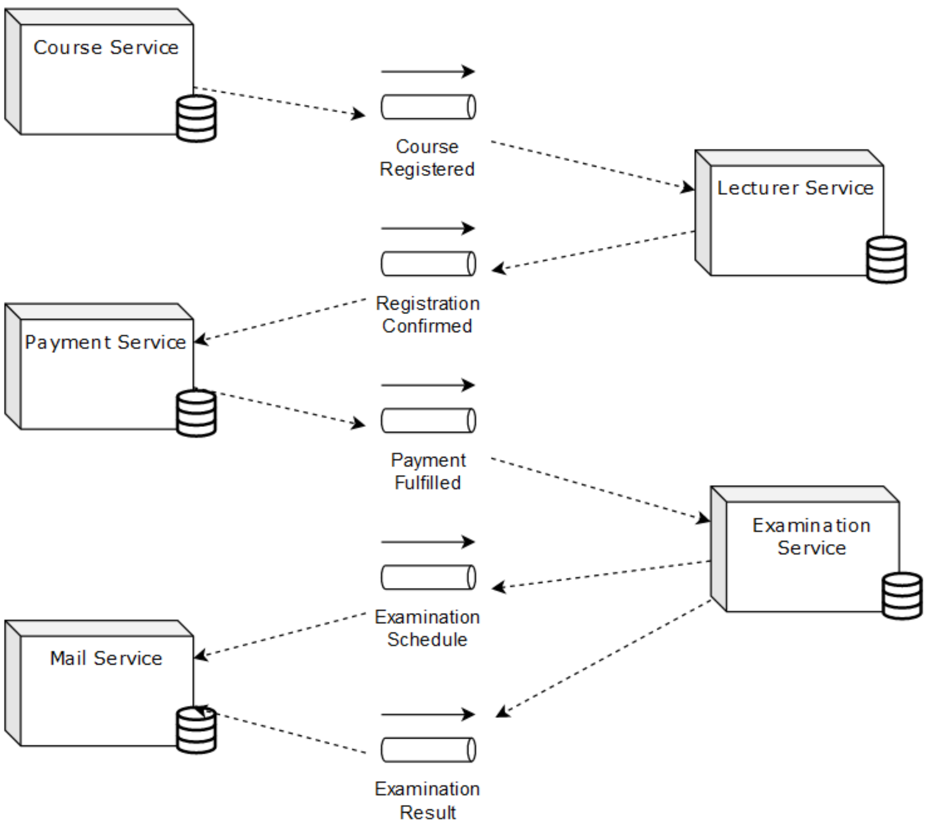


Figure 7: Semester flow without mediator

First, after the courses are registered, it is published to *Course Registered* topic, which then be consumed by *Lecturer Service*. After it is confirmed, a new event will appear in *Registration Confirmed* and continue to the payment step. And so on, we have *Payment Fulfilled*, *Examination Schedule*, and *Examination Result*.

As seen in the diagram, although all the communications are done event-oriented, the entire logic of the flow is controlled by every single service in the system, which can make the events hard to manange, in addition, each processor has to subscribe to unnecessary topics beyond its responsibilities.

Semester flow with mediator:

To solve this problem, we add another central service called a mediator, and it serves as a medium between the event processors, and it contains all of the flow logics that are more flexible and achieve decouplization:

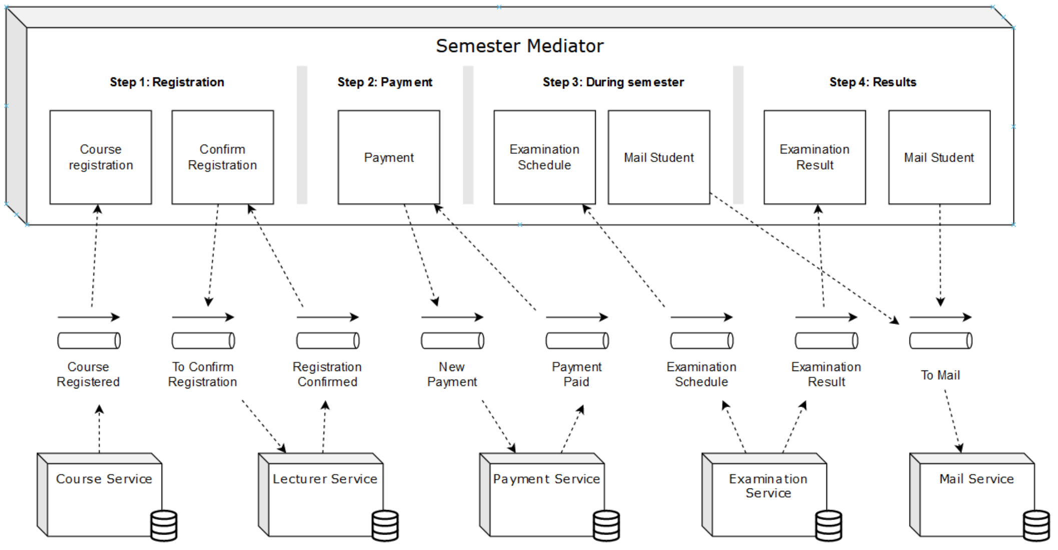


Figure 8: Semester flow with mediator

Internally, the mediator is separated into steps (4 in this case) and it will consume as well as produce to various topics related to the flow. To demonstrate, at the first step, after the courses are registered, the mediator will pick up the signal and transfer it to *To Confirm Registration* for the *Lecturer Service* to receive. Then, when the registrations are confirmed, the mediator gets it back from *Registration Confirmed* topic. And after all entries of a step is completed (indicated by the rectangles inside each step), the progress will be set to the next step (Payment). Similar activities go on in step 2 to 4, where the mediator notifies and receives messages from different services from payment to examination, to mailing.

By using this pattern, we have eliminated convoluted relationships between unrelated services. For instance, the *Lecturer Service* only cares about the 2 registration confirmation topics, *Mail Service* does not need to subscribe to the *Examination Schedule* or *Examination Result* topic. Secondly, the entire order of executions – which events fire first, under which conditions should a service begin processing, is contained only inside the mediator and easy to modify if requirements change.

Two mediators to be implemented for illustration:

In this project, however, few services of the flow are implemented that enough to illustrate the idea. In this case, I have selected *Course Service* and *Lecturer Service* to interact with the *Semester Mediator*. In addition, we need to show the extensibility of the system, to do that another meditor is also setup and it is *General Mediator*, unlike semester, general means to keep track of the progress of the students overall, from entering the university to graduation. I have also implemented two more services: *Certificate Service* and *IUOSS Service* to show the concurrent processing ability of the mediator in one of the steps.

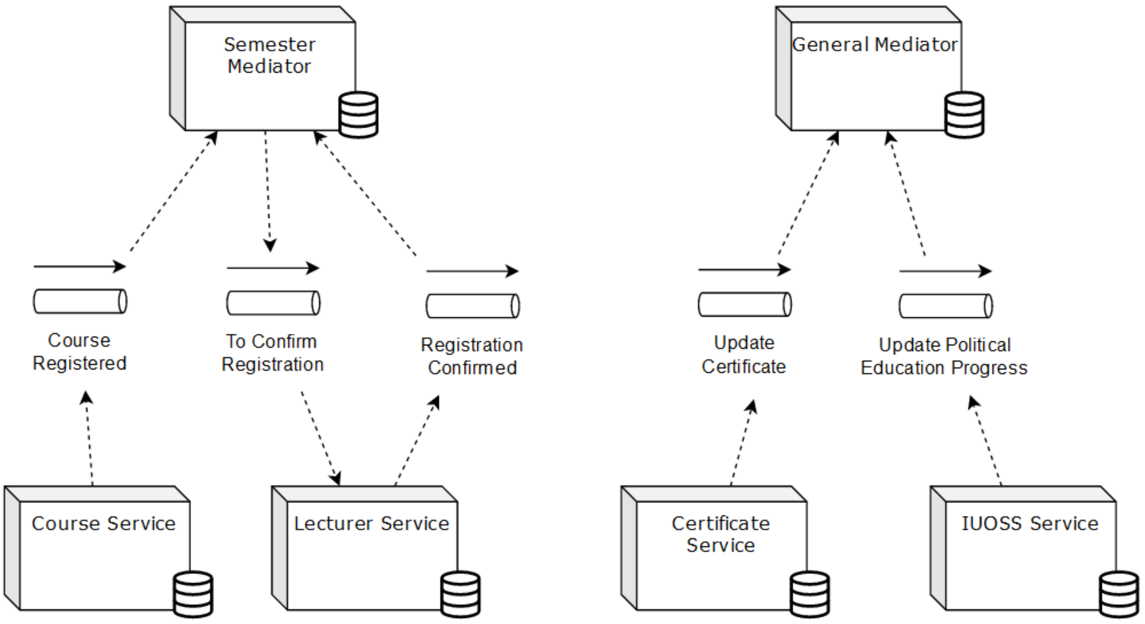


Figure 9: Two mediators for demonstration

3.2.2. Managing services

In order to manage the services, a service manager or shortly, manager, is built and it has several queues that it mainly works with, namely *Service On*, *Service Broadcast*, *Service Error*. For monitoring services' statuses, the diagram below shows how to do it:

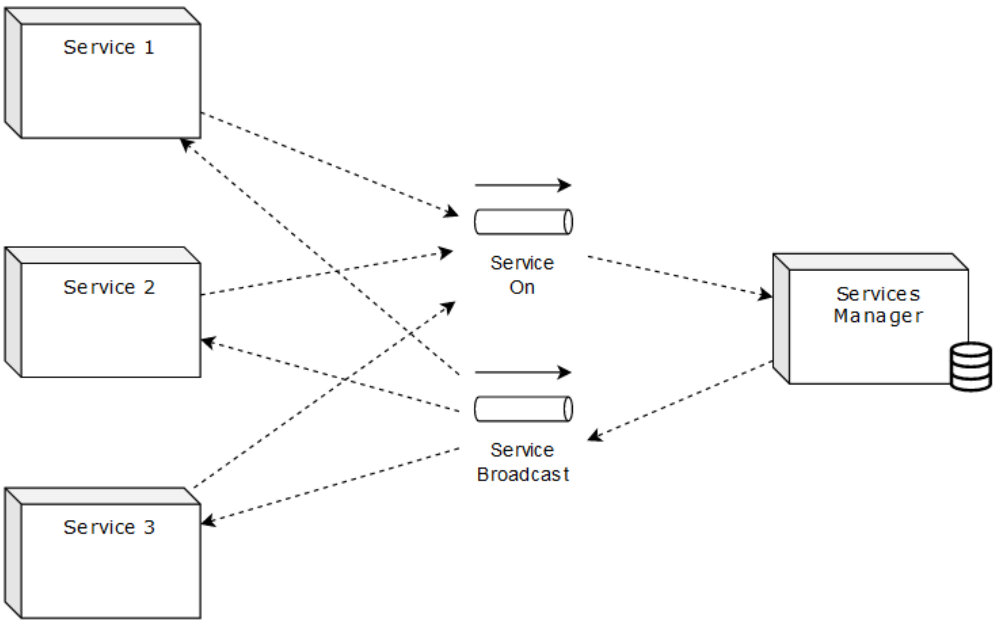


Figure 10: Managing services

Overall, the communications between services and the manager are all event-driven, not directly dependent. When a service starts running, it will push a notification to the *Service On* queue, containing the information of id, name, port and the service-group-id that it wants to register to, the manager then receives the message and register them to the group. The services do not need to signal when they terminate because the manger will auto refresh itself after a fixed period of time. The broadcast topic is used when the manager wants to send to all connected services to reset or related commands.

3.2.3. Error handling design

In addition to the way of manging services, error handling mechanism is also integrated in our system. As seen in the below figure, the *Service Error* topic is introduced and with two other example services, presents a way to catch messages with errors as well as fixing them:

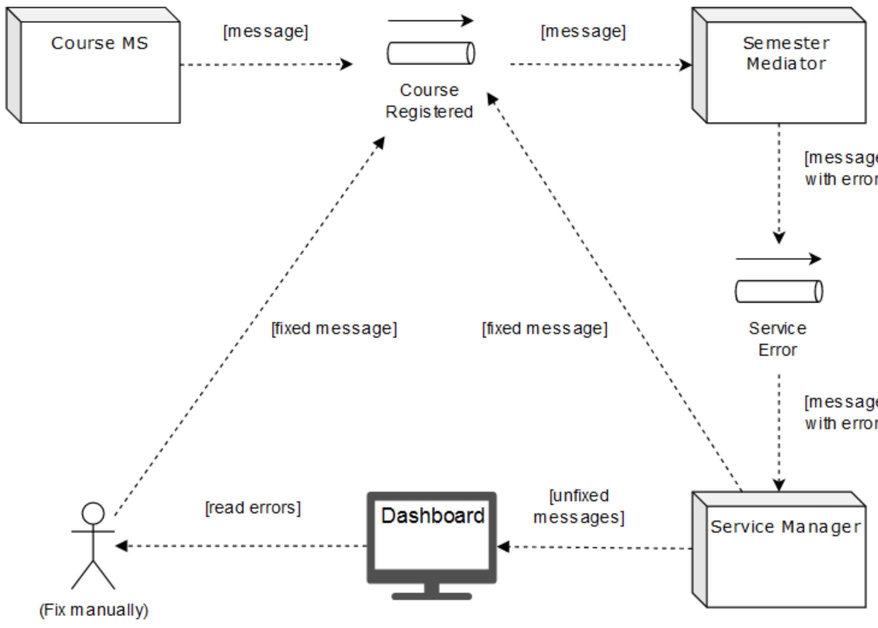


Figure 11: Error handling design

The first action is taken by the *Course MS* pushing a message to the *Course Registered* queue, then *Semester Mediator* consume the event and, in some case, the mediator realized the message has some problem, it will then be forwarded to the *Service Error*, which then goes to the *Service Manager*, the manager will then analyze the content of the message. If the error is well-known and can be fixed directly, the *Service Manager* will do it and post the fixed message back to its queue (*Course Registered*) for the mediator to consume it again. However, if the manager service does not know how to handle the message, it will be kept there and displayed to the dashboard. Then when the administrator notices the error, they can fix it manually and republish it to the topic.

By doing this, we minimize the risk that some service in the flow encounters trouble but cannot be detected especially in a big system. Additionally, this is also a way to handle and fix error automatically using a pure event-driven style.

3.2.4. Complete system diagram

Having discussed about different designs above including mediator, service registrations and error handlings, let’s put them together and see the full diagram:

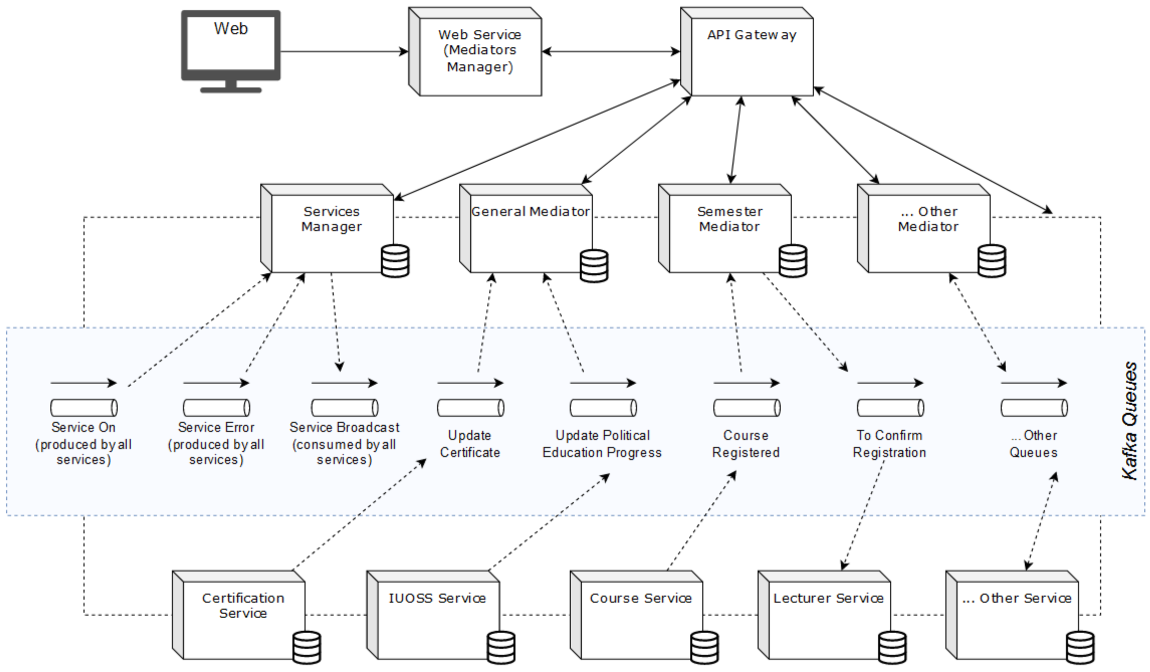


Figure 12: Complete system diagram

We now have several queues in Kafka as mentioned before, three of them (*Service On*, *Service Error*, *Service Broadcast*) are used for service managing and connected by all services (it is a tangled mess if we draw all the arrows). Apart from the *Service Manager*, surrounding Kafka are the services (certification, course, lecturer, ...) and mediators (general, semester). Together, they are accessed by the *API Gateway*, which makes the process of requesting information more convenient, especially for the web service to get the data from all mediators and all services information to display on the screen.

CHAPTER 4

IMPLEMENTATION AND RESULTS

Having discussed about the designs above, to actualize the topology, this chapter will show the inner details on how to implement the core components of the system as well as some of the demonstrations after they are built.

4.1. Implementation

4.1.1. Overview

This is a table of different parts for the coding and their associated technologies:

Type	Description	Technology	Note
Event driven flatform	Event streaming platform	Apache Kafka	
	Distributed configuration	Apache ZooKeeper	required for Kafka
Back-end	Platform	Node.js	
	RESTful API	Express.js	
	API gateway	fast-gateway	
	Kafka client	kafka.js	
	Development tool	nodemon	
Front-end	Platform	Node.js	
	Framework	React.js	
	Styling	tailwind	
	State management	Context API	
	Components	react-router-dom rechart react-select react-icons	
Other	RESTful request	axios	used for both back-end and front-end
	IDE	Visual Studio Code	
	Version control	Git, Github	2 repositories

Table 5: Coding technology stack

We distribute this project in two separate folders, one for back-end and one for front-end. On the other hand, Apache Kafka is manually set up in the computer alone and explained in the following section.

4.1.2. Kafka implementation

To have a message broker running for our project, Kafka and Zookeeper need to be installed in the machine through the below steps, refered to the detailed instructions on dreamix.eu blog[6]:

Installing ZooKeeper:

```
$ sudo apt-get install zookeeperd
```

Start ZooKeeper:

```
$ sudo service zookeeper start
```

Download latest Kafka at website: https://kafka.apache.org/downloads and download the file with tgz extention after that run these commands:

```
$ mkdir ~/kafka
$ wget http://apache.cbox.biz/kafka/0.11.0.1/kafka_2.11-0.11.0.1.tgz
$ tar -xvzf kafka_2.11-0.11.0.1.tgz ~/kafka
```

At some point to reset Kafka, it is required to delete Kafka logs:

```
$ sudo rm -r /tmp/kafka-logs
```

Finally, to start Kafka, run:

```
$ sudo ~/kafka/bin/kafka-server-start.sh ~/kafka/config/server.properties
```

Other useful Kafka commands below:

1) Describing a topic:

```
$ ~/kafka/bin/kafka-topics.sh  --describe --bootstrap-server localhost:9092
--topic topic-name
```

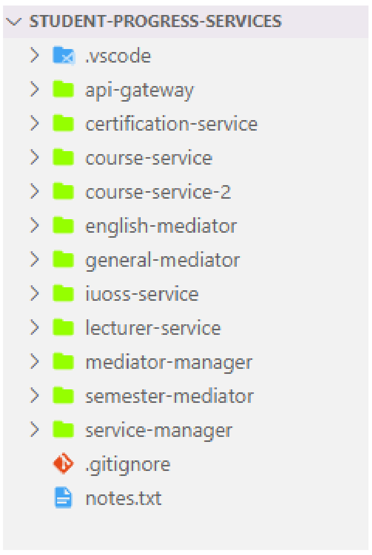
2) Change partitions of a topic:

```
$ ~/kafka/bin/kafka-topics.sh --bootstrap-server localhost:9092 --alter --
topic topic-name --partitions 6
```

4.1.3. Back-end implementation

The back-end side of the system contains multiple microservices that utilize the event broker set up above. In this folder, we can find sub-folders with each one corresponds to a microservice. Along with that is *.vscode* file contains instructions for VSCode to run the services at once without manually typing the commands.

The repository of this part can be visited at: https://github.com/thoai-atb/student-progress-services



```
STUDENT-PROGRESS-SERVICES
  > .vscode
  > api-gateway
  > certification-service
  > course-service
  > course-service-2
  > english-mediator
  > general-mediator
  > iuoss-service
  > lecturer-service
  > mediator-manager
  > semester-mediator
  > service-manager
    .gitignore
    notes.txt
```

Figure 13: Back-end folder structure

Services implementation

This implementation section is for all services and mediators, indicated yellow in the below figure:

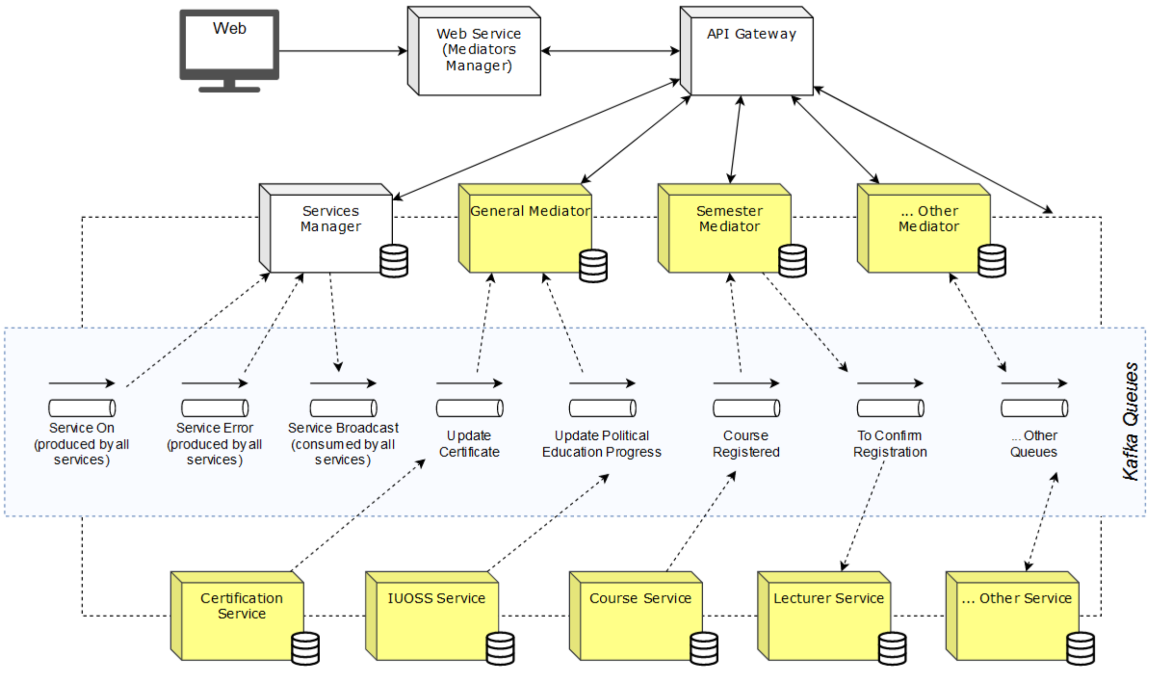


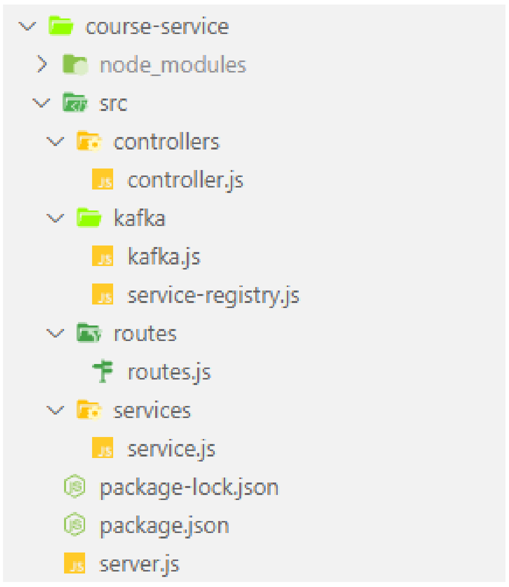
Figure 14: Services and mediators

For the individual folders, Node is the framework selected to run these servers because it is fast and easy to develop especially when dealing with multiple services like this for demonstration and no need for heavy computational work.

Description	**Technology**
Platform	Node.js
RESTful API	Express.js
API gateway library	fast-gateway
Kafka client	kafka.js
Development tool	nodemon
API request	axios

Table 6: Back-end libraries

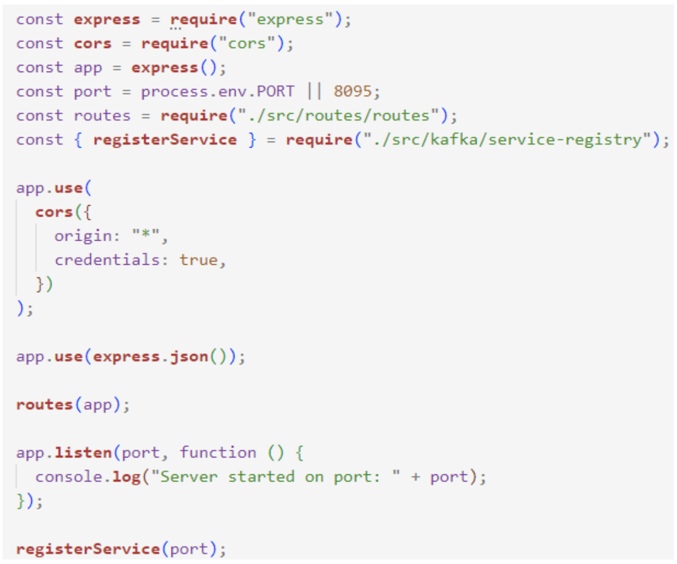
After installing all dependencies, each service will follow this folder structure:



```
course-service
  node_modules
  src
    controllers
      controller.js
    kafka
      kafka.js
      service-registry.js
    routes
      routes.js
    services
      service.js
  package-lock.json
  package.json
  server.js
```

Figure 15: Service folder structure

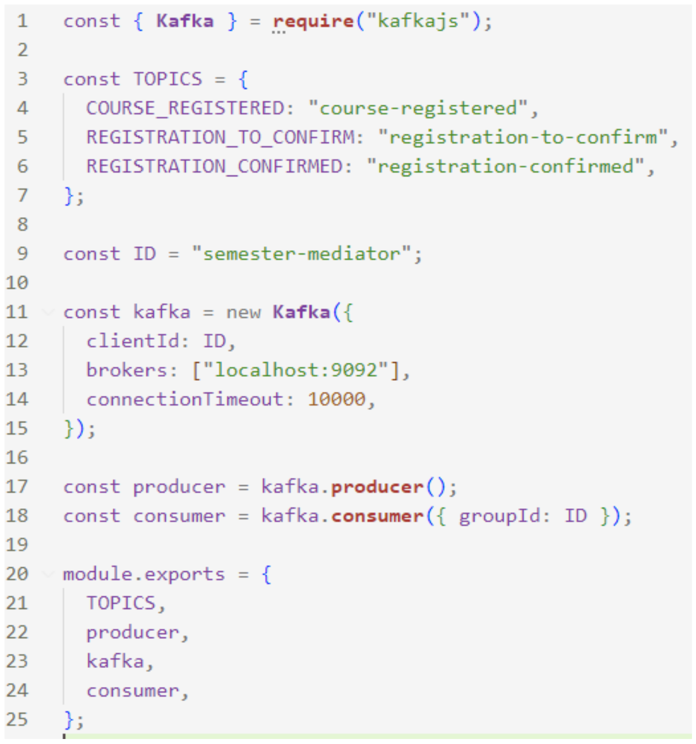
We have *server.js* as the entry point of node, all libraries are inside the *node-modules* folder. Next, we have src that contain smaller folders, that are, controllers, kafka, routes, services. First of all, let's examine the file server.js, which acts like a main function:



```
const express = require("express");
const cors = require("cors");
const app = express();
const port = process.env.PORT || 8095;
const routes = require("./src/routes/routes");
const { registerService } = require("./src/kafka/service-registry");

app.use(
  cors({
    origin: "*",
    credentials: true,
  })
);

app.use(express.json());

routes(app);

app.listen(port, function () {
  console.log("Server started on port: " + port);
});

registerService(port);
```

Figure 16: server.js

We use the `routes()` exported from the routes folder to apply routing to app. When it is ready, `app.listen()` will initiate the app on the specified port. Finally, we register the service to the service manager by using `registerService()` from kafka folder. Assuming the basics are understood, we focus on the event driven related implementations. Now in the kafka folder, the *kafka.js* file defines the topics and information about the brokers as well as setting ID for the current service on Kafka.



```
const { Kafka } = require("kafkajs");

const TOPICS = {
  COURSE_REGISTERED: "course-registered",
  REGISTRATION_TO_CONFIRM: "registration-to-confirm",
  REGISTRATION_CONFIRMED: "registration-confirmed",
};

const ID = "semester-mediator";

const kafka = new Kafka({
  clientId: ID,
  brokers: ["localhost:9092"],
  connectionTimeout: 10000,
});

const producer = kafka.producer();
const consumer = kafka.consumer({ groupId: ID });

module.exports = {
  TOPICS,
  producer,
  kafka,
  consumer,
};
```

Figure 17: kafka.js (semester mediator)

Next, in order to register itself to the service manager, we need to configure the service to produce and consume the topics like the diagram below:

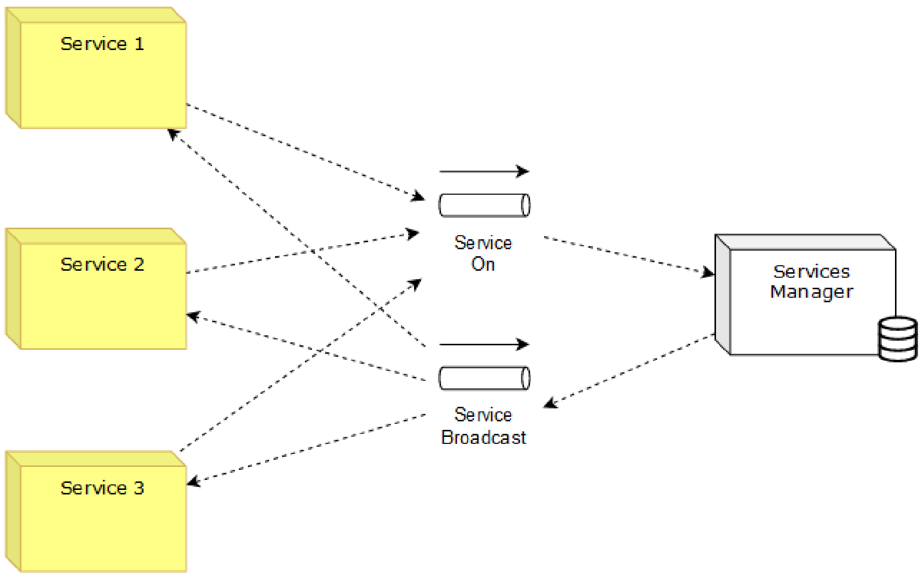
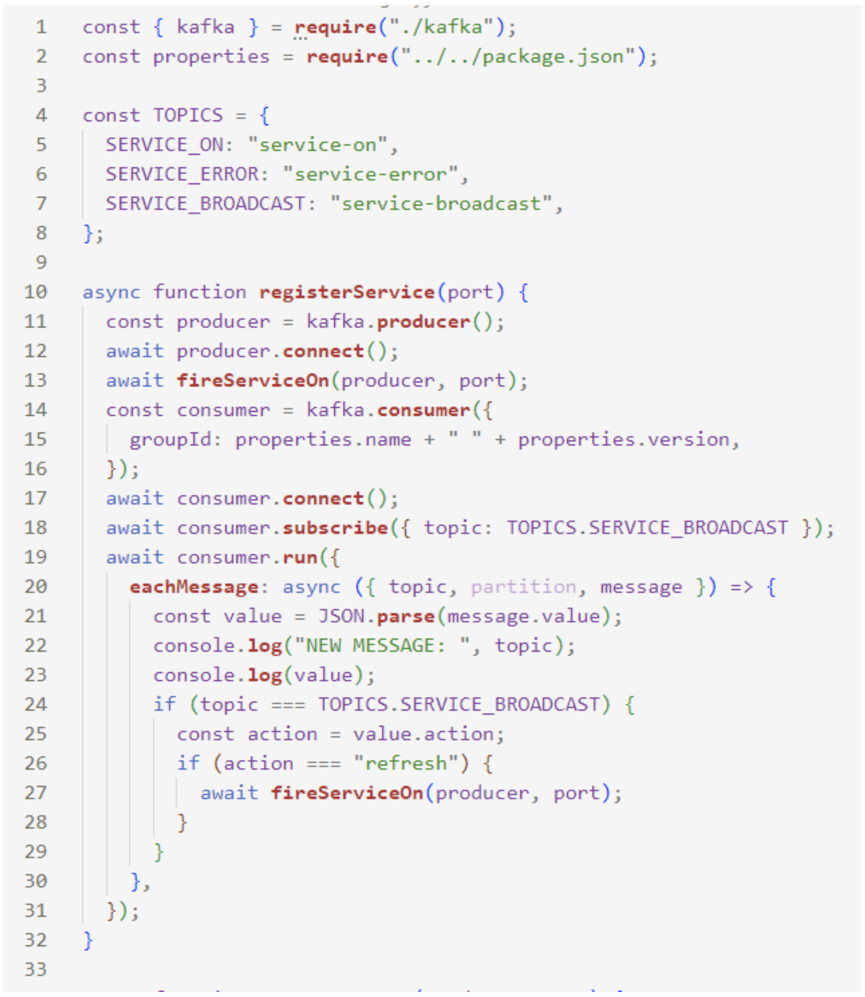


Figure 18: Services managing

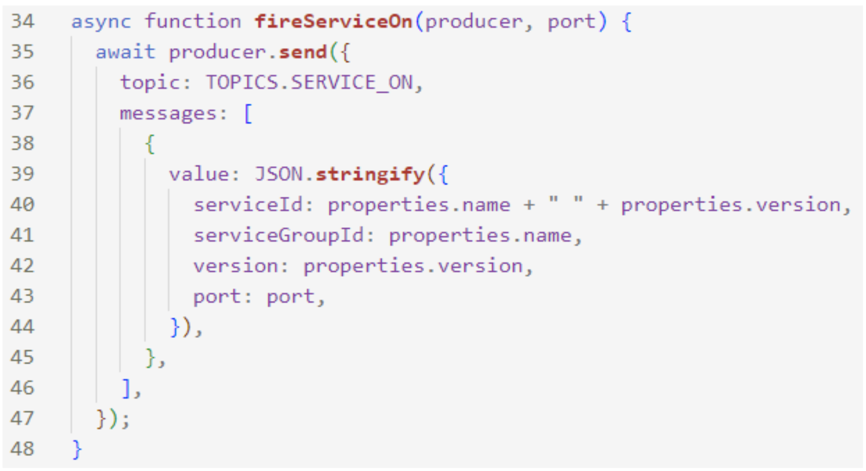
To do this, sitting in the same folder as *kafka.js*, *service-registry.js* is a file that every service must have, the file has the responsibility of handle the service managements. Below is the snip of it:



```
const { kafka } = require("./kafka");
const properties = require("../../package.json");

const TOPICS = {
  SERVICE_ON: "service-on",
  SERVICE_ERROR: "service-error",
  SERVICE_BROADCAST: "service-broadcast",
};

async function registerService(port) {
  const producer = kafka.producer();
  await producer.connect();
  await fireServiceOn(producer, port);
  const consumer = kafka.consumer({
    groupId: properties.name + " " + properties.version,
  });
  await consumer.connect();
  await consumer.subscribe({ topic: TOPICS.SERVICE_BROADCAST });
  await consumer.run({
    eachMessage: async ({ topic, partition, message }) => {
      const value = JSON.parse(message.value);
      console.log("NEW MESSAGE: ", topic);
      console.log(value);
      if (topic === TOPICS.SERVICE_BROADCAST) {
        const action = value.action;
        if (action === "refresh") {
          await fireServiceOn(producer, port);
        }
      }
    },
  });
}

```

Figure 19: service-registry.js

Its concerns are topics related to service management including *SERVICE_ON*, *SERVICE_ERROR*, and *SERVICE_BROADCAST*. It also offers some functions to intiniate actions such as *registerService*, *fireServiceError*. As an example, *registerService* does its job by getting the producer and consumer from *kafka.js* and it subscribes to certain topics and producing to topics.



```
async function fireServiceOn(producer, port) {
  await producer.send({
    topic: TOPICS.SERVICE_ON,
    messages: [
      {
        value: JSON.stringify({
          serviceId: properties.name + " " + properties.version,
          serviceGroupId: properties.name,
          version: properties.version,
          port: port,
        }),
      },
    ],
  });
}
```

Figure 20: fireServiceOn method

Above is an example for producing a topic using producer, the parameters are *topic* and *messages*, where we put the payload of the event inside under the *value* attribute.

Mediator implementation

Now, for the crucial part, let’s move on to mediator implementation:

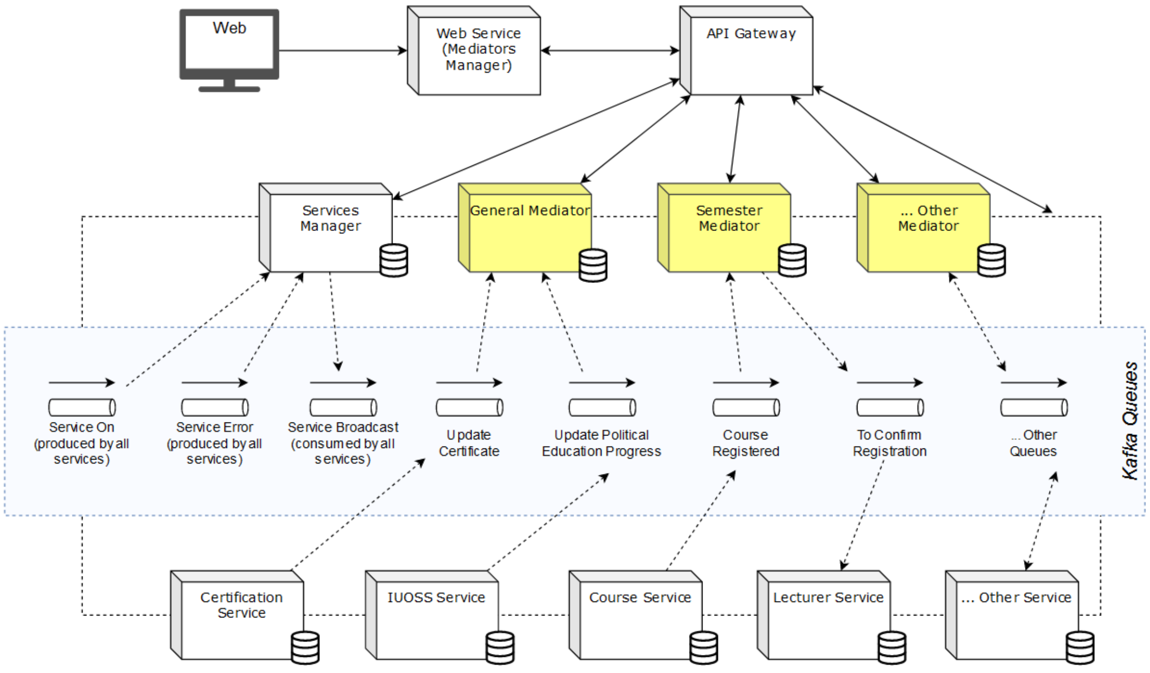
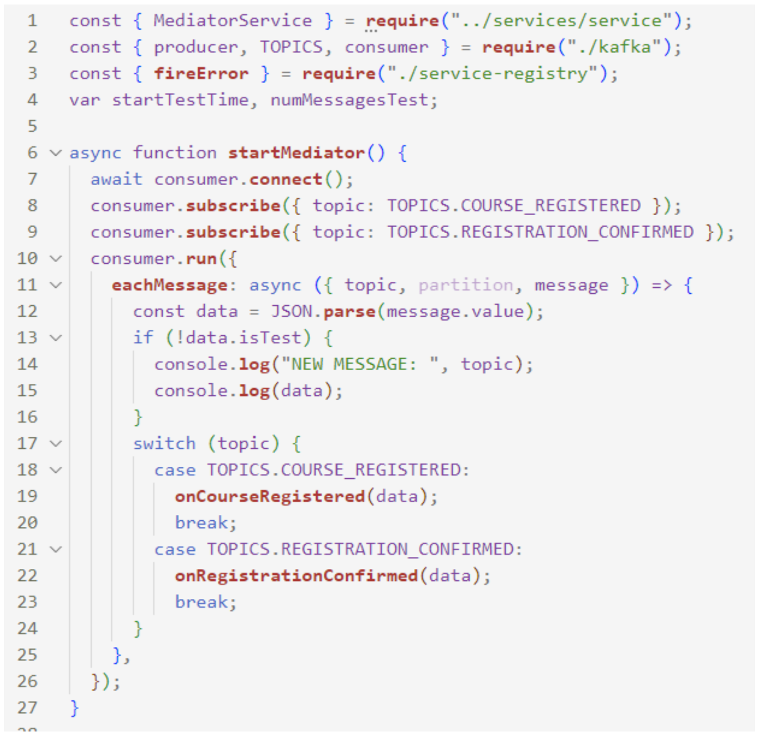


Figure 21: Mediators

This section is for how to manage logics in mediators, we will go through the implementation of semester mediator in the figure below with code snippet:

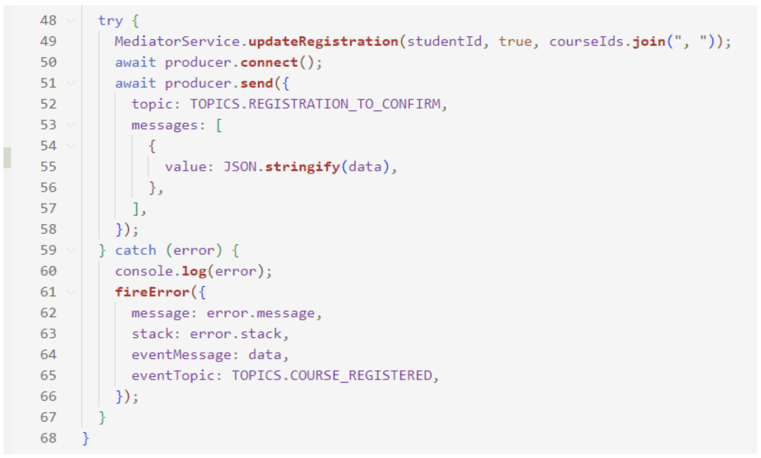


```
const { MediatorService } = require("../services/service");
const { producer, TOPICS, consumer } = require("./kafka");
const { fireError } = require("./service-registry");
var startTestTime, numMessagesTest;

async function startMediator() {
  await consumer.connect();
  consumer.subscribe({ topic: TOPICS.COURSE_REGISTERED });
  consumer.subscribe({ topic: TOPICS.REGISTRATION_CONFIRMED });
  consumer.run({
    eachMessage: async ({ topic, partition, message }) => {
      const data = JSON.parse(message.value);
      if (!data.isTest) {
        console.log("NEW MESSAGE: ", topic);
        console.log(data);
      }
      switch (topic) {
        case TOPICS.COURSE_REGISTERED:
          onCourseRegistered(data);
          break;
        case TOPICS.REGISTRATION_CONFIRMED:
          onRegistrationConfirmed(data);
          break;
      }
    },
  });
}
```

Figure 22: Semester’s mediator.js

Like the other kafka file, this mediator leverages kafka’s comsumer to listen to several topics, and call the appropriate function for each topic, in this case: *onCourseRegistered*, *onRegistrationConfirmed*. To add a new topic, insert a *consumer.subscribe* line for the required topic as well as insert a new case for the switch statement below and implement the function afterward.

Next, let’s see how a single update registration function is done under the following lines of code.



```
  try {
    MediatorService.updateRegistration(studentId, true, courseIds.join(", "));
    await producer.connect();
    await producer.send({
      topic: TOPICS.REGISTRATION_TO_CONFIRM,
      messages: [
        {
          value: JSON.stringify(data),
        },
      ],
    });
  } catch (error) {
    console.log(error);
    fireError({
      message: error.message,
      stack: error.stack,
      eventMessage: data,
      eventTopic: TOPICS.COURSE_REGISTERED,
    });
  }
}
```

Figure 23: onCourseRegistered in mediator.js

When the mediator receives an event from *COURSE_REGISTERED* topic, it will call updateRegistration to update the state of the student in the mediator, after that, the producer instance is connected to Kafka and send the forward message to *REGISTRATION_TO_CONFIRM* for the lecturer service to do the next step.

Services managing implementation

Next is the implementations of the services manager:

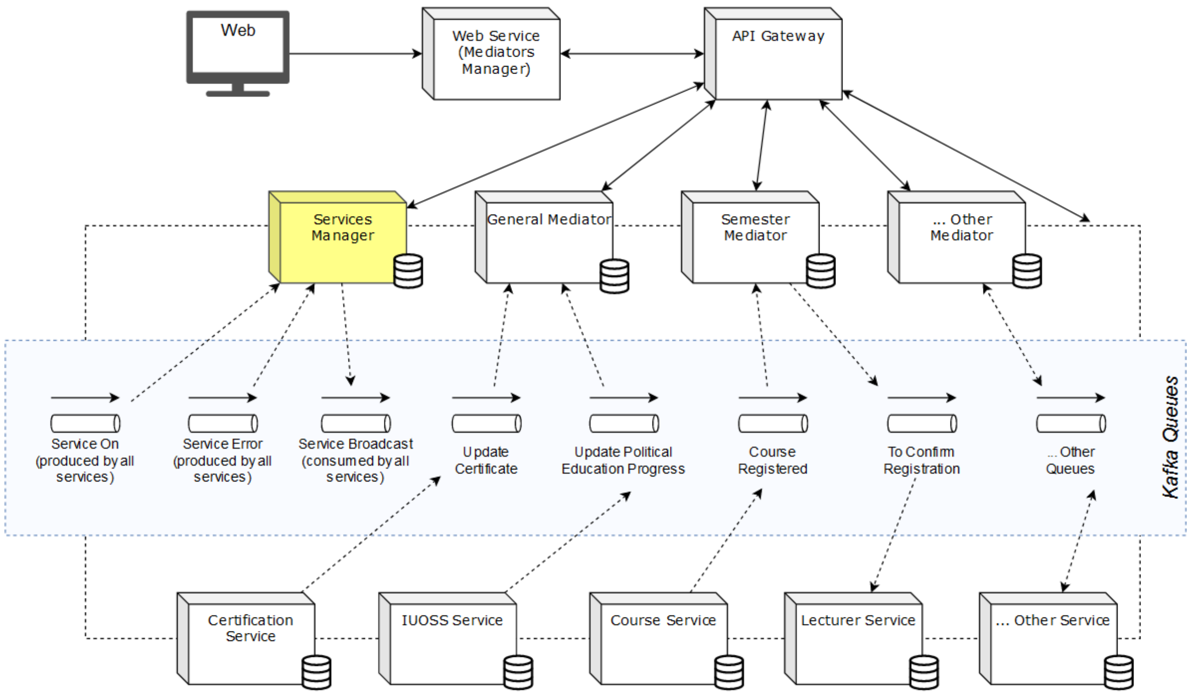


Figure 24: Services Manager

The way services register by publishing events was covered earlier. Now, on the other side, we have the service manager listens to those topics and processes them:

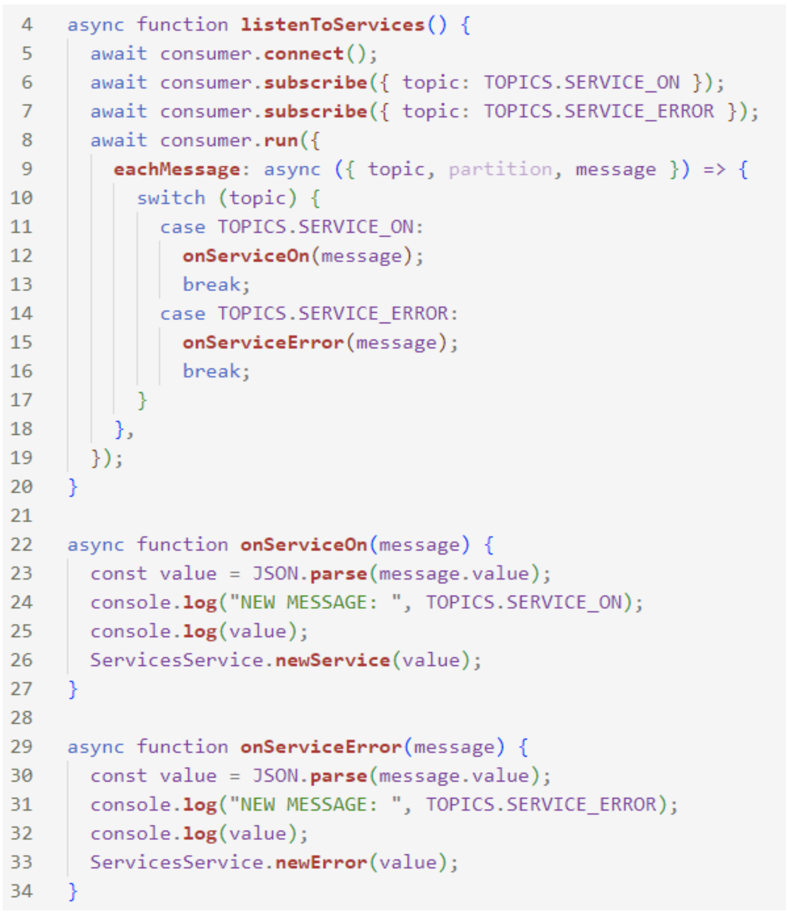
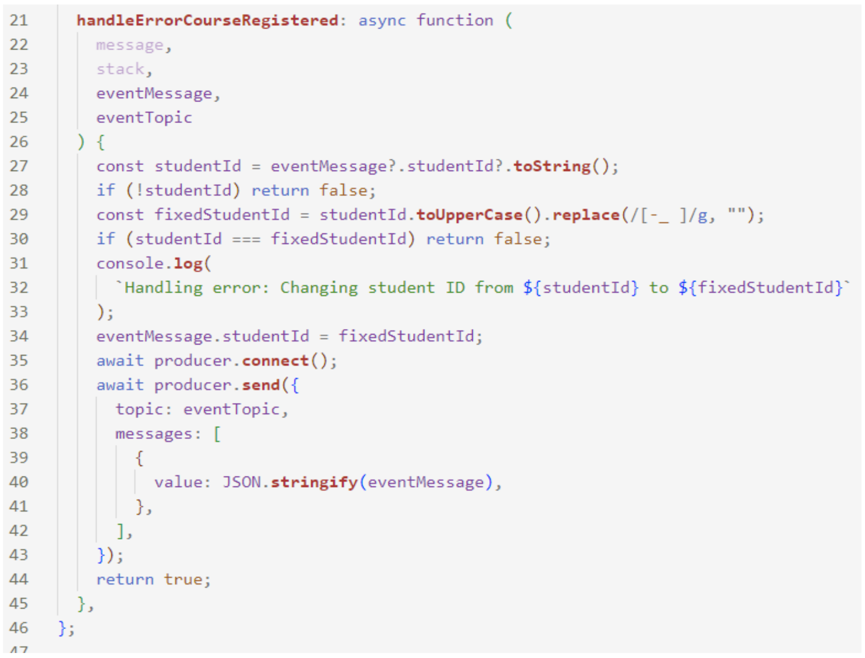

```
async function listenToServices() {
  await consumer.connect();
  await consumer.subscribe({ topic: TOPICS.SERVICE_ON });
  await consumer.subscribe({ topic: TOPICS.SERVICE_ERROR });
  await consumer.run({
    eachMessage: async ({ topic, partition, message }) => {
      switch (topic) {
        case TOPICS.SERVICE_ON:
          onServiceOn(message);
          break;
        case TOPICS.SERVICE_ERROR:
          onServiceError(message);
          break;
      }
    },
  });
}

async function onServiceOn(message) {
  const value = JSON.parse(message.value);
  console.log("NEW MESSAGE: ", TOPICS.SERVICE_ON);
  console.log(value);
  ServicesService.newService(value);
}

async function onServiceError(message) {
  const value = JSON.parse(message.value);
  console.log("NEW MESSAGE: ", TOPICS.SERVICE_ERROR);
  console.log(value);
  ServicesService.newError(value);
}
```

Figure 25: Service manager listening to events

Similar to the implementations of mediators, the manager connects the consumer to subscribe to *SERVICE_ON* and *SERVICE_ERROR*, and then handle each of the events by delegating them to respective functions.

Error handling implementation

Next, let’s move on how to program the manager to automatically fix well-known errors in the system and push them back to the original queues. Below is the screenshot of the function that does just that:



```
  handleErrorCourseRegistered: async function (
    message,
    stack,
    eventMessage,
    eventTopic
  ) {
    const studentId = eventMessage?.studentId?.toString();
    if (!studentId) return false;
    const fixedStudentId = studentId.toUpperCase().replace(/[-_ ]/g, "");
    if (studentId === fixedStudentId) return false;
    console.log(
      `Handling error: Changing student ID from ${studentId} to ${fixedStudentId}`
    );
    eventMessage.studentId = fixedStudentId;
    await producer.connect();
    await producer.send({
      topic: eventTopic,
      messages: [
        {
          value: JSON.stringify(eventMessage),
        },
      ],
    });
    return true;
  },
};
```

Figure 26: Error handling implementation

In this example, the topic of course registration is chosen and the function can handle wrong formatted student’s IDs in the message (for instance: *itit iu-18122* instead of *ITITIU18122*). The process begins by retrieving the attribute and change them by using *toUpperCase* followed by regex replacement of the unexpected characters like dash, hyphen and space. If the modified ID is not the same, then the handler adjusts the payload with the change, produces them to the initiating topic and signals as fixed.

The web service implementation:

Next, let’s look at the web service, or mediators’ manager, is a service that gather information and communicate with the services manager and give data for front-end:

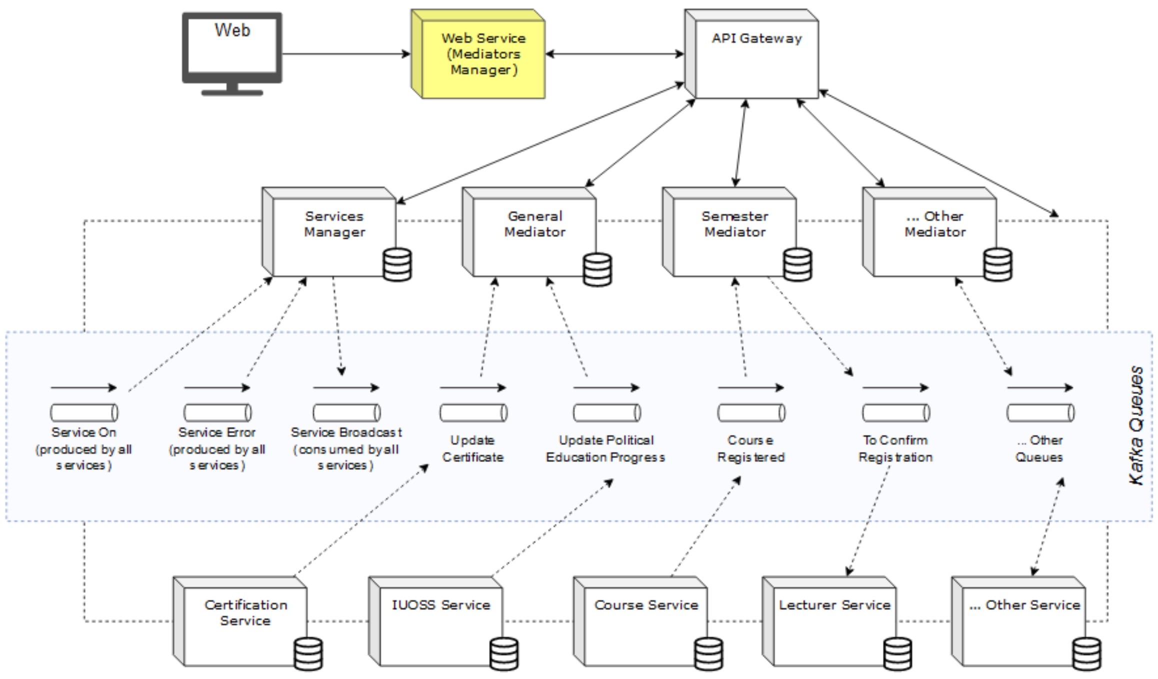
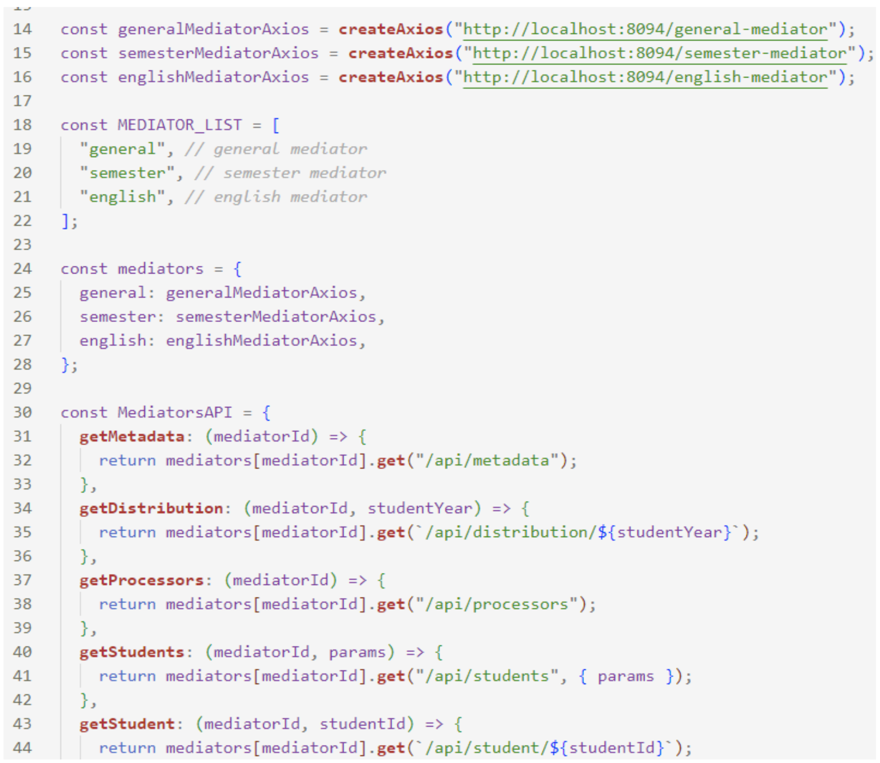


Figure 27: Web service (mediator manager)

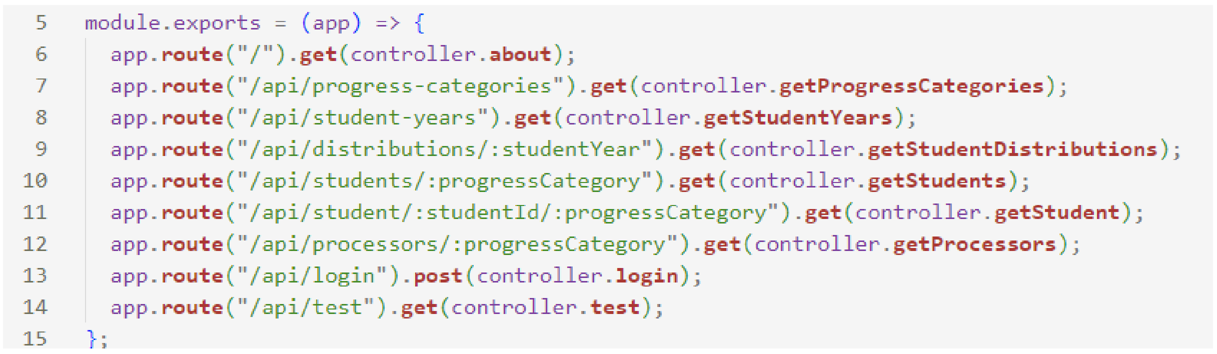
This service has a list of mediators and their connection urls, it also has a defined set of APIs that mediators need to follow to retrieve data.



```
const generalMediatorAxios = createAxios("http://localhost:8094/general-mediator");
const semesterMediatorAxios = createAxios("http://localhost:8094/semester-mediator");
const englishMediatorAxios = createAxios("http://localhost:8094/english-mediator");

const MEDIATOR_LIST = [
  "general", // general mediator
  "semester", // semester mediator
  "english", // english mediator
];

const mediators = {
  general: generalMediatorAxios,
  semester: semesterMediatorAxios,
  english: englishMediatorAxios,
};

const MediatorsAPI = {
  getMetadata: (mediatorId) => {
    return mediators[mediatorId].get("/api/metadata");
  },
  getDistribution: (mediatorId, studentYear) => {
    return mediators[mediatorId].get(`/api/distribution/${studentYear}`);
  },
  getProcessors: (mediatorId) => {
    return mediators[mediatorId].get("/api/processors");
  },
  getStudents: (mediatorId, params) => {
    return mediators[mediatorId].get("/api/students", { params });
  },
  getStudent: (mediatorId, studentId) => {
    return mediators[mediatorId].get(`/api/student/${studentId}`);
```

Figure 28: Web service implementation

Therefore, to add a new mediator, we need to insert it into this file manually by the developers and it has to follow the conventional way to expose its API. As much work as it is, this can further be decoupilized by using the proxy pattern but this is enough to have everything working.



```javascript
module.exports = (app) => {
  app.route("/").get(controller.about);
  app.route("/api/progress-categories").get(controller.getProgressCategories);
  app.route("/api/student-years").get(controller.getStudentYears);
  app.route("/api/distributions/:studentYear").get(controller.getStudentDistributions);
  app.route("/api/students/:progressCategory").get(controller.getStudents);
  app.route("/api/student/:studentId/:progressCategory").get(controller.getStudent);
  app.route("/api/processors/:progressCategory").get(controller.getProcessors);
  app.route("/api/login").post(controller.login);
  app.route("/api/test").get(controller.test);
};
```

Figure 29: APIs for front-end (routes.js)

After the connections with the mediators are met, the service can deliver the API set that front-end can call to display the content on the screen.

This is the end of back-end implementation.

4.1.4. Front-end implementation

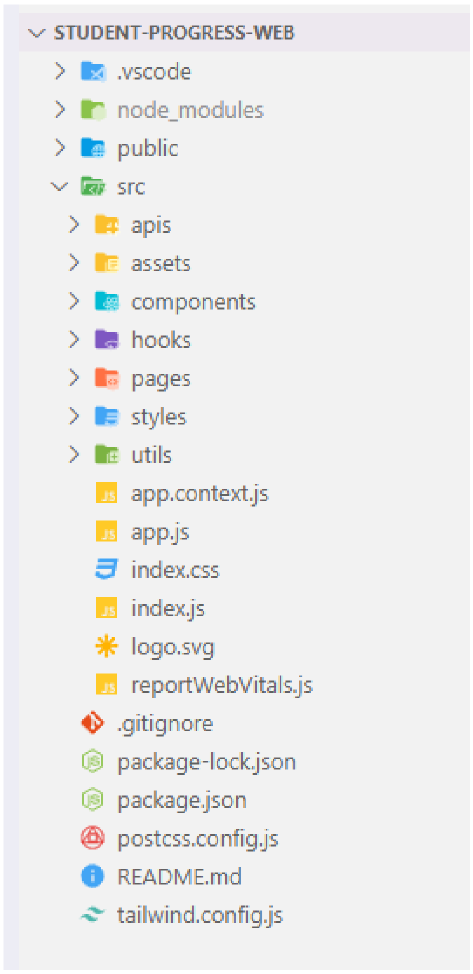
For displaying relavent info from the complex back-end services, I built a user interface for monitoring the whole system using React, a recent popular front-end framework that runs on Nodejs. This is put into a different folder as well as different git repository from the back-end's.

The repository of this part can be visited at: https://github.com/thoai-atb/student-progress-web

Description	Technology
Platform	Node.js
Framework	React.js
Styling	tailwind
State management	Context API
Components	react-router-dom rechart react-select react-icons
API requests	axios

Table 7: Front-end libraries

This is the folder structure of student progress management web:

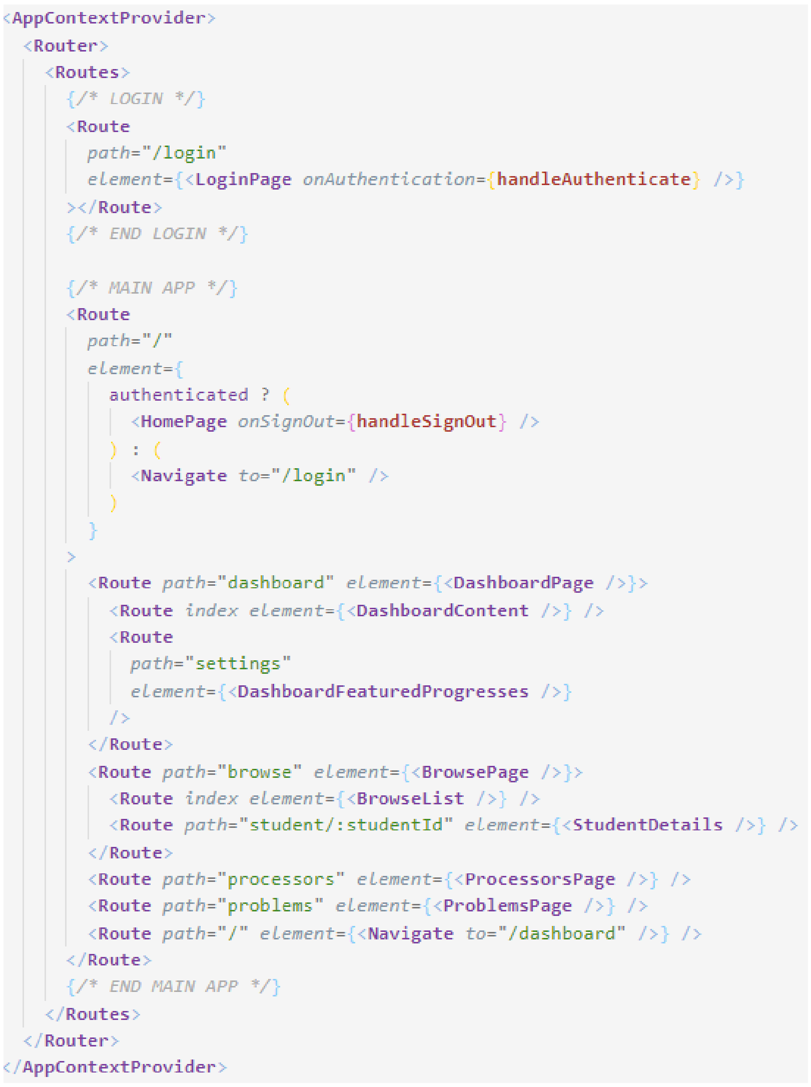


```
STUDENT-PROGRESS-WEB
  .vscode
  node_modules
  public
  src
    apis
    assets
    components
    hooks
    pages
    styles
    utils
    app.context.js
    app.js
    index.css
    index.js
    logo.svg
    reportWebVitals.js
  .gitignore
  package-lock.json
  package.json
  postcss.config.js
  README.md
  tailwind.config.js
```

Figure 30: Front-end folder structure

This project is structured so that inside src folder are apis, assets, components, hooks, pages, styles, and utils. Like all React projects, the entry point is *index.js* and the main component is *app.js*.

In app.js we put the routing structure of the web that navigates to every component inside the *pages* folder, including login page, dashboard page, browse page, student details, processors page, and problems page. A context provider wrapper is also implemented to manage the state of the whole app.

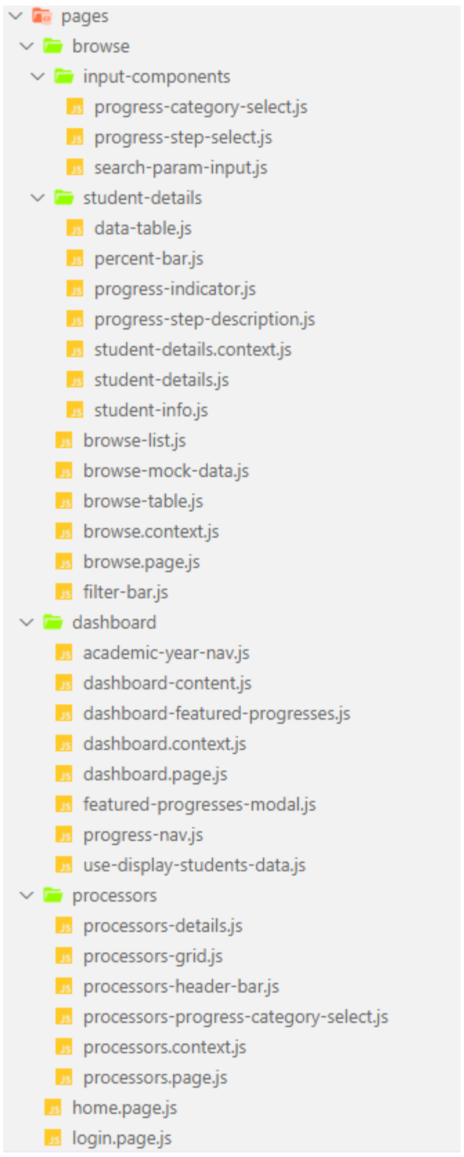


```
<AppContextProvider>
  <Router>
    <Routes>
      {/* LOGIN */}
      <Route
        path="/login"
        element={<LoginPage onAuthentication={handleAuthenticate} />}
      ></Route>
      {/* END LOGIN */}

      {/* MAIN APP */}
      <Route
        path="/"
        element={
          authenticated ? (
            <HomePage onSignOut={handleSignOut} />
          ) : (
            <Navigate to="/login" />
          )
        }
      >
        <Route path="dashboard" element={<DashboardPage />}>
          <Route index element={<DashboardContent />} />
          <Route
            path="settings"
            element={<DashboardFeaturedProgresses />}
          />
        </Route>
        <Route path="browse" element={<BrowsePage />}>
          <Route index element={<BrowseList />} />
          <Route path="student/:studentId" element={<StudentDetails />} />
        </Route>
        <Route path="processors" element={<ProcessorsPage />} />
        <Route path="problems" element={<ProblemsPage />} />
        <Route path="/" element={<Navigate to="/dashboard" />} />
      </Route>
      {/* END MAIN APP */}
    </Routes>
  </Router>
</AppContextProvider>
```

Figure 31: Front-end routing (app.js)

With the new updates of react router dom v6, we can now nest children's and parent's routes together in one file, that makes the routing much easier to read.

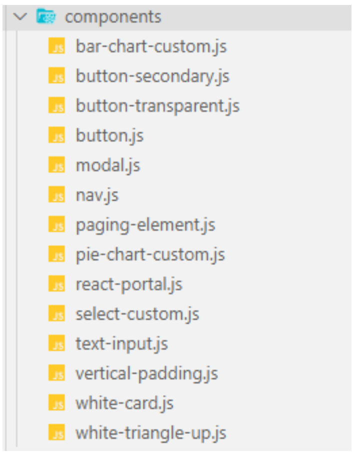
Inside the *pages* folder, we assign each page to a different folder with the corresponding name. all of the related components are put inside the same folder. For home and login page, there are simple enough so they are left outside.



```
pages
  browse
    input-components
      progress-category-select.js
      progress-step-select.js
      search-param-input.js
    student-details
      data-table.js
      percent-bar.js
      progress-indicator.js
      progress-step-description.js
      student-details.context.js
      student-details.js
      student-info.js
    browse-list.js
    browse-mock-data.js
    browse-table.js
    browse.context.js
    browse.page.js
    filter-bar.js
  dashboard
    academic-year-nav.js
    dashboard-content.js
    dashboard-featured-progresses.js
    dashboard.context.js
    dashboard.page.js
    featured-progresses-modal.js
    progress-nav.js
    use-display-students-data.js
  processors
    processors-details.js
    processors-grid.js
    processors-header-bar.js
    processors-progress-category-select.js
    processors.context.js
    processors.page.js
  home.page.js
  login.page.js
```

Figure 32: Pages folder

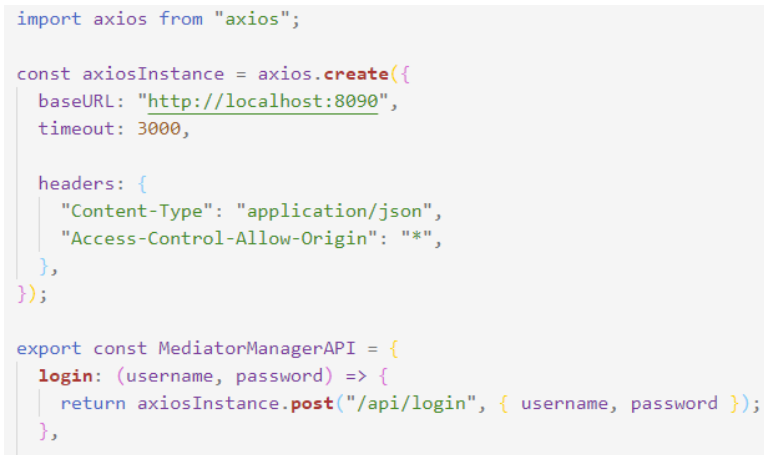
If the components of the apps are shared among pages, such as buttons, text inputs, cards; they are put into the *components* folder like this:



```
components
  bar-chart-custom.js
  button-secondary.js
  button-transparent.js
  button.js
  modal.js
  nav.js
  paging-element.js
  pie-chart-custom.js
  react-portal.js
  select-custom.js
  text-input.js
  vertical-padding.js
  white-card.js
  white-triangle-up.js
```

Figure 33: Components folder

Next, let’s see how the connection is made to the web service under back-end:

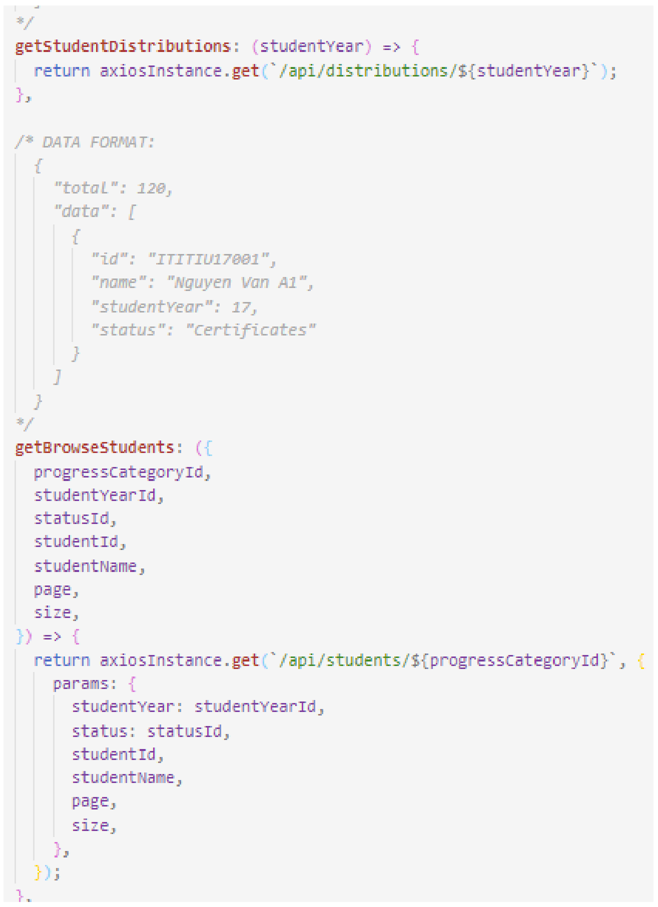


```
import axios from "axios";

const axiosInstance = axios.create({
  baseURL: "http://localhost:8090",
  timeout: 3000,

  headers: {
    "Content-Type": "application/json",
    "Access-Control-Allow-Origin": "*",
  },
});

export const MediatorManagerAPI = {
  login: (username, password) => {
    return axiosInstance.post("/api/login", { username, password });
  },
```

Figure 34: Using axios in front-end

We use *axios* for better supported fetching API requests. Inside *mediator-manager-api.js*, different APIs are listed and converted to functions inside object *MediatorManagerAPI* that will be used later on.

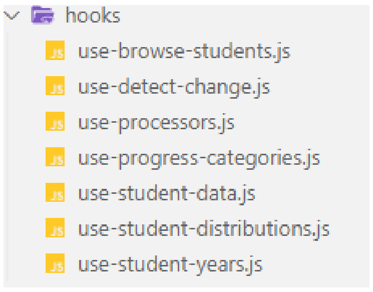
The figure below shows the examples for getting student distribution and browsing using *axios* instance:



```
*/
getStudentDistributions: (studentYear) => {
  return axiosInstance.get(`/api/distributions/${studentYear}`);
},

/* DATA FORMAT:
  {
    "total": 120,
    "data": [
      {
        "id": "ITITIU17001",
        "name": "Nguyen Van A1",
        "studentYear": 17,
        "status": "Certificates"
      }
    ]
  }
*/
getBrowseStudents: ({
  progressCategoryId,
  studentYearId,
  statusId,
  studentId,
  studentName,
  page,
  size,
}) => {
  return axiosInstance.get(`/api/students/${progressCategoryId}`, {
    params: {
      studentYear: studentYearId,
      status: statusId,
      studentId,
      studentName,
      page,
      size,
    },
  });
},
```

Figure 35: Few examples for getting data from back-end APIs using axios instance (mediator api)

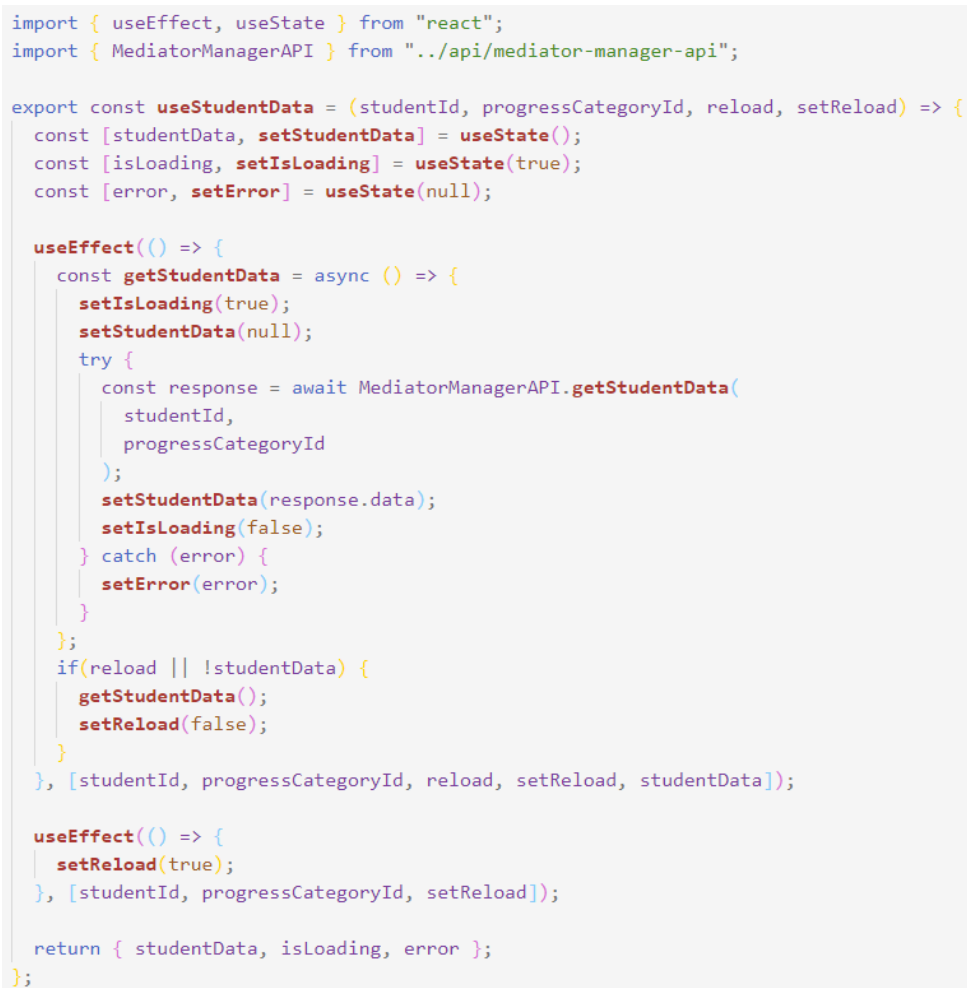
The next layer of request is in the *hooks* folder, there are different hooks for different requests like this:



```
hooks
    use-browse-students.js
    use-detect-change.js
    use-processors.js
    use-progress-categories.js
    use-student-data.js
    use-student-distributions.js
    use-student-years.js
```

Figure 36: Hooks folder

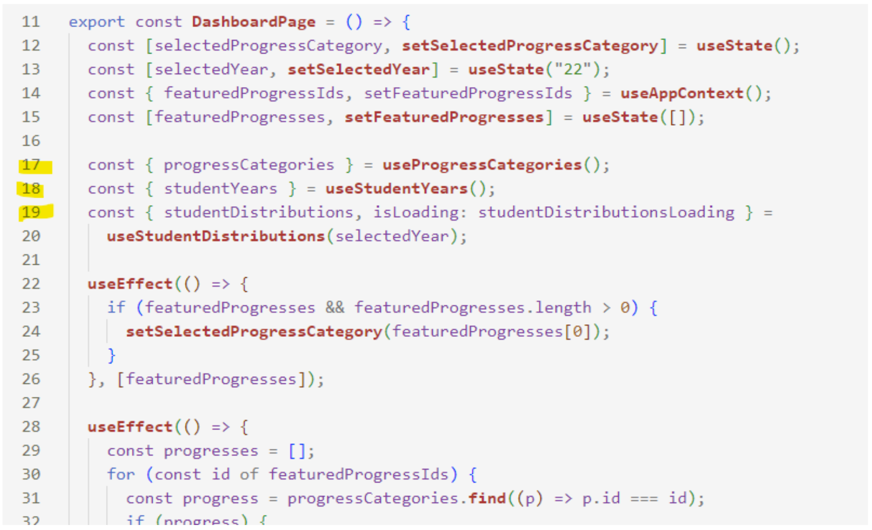
A hook is just a function that leverages *useState* and *useEffect* just like a component but does not render anything. We use hooks to decouple request logic from representational logic.



```
import { useEffect, useState } from "react";
import { MediatorManagerAPI } from "../api/mediator-manager-api";

export const useStudentData = (studentId, progressCategoryId, reload, setReload) => {
  const [studentData, setStudentData] = useState();
  const [isLoading, setIsLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    const getStudentData = async () => {
      setIsLoading(true);
      setStudentData(null);
      try {
        const response = await MediatorManagerAPI.getStudentData(
          studentId,
          progressCategoryId
        );
        setStudentData(response.data);
        setIsLoading(false);
      } catch (error) {
        setError(error);
      }
    };
    if(reload || !studentData) {
      getStudentData();
      setReload(false);
    }
  }, [studentId, progressCategoryId, reload, setReload, studentData]);

  useEffect(() => {
    setReload(true);
  }, [studentId, progressCategoryId, setReload]);

  return { studentData, isLoading, error };
};
```

Figure 37: Custom hook that calls API to get data

Finally, inside the component page, we need to import and use the defined hooks to get the data:



```
export const DashboardPage = () => {
  const [selectedProgressCategory, setSelectedProgressCategory] = useState();
  const [selectedYear, setSelectedYear] = useState("22");
  const { featuredProgressIds, setFeaturedProgressIds } = useAppContext();
  const [featuredProgresses, setFeaturedProgresses] = useState([]);

  const { progressCategories } = useProgressCategories();
  const { studentYears } = useStudentYears();
  const { studentDistributions, isLoading: studentDistributionsLoading } =
    useStudentDistributions(selectedYear);

  useEffect(() => {
    if (featuredProgresses && featuredProgresses.length > 0) {
      setSelectedProgressCategory(featuredProgresses[0]);
    }
  }, [featuredProgresses]);

  useEffect(() => {
    const progresses = [];
    for (const id of featuredProgressIds) {
      const progress = progressCategories.find((p) => p.id === id);
      if (progress) {
```

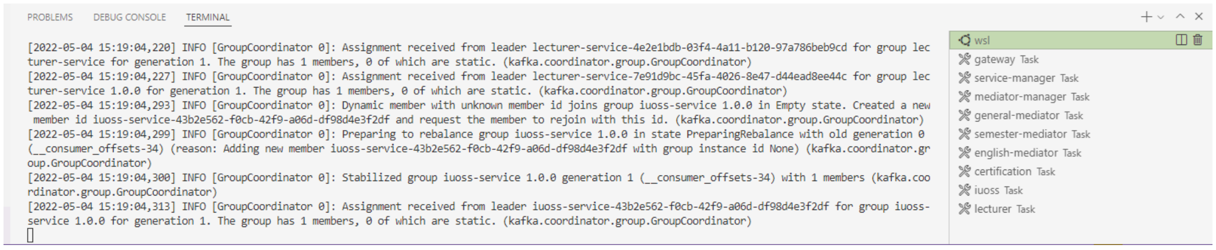
Figure 38: Using hooks in dashboard page

Above is the dashboard page that uses the request to progress categories, student years and student distributions to display.

This is the end of front-end implementation.

4.2. Results

First, for the services running properly, we start Kafka first as explained above, then start all the services (by utilizing VSCode's help for running multiple services for developing by Ctrl + Shift + P -> Tasks: Run Task, and select "start all" as configured inside .vscode/tasks.json)



```
[2022-05-04 15:19:04,220] INFO [GroupCoordinator 0]: Assignment received from leader lecturer-service-4e2e1bdb-03f4-4a11-b120-97a786beb9cd for group lec
turer-service for generation 1. The group has 1 members, 0 of which are static. (kafka.coordinator.group.GroupCoordinator)
[2022-05-04 15:19:04,227] INFO [GroupCoordinator 0]: Assignment received from leader lecturer-service-7e91d9bc-45fa-4026-8e47-d44ead8ee44c for group lec
turer-service 1.0.0 for generation 1. The group has 1 members, 0 of which are static. (kafka.coordinator.group.GroupCoordinator)
[2022-05-04 15:19:04,293] INFO [GroupCoordinator 0]: Dynamic member with unknown member id joins group iuoss-service 1.0.0 in Empty state. Created a new
 member id iuoss-service-43b2e562-f0cb-42f9-a06d-df98d4e3f2df and request the member to rejoin with this id. (kafka.coordinator.group.GroupCoordinator)
[2022-05-04 15:19:04,299] INFO [GroupCoordinator 0]: Preparing to rebalance group iuoss-service 1.0.0 in state PreparingRebalance with old generation 0
(__consumer_offsets-34) (reason: Adding new member iuoss-service-43b2e562-f0cb-42f9-a06d-df98d4e3f2df with group instance id None) (kafka.coordinator.gr
oup.GroupCoordinator)
[2022-05-04 15:19:04,300] INFO [GroupCoordinator 0]: Stabilized group iuoss-service 1.0.0 generation 1 (__consumer_offsets-34) with 1 members (kafka.coo
rdinator.group.GroupCoordinator)
[2022-05-04 15:19:04,313] INFO [GroupCoordinator 0]: Assignment received from leader iuoss-service-43b2e562-f0cb-42f9-a06d-df98d4e3f2df for group iuoss-
service 1.0.0 for generation 1. The group has 1 members, 0 of which are static. (kafka.coordinator.group.GroupCoordinator)
```

Figure 39: Starting Kafka and all services

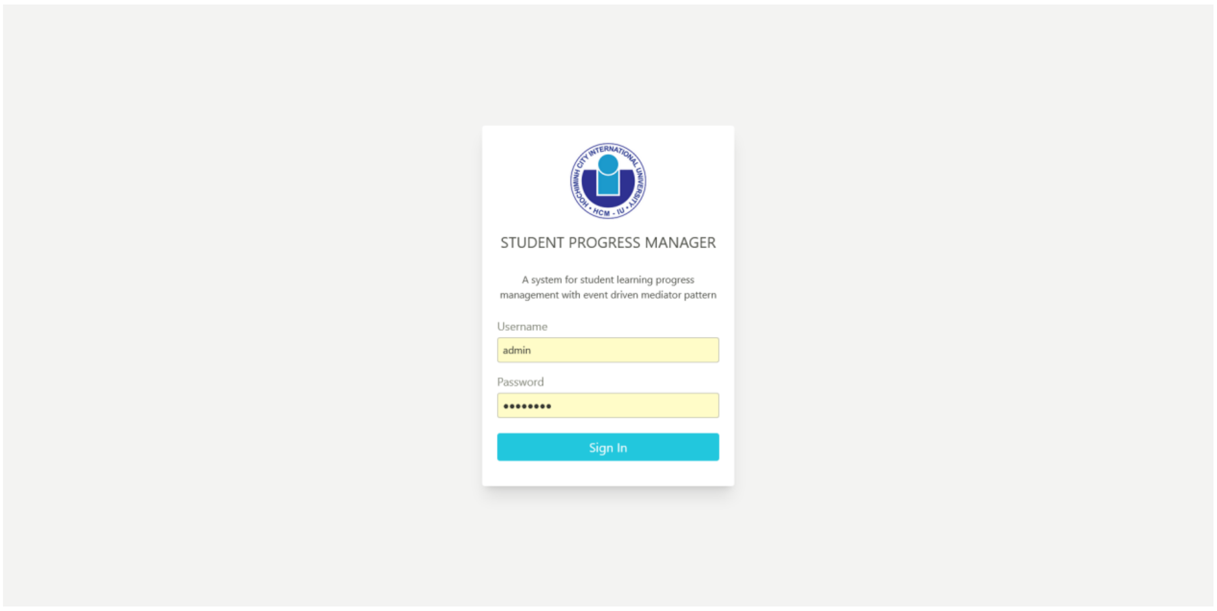
As seen in the figure above, we have the wsl (Kafka), mediators (general, semester, english) and other services running, additionally web service (mediator manager) and gateway.

Finally start the web server by running `npm run start` inside the front-end directory.

This section is divided into user interface, mediator demo, services management and error handling demo with results captured on the screen as well as back-end logs.

4.2.1. User interface

At the beginning, a login form below is displayed and the user need to fill in username and password, for the scope of this thesis, authentication is simply hard-coded into back-end without having to create manager accounts.


STUDENT PROGRESS MANAGER

A system for student learning progress management with event driven mediator pattern

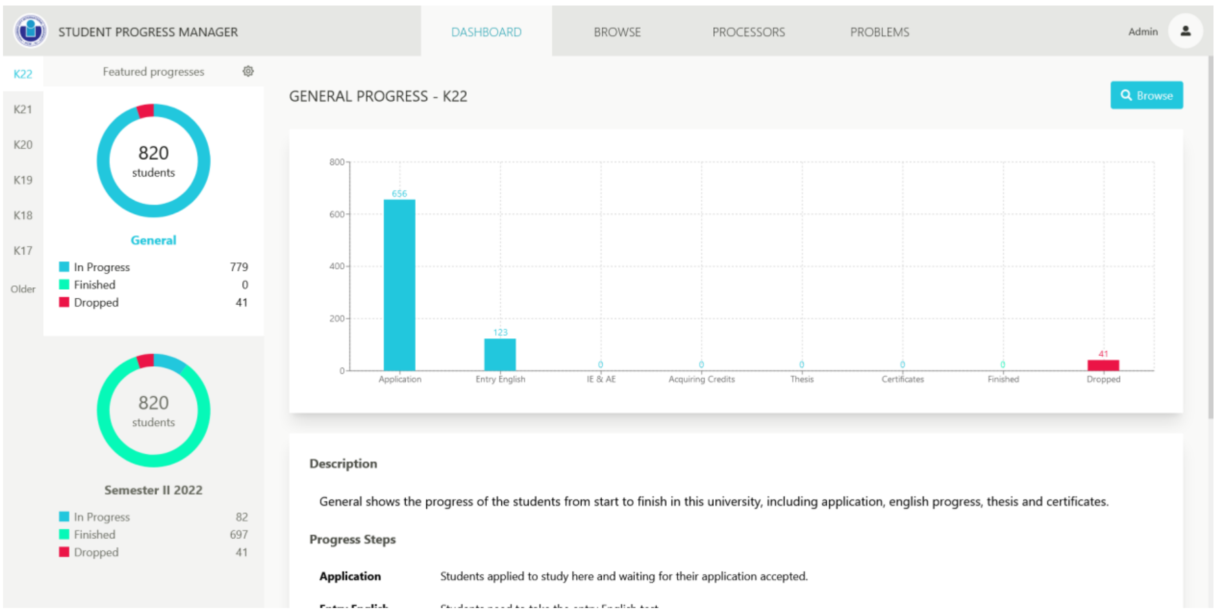
Username: admin

Password: ••••••••

Sign In

Figure 40: Login page

After this, each part of the application (dashboard, browse, processors) will be covered one by one. First, the homepage of student progress management is the dashboard as seen below. On the left-hand side of the screen is a thin vertical navigation for student years, it is used to choose the year that the students joined the university and the older button to see more years before k17 that are hidden.


STUDENT PROGRESS MANAGER | DASHBOARD | BROWSE | PROCESSORS | PROBLEMS | Admin

K22, K21, K20, K19, K18, K17, Older

Featured progresses

820 students — General

🟦 In Progress	779
🟩 Finished	0
🟥 Dropped	41

820 students — Semester II 2022

🟦 In Progress	82
🟩 Finished	697
🟥 Dropped	41

GENERAL PROGRESS - K22

Application	Entry English	IE & AE	Acquiring Credits	Thesis	Certificates	Finished	Dropped
656	123	0	0	0	0	0	41

Description

General shows the progress of the students from start to finish in this university, including application, english progress, thesis and certificates.

Progress Steps

Application Students applied to study here and waiting for their application accepted.

Figure 41: Dashboard page

To the right of that is another column containing the pie charts (General, Semester II 2022) where the featured progresses of all students are displayed. We can click on them to switch between mediators progresses that will be displayed in the main area.

In the main area, there are three components. First, in the header is the title of the selected progress (in this case general progress – k22) with the browse button to the right to quickly jump to the browse page with selected progress and year.

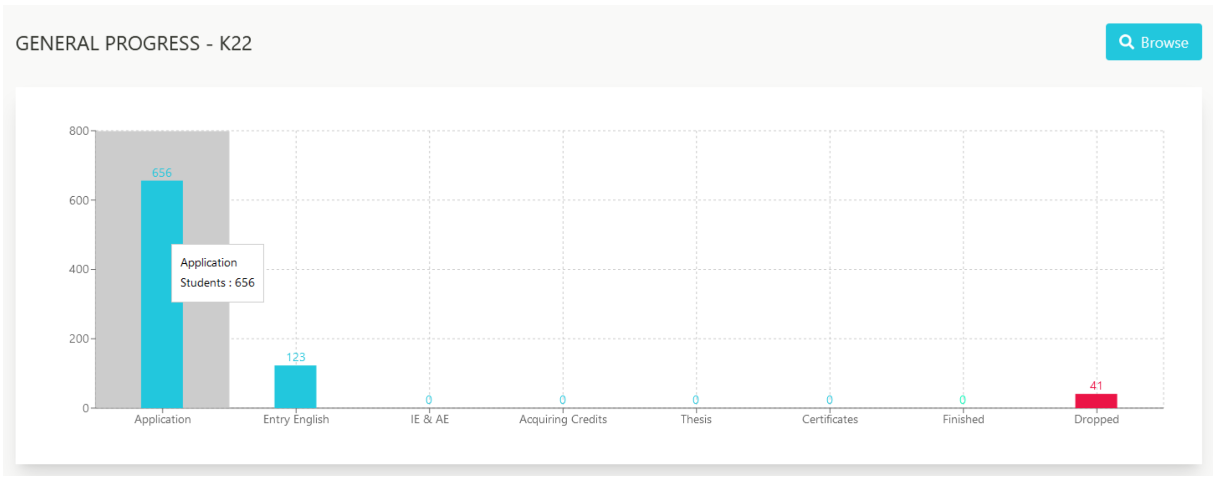
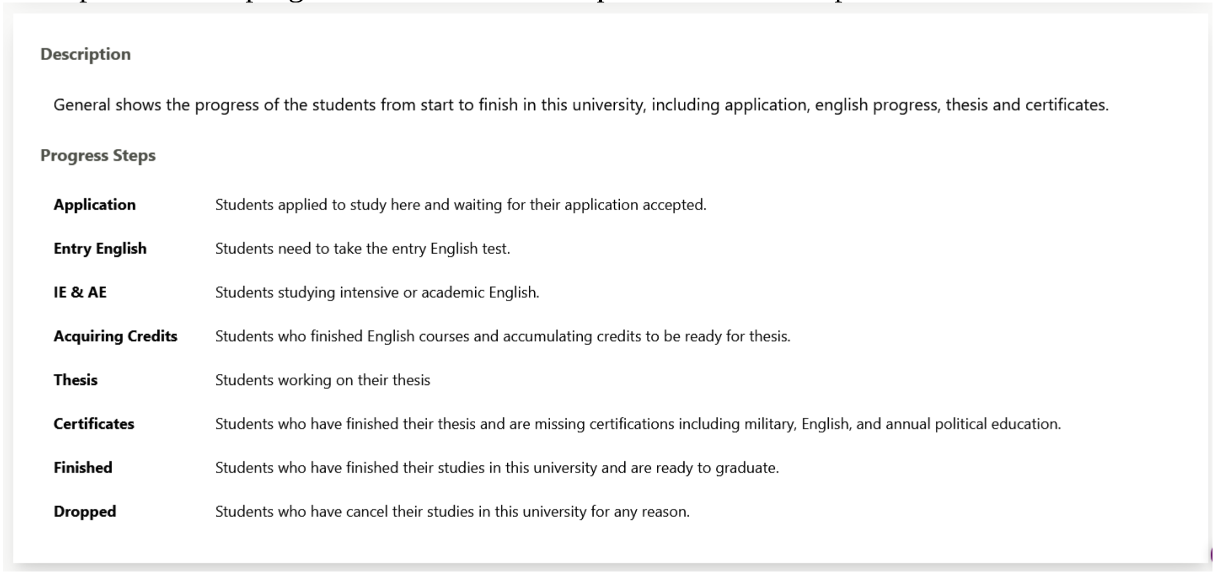


Figure 42: Student distribution on bar chart

Second, as seen in the figure, is a bar chart that displays the student distribution and hoverable using the mouse, there are currently 656 students in the Application step, 123 in Entry English and 41 Dropped.

Finally, there is the information panel to find more details for a specific progress, with the description for the progress as well as short explanation of all steps in the mediator.



Description

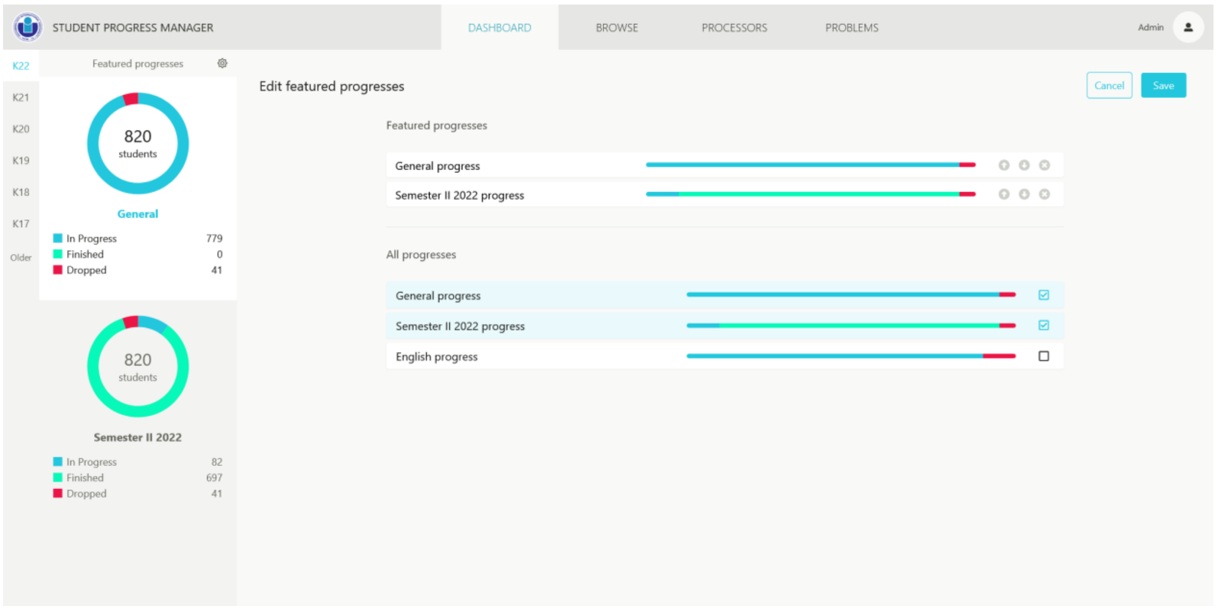
General shows the progress of the students from start to finish in this university, including application, english progress, thesis and certificates.

Progress Steps

Application	Students applied to study here and waiting for their application accepted.
Entry English	Students need to take the entry English test.
IE & AE	Students studying intensive or academic English.
Acquiring Credits	Students who finished English courses and accumulating credits to be ready for thesis.
Thesis	Students working on their thesis
Certificates	Students who have finished their thesis and are missing certifications including military, English, and annual political education.
Finished	Students who have finished their studies in this university and are ready to graduate.
Dropped	Students who have cancel their studies in this university for any reason.

Figure 43: Progress information

To change which progresses display, click on the gear button above the pie charts. We can select and deselect, as well as modifying their order, then click Save to update the board:



Edit featured progresses

Featured progresses

- General progress
- Semester II 2022 progress

All progresses

- General progress ☑
- Semester II 2022 progress ☑
- English progress ☐

820 students

General

🟦 In Progress	779
🟩 Finished	0
🟥 Dropped	41

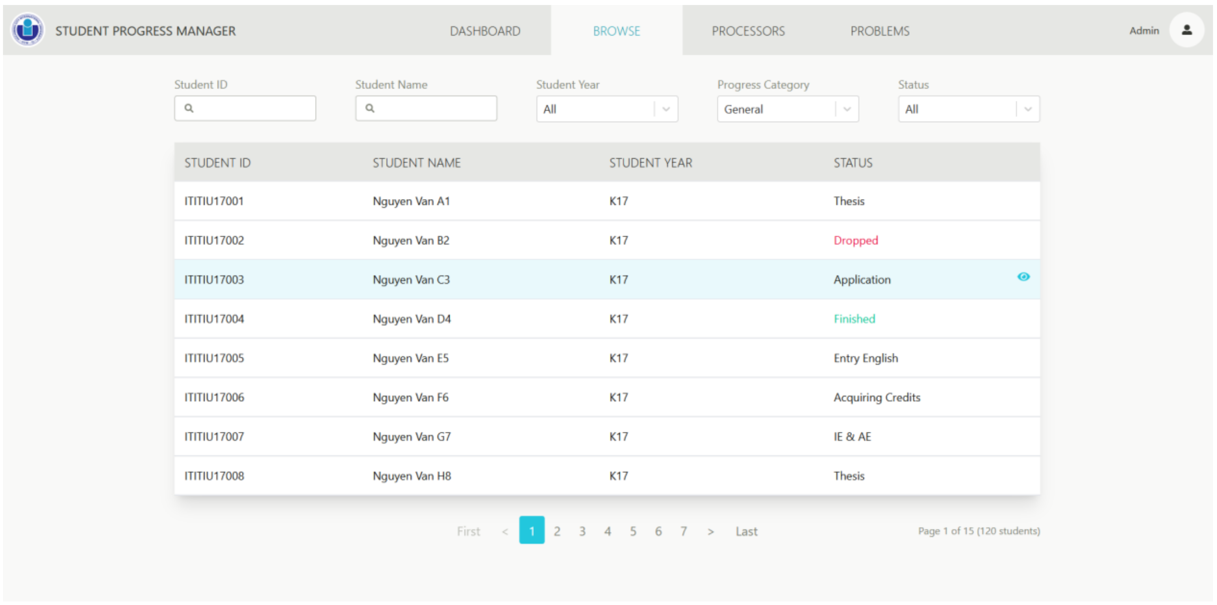
820 students

Semester II 2022

🟦 In Progress	82
🟩 Finished	697
🟥 Dropped	41

Figure 44: Edit featured progresses

Navigating to the browse tab, there is a list of students along with their current statuses with respect to the selected progress category:



Student ID:
Student Name:
Student Year: All
Progress Category: General
Status: All

STUDENT ID	STUDENT NAME	STUDENT YEAR	STATUS
ITITIU17001	Nguyen Van A1	K17	Thesis
ITITIU17002	Nguyen Van B2	K17	Dropped
ITITIU17003	Nguyen Van C3	K17	Application
ITITIU17004	Nguyen Van D4	K17	Finished
ITITIU17005	Nguyen Van E5	K17	Entry English
ITITIU17006	Nguyen Van F6	K17	Acquiring Credits
ITITIU17007	Nguyen Van G7	K17	IE & AE
ITITIU17008	Nguyen Van H8	K17	Thesis

Page 1 of 15 (120 students)

Figure 45: Browse page

Now there are five filtering options: Student ID, Student Name, Progress Category and Status to find the desired record. Progress Category includes all the progress types that corresponds to each mediator (general, semester, english, and so on). Status means the current step of the student in that mediator. The list will be returned with paging option.

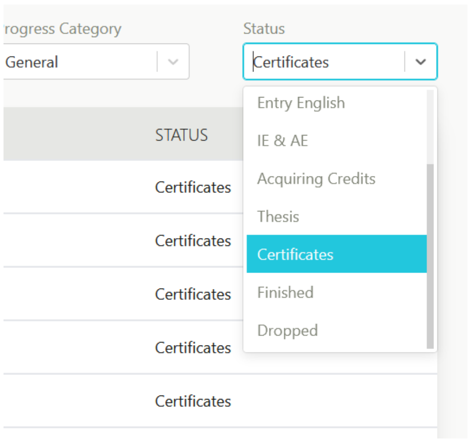
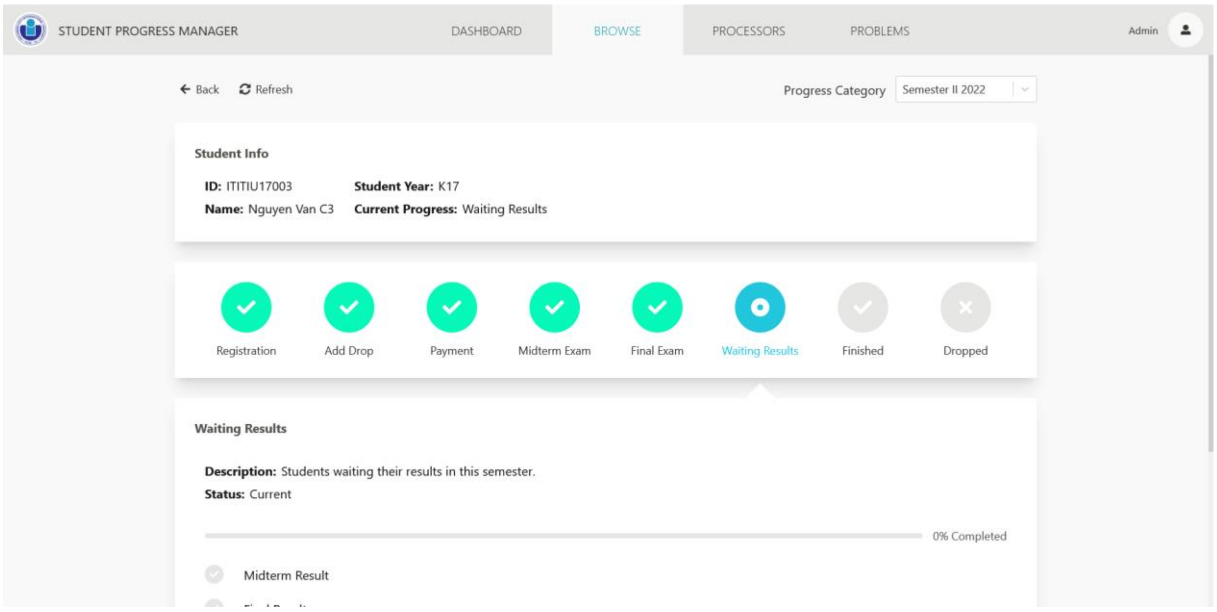


Figure 46: Selecting filtering

When a record is selected, the page is redirected to the detailed progress as seen below:



Progress Category: Semester II 2022

Student Info

ID: ITITIU17003 **Student Year:** K17

Name: Nguyen Van C3 **Current Progress:** Waiting Results

Waiting Results

Description: Students waiting their results in this semester.

Status: Current

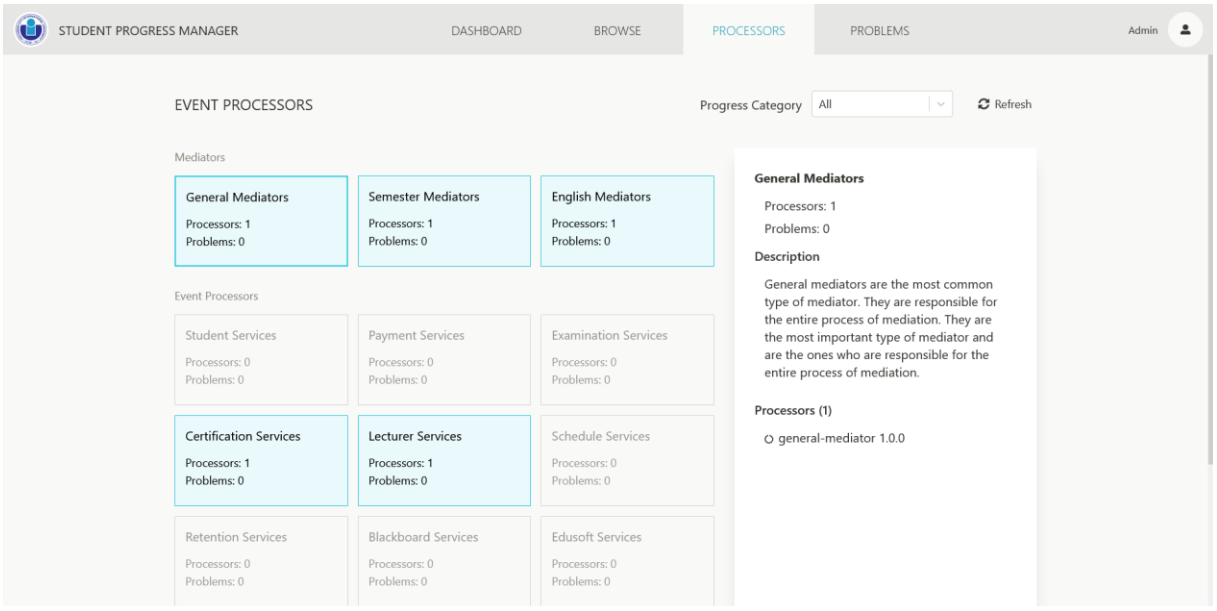
0% Completed

Midterm Result

Figure 47: View student progress

The student's related info is displayed on top, under that we can view the steps progress swith green check mark means completed and blue circle means it is the current step. Below them is another section for more detailed information about the current step like description, list of smaller items to be finished and the percentage. To change progress type, click on the select Progress Category on the top right corner.

Finally, the processors page is used to show different connected mediators and services:


STUDENT PROGRESS MANAGER

DASHBOARD BROWSE PROCESSORS PROBLEMS Admin

EVENT PROCESSORS

Progress Category: All

Refresh

Mediators

General Mediators
Processors: 1
Problems: 0

Semester Mediators
Processors: 1
Problems: 0

English Mediators
Processors: 1
Problems: 0

Event Processors

Student Services
Processors: 0
Problems: 0

Payment Services
Processors: 0
Problems: 0

Examination Services
Processors: 0
Problems: 0

Certification Services
Processors: 1
Problems: 0

Lecturer Services
Processors: 1
Problems: 0

Schedule Services
Processors: 0
Problems: 0

Retention Services
Processors: 0
Problems: 0

Blackboard Services
Processors: 0
Problems: 0

Edusoft Services
Processors: 0
Problems: 0

General Mediators
Processors: 1
Problems: 0

Description

General mediators are the most common type of mediator. They are responsible for the entire process of mediation. They are the most important type of mediator and are the ones who are responsible for the entire process of mediation.

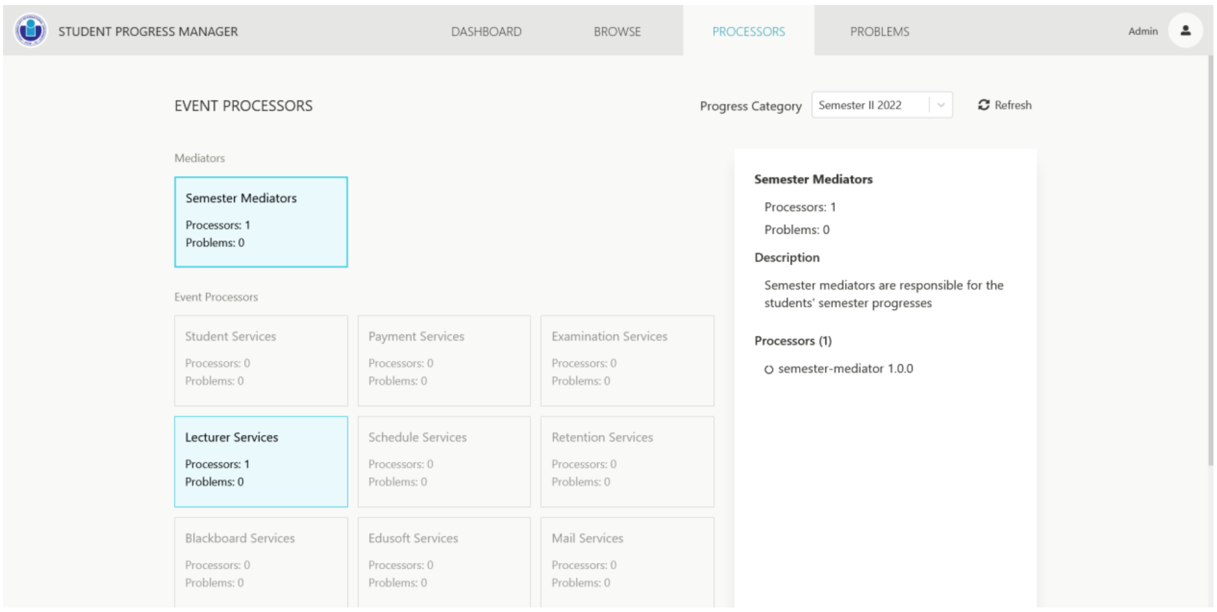
Processors (1)

- general-mediator 1.0.0

Figure 48: Processors page

Here they are divided by service groups, the service manager has stored different service groups like Student services, payment services, etc. And for each service registered to the manager, it increases the processors count by one. The blue rectangles are the ones with at least one processor, the grayed ones have no processors at the moment.
On the right-hand side is a panel that display more information such as descriptions and the list of processors registered to the group.

Furthermore, we can use the Progress Category select to filter to only the services related to the selected mediator.


STUDENT PROGRESS MANAGER

DASHBOARD BROWSE PROCESSORS PROBLEMS Admin

EVENT PROCESSORS

Progress Category: Semester II 2022

Refresh

Mediators

Semester Mediators
Processors: 1
Problems: 0

Event Processors

Student Services
Processors: 0
Problems: 0

Payment Services
Processors: 0
Problems: 0

Examination Services
Processors: 0
Problems: 0

Lecturer Services
Processors: 1
Problems: 0

Schedule Services
Processors: 0
Problems: 0

Retention Services
Processors: 0
Problems: 0

Blackboard Services
Processors: 0
Problems: 0

Edusoft Services
Processors: 0
Problems: 0

Mail Services
Processors: 0
Problems: 0

Semester Mediators
Processors: 1
Problems: 0

Description

Semester mediators are responsible for the students' semester progresses

Processors (1)

- semester-mediator 1.0.0

Figure 49: Filtered processors

4.2.2. Mediator demo

As mentioned before, we have implemented two mediators for Semester and General progresses, in this section we will look at semester flow that includes course service, lecturer service and semester mediator:

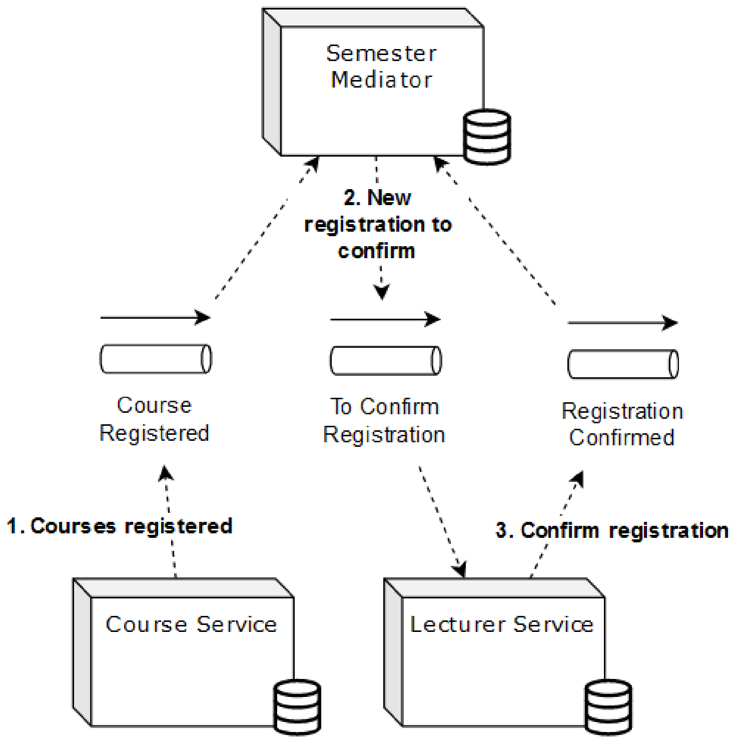
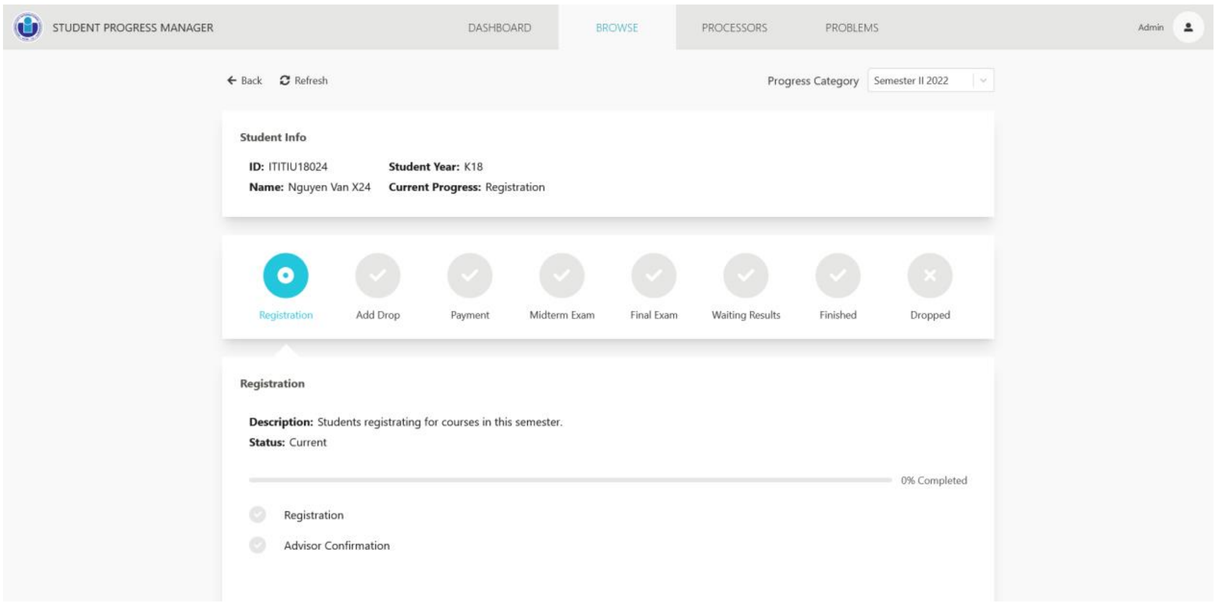


Figure 50: Semester mediator scenario

This has three stages, corresponding to three events: registering courses by the course service, creating new to-confirm registrations by the semester mediator, and confirming registration by the lecturer service. This demo will walk through them.

To begin, a student in registration step is picked and shown in the below figure:



Student Info

ID: ITITIU18024 **Student Year:** K18
Name: Nguyen Van X24 **Current Progress:** Registration

Registration

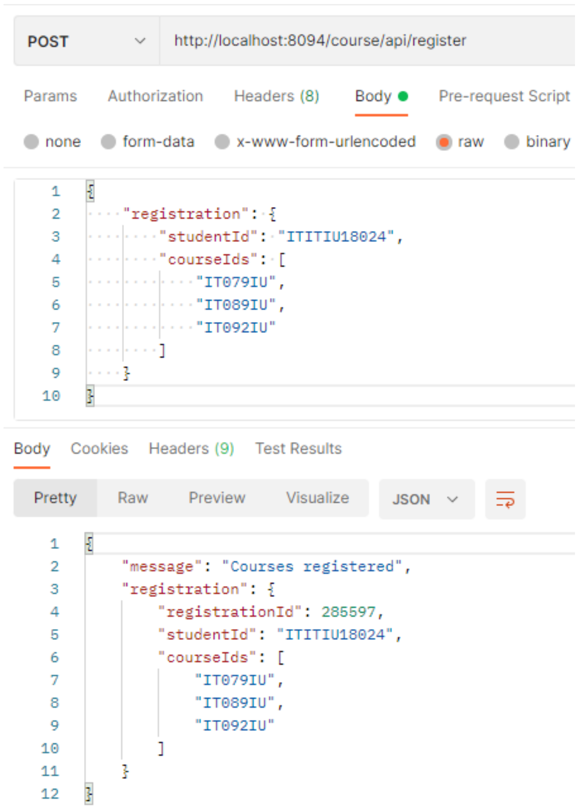
Description: Students registrating for courses in this semester.
Status: Current

0% Completed

Registration
Advisor Confirmation

Figure 51: Student at registration step

We have two tasks for this step, namely *Registration* and *Avisor Confirmation*. The first task is done by the student using the existing website, and the second is by the advisor. In this demo, we simulate the registration by using Postman and request the course service like this:



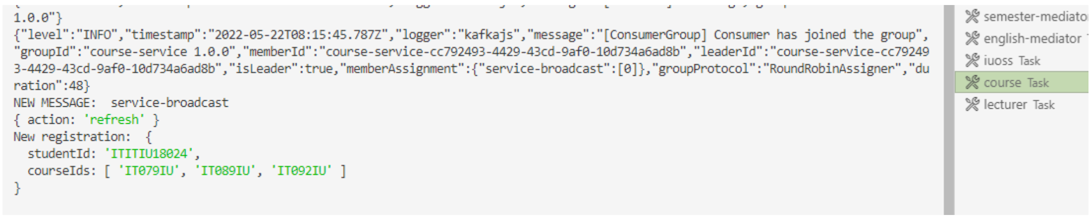
```
POST http://localhost:8094/course/api/register
```

```
{
    "registration": {
        "studentId": "ITITIU18024",
        "courseIds": [
            "IT079IU",
            "IT089IU",
            "IT092IU"
        ]
    }
}
```

```
{
    "message": "Courses registered",
    "registration": {
        "registrationId": 285597,
        "studentId": "ITITIU18024",
        "courseIds": [
            "IT079IU",
            "IT089IU",
            "IT092IU"
        ]
    }
}
```

Figure 52: Register courses using Postman

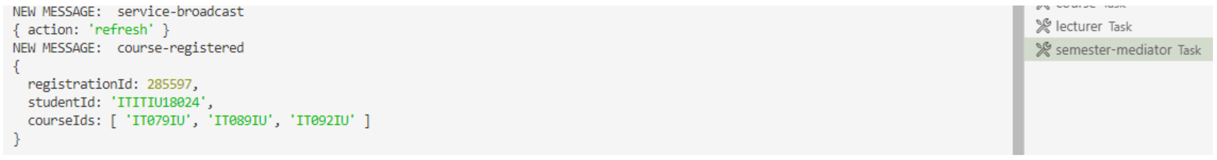
The payload includes a student id and the list of courses, the course service receives the registration, does what it needs to do with the data, and then publish a new event to the *course-registered* topic.



```
1.0.0"}
{"level":"INFO","timestamp":"2022-05-22T08:15:45.787Z","logger":"kafkajs","message":"[ConsumerGroup] Consumer has joined the group",
"groupId":"course-service 1.0.0","memberId":"course-service-cc792493-4429-43cd-9af0-10d734a6ad8b","leaderId":"course-service-cc79249
3-4429-43cd-9af0-10d734a6ad8b","isLeader":true,"memberAssignment":{"service-broadcast":[0]},"groupProtocol":"RoundRobinAssigner","du
ration":48}
NEW MESSAGE:  service-broadcast
{ action: 'refresh' }
New registration:  {
  studentId: 'ITITIU18024',
  courseIds: [ 'IT079IU', 'IT089IU', 'IT092IU' ]
}
```

Figure 53: Course service received registration

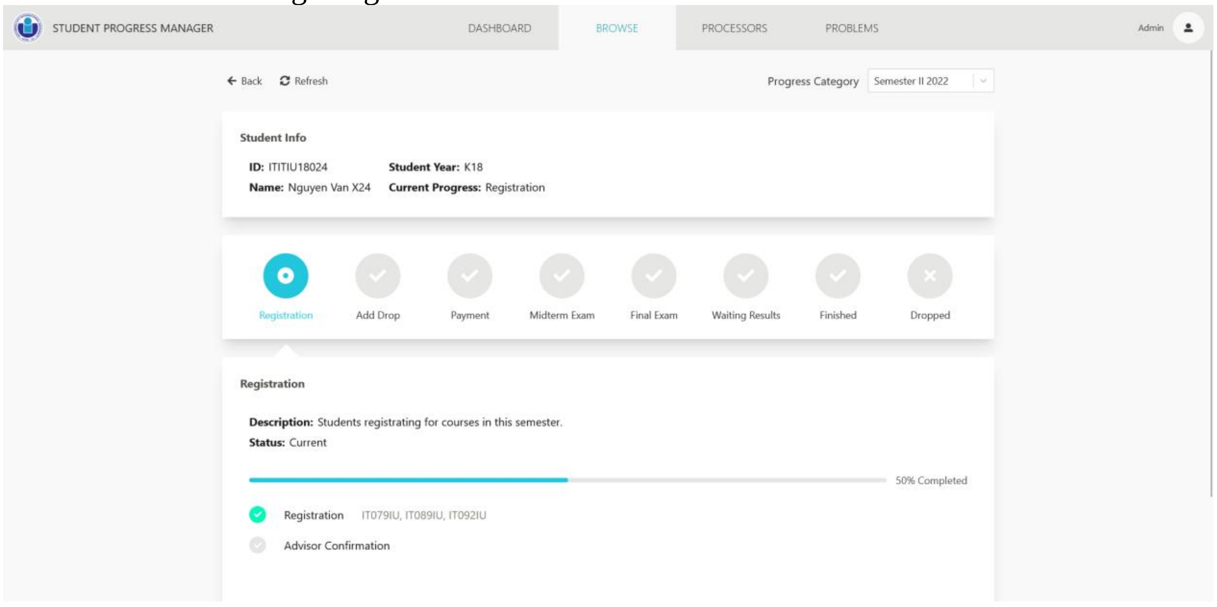
After that, semester mediator will immediately consume the message and update the status of the student.



```
NEW MESSAGE:  service-broadcast
{ action: 'refresh' }
NEW MESSAGE:  course-registered
{
  registrationId: 285597,
  studentId: 'ITITIU18024',
  courseIds: [ 'IT079IU', 'IT089IU', 'IT092IU' ]
}
```

Figure 54: Semester mediator consumes course registration event

When we refresh the page, the task of *Registration* is finished and progress bar jumps to 50% as seen in the following image:



Progress Category: Semester II 2022

Student Info

ID: ITITIU18024 **Student Year:** K18
Name: Nguyen Van X24 **Current Progress:** Registration

Registration

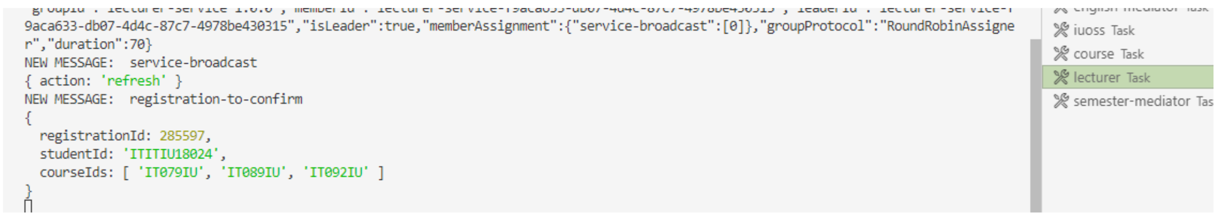
Description: Students registrating for courses in this semester.
Status: Current

50% Completed

✓ Registration IT079IU, IT089IU, IT092IU
Advisor Confirmation

Figure 55: Student's registration progress updated

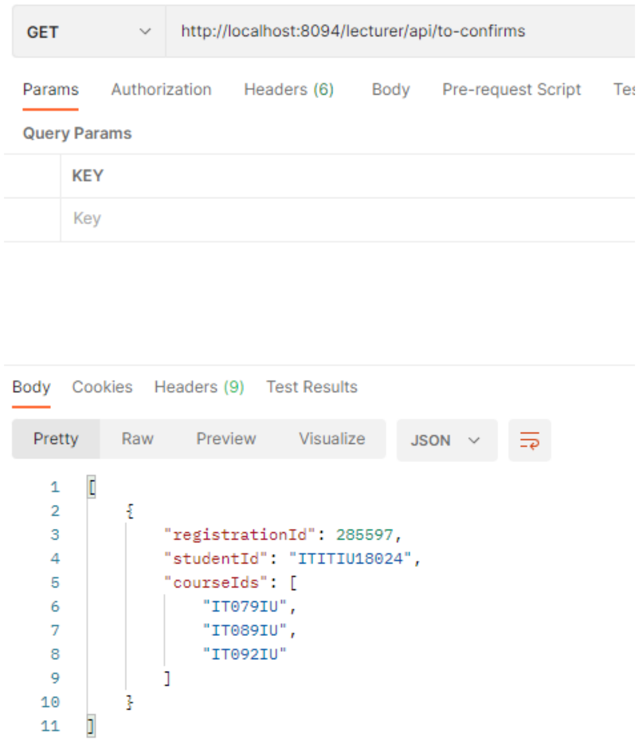
In addition, as a responsibility of a mediator to control the flow of the system, after having listened to the registration message, the semester mediator will publish another one to *registration-to-confirm* topic, which will be consumed by the lecturer service.



```
9aca633-db07-4d4c-87c7-4978be430315","isLeader":true,"memberAssignment":{"service-broadcast":[0]},"groupProtocol":"RoundRobinAssigne
r","duration":70}
NEW MESSAGE:  service-broadcast
{ action: 'refresh' }
NEW MESSAGE:  registration-to-confirm
{
  registrationId: 285597,
  studentId: 'ITITIU18024',
  courseIds: [ 'IT079IU', 'IT089IU', 'IT092IU' ]
}
```

Figure 56: Lecturer service receives request event from mediator

Next, the lecturer service receives the request in the queue and update the data, if we get the list of all registrations to be confirmed we will see this:

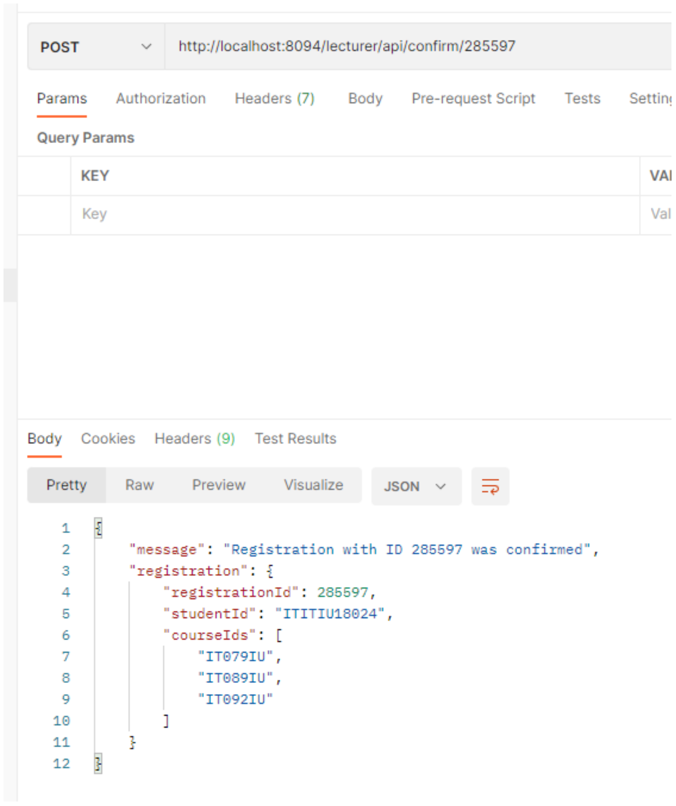


http://localhost:8094/lecturer/api/to-confirms

```
[
    {
        "registrationId": 285597,
        "studentId": "ITITIU18024",
        "courseIds": [
            "IT079IU",
            "IT089IU",
            "IT092IU"
        ]
    }
]
```

Figure 57: Getting registrations to be confirmed

And to complete the flow, we can simulate confirming a registration by using a post request to the lecturer service:

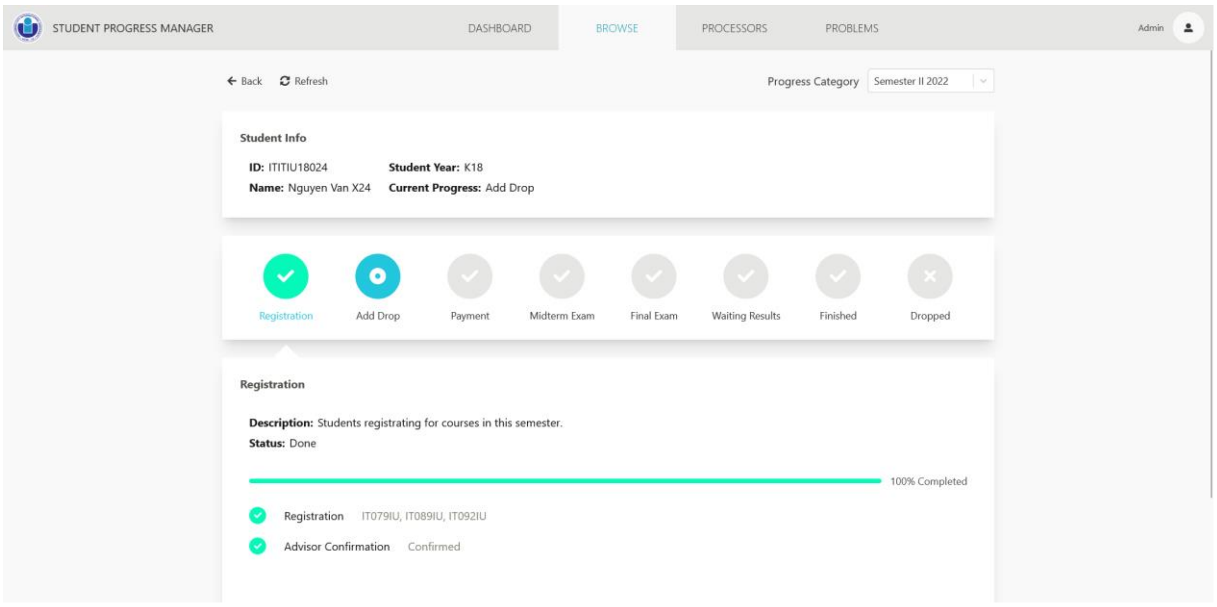


http://localhost:8094/lecturer/api/confirm/285597

```
{
    "message": "Registration with ID 285597 was confirmed",
    "registration": {
        "registrationId": 285597,
        "studentId": "ITITIU18024",
        "courseIds": [
            "IT079IU",
            "IT089IU",
            "IT092IU"
        ]
    }
}
```

Figure 58: Confirming registration

After that, refresh the page and the *Registration* progress of the student should be 100% and it moves to the next step which is *Add Drop*:



Student Info

ID: ITITIU18024 **Student Year:** K18
Name: Nguyen Van X24 **Current Progress:** Add Drop

Registration

Description: Students registrating for courses in this semester.
Status: Done

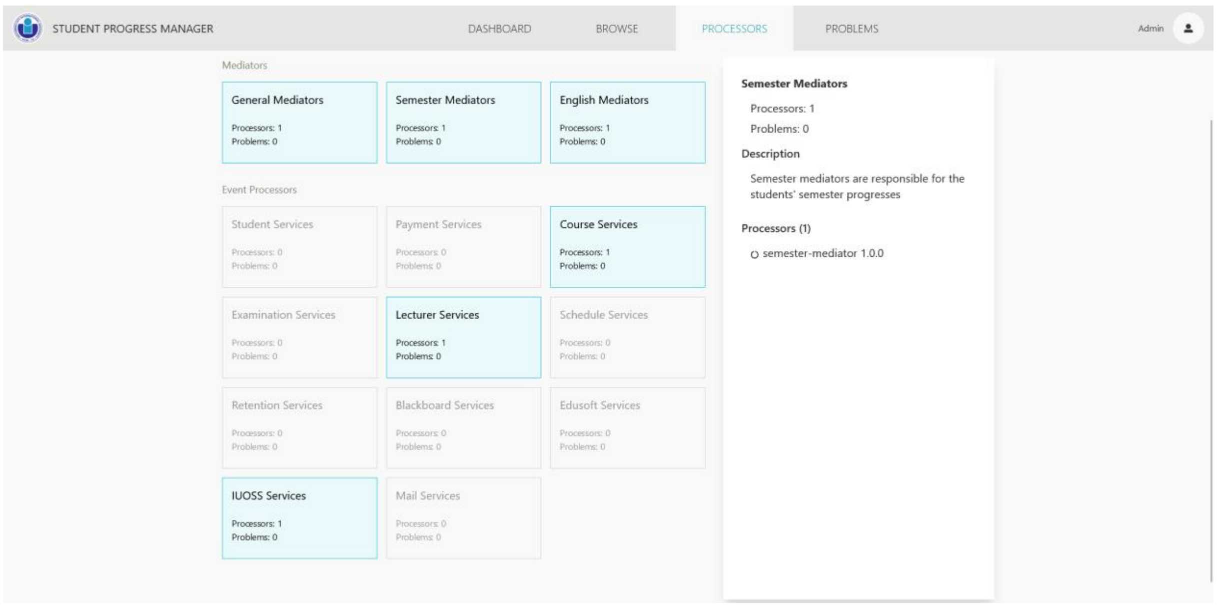
100% Completed

Registration	IT079IU, IT089IU, IT092IU
Advisor Confirmation	Confirmed

Figure 59: Registration step finished

4.2.3. Adding new services

This section will demonstrate how to connect to a new service in the system. Let’s go back to processors page, we are going to add a service group for certification as it is missing currently:



Mediators

Group	Processors	Problems
General Mediators	1	0
Semester Mediators	1	0
English Mediators	1	0

Event Processors

Group	Processors	Problems
Student Services	0	0
Payment Services	0	0
Course Services	1	0
Examination Services	0	0
Lecturer Services	1	0
Schedule Services	0	0
Retention Services	0	0
Blackboard Services	0	0
Edusoft Services	0	0
IUOSS Services	1	0
Mail Services	0	0

Semester Mediators

Processors: 1
Problems: 0

Description

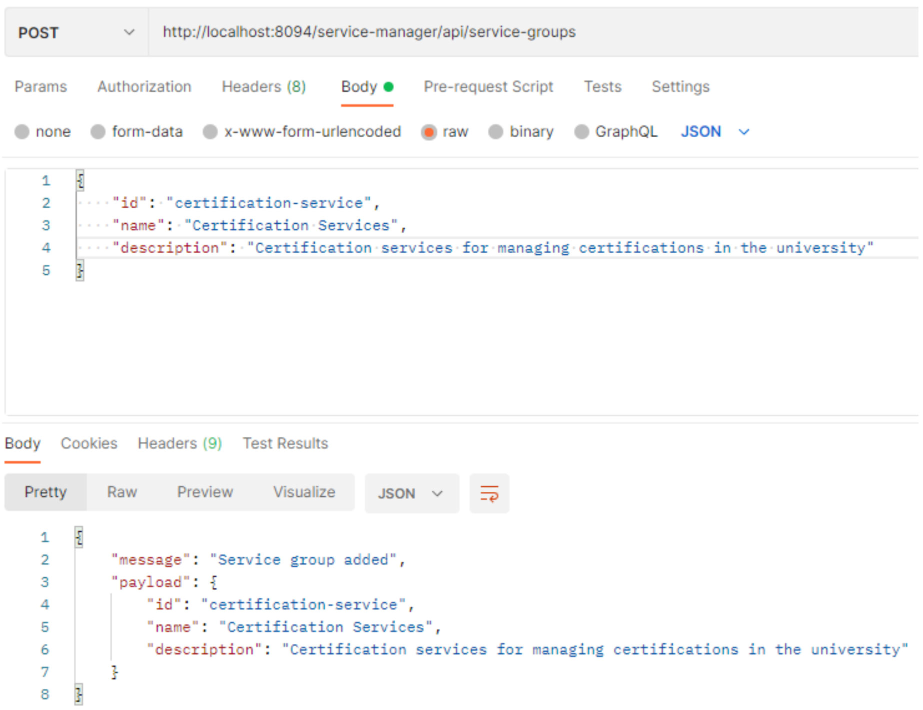
Semester mediators are responsible for the students' semester progresses

Processors (1)

semester-mediator 1.0.0

Figure 60: Proccessors page

To do this, again, Postman is used to request a new group with id, name and description:



POST http://localhost:8094/service-manager/api/service-groups

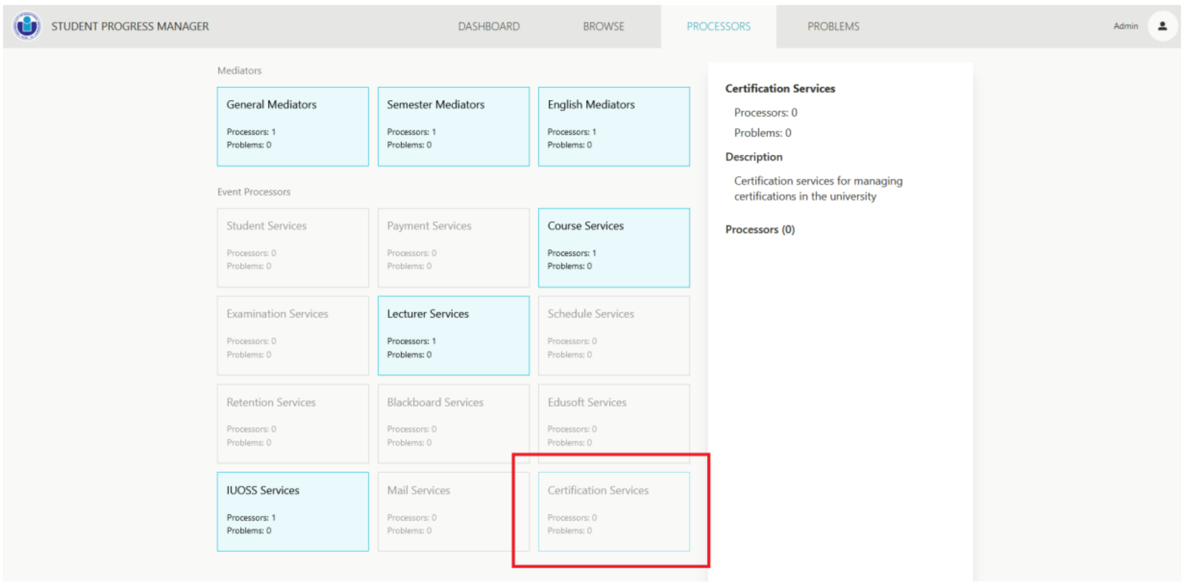
```
{
    "id": "certification-service",
    "name": "Certification Services",
    "description": "Certification services for managing certifications in the university"
}
```

Response Body (Pretty, JSON):

```
{
    "message": "Service group added",
    "payload": {
        "id": "certification-service",
        "name": "Certification Services",
        "description": "Certification services for managing certifications in the university"
    }
}
```

Figure 61: Adding certification service group to service manager

Refreshing the page, we can see that the certification services group appears under the screen like this:



STUDENT PROGRESS MANAGER — DASHBOARD, BROWSE, PROCESSORS, PROBLEMS — Admin

Mediators

General Mediators Processors: 1 Problems: 0	Semester Mediators Processors: 1 Problems: 0	English Mediators Processors: 1 Problems: 0

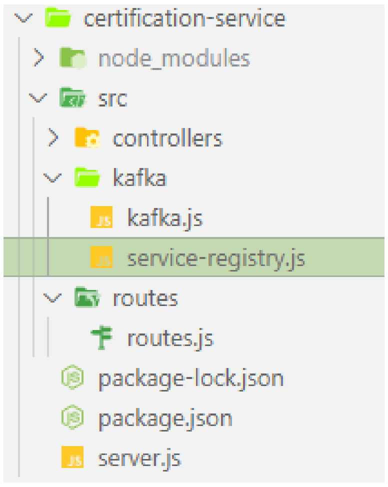
Event Processors

Student Services Processors: 0 Problems: 0	Payment Services Processors: 0 Problems: 0	Course Services Processors: 1 Problems: 0
Examination Services Processors: 0 Problems: 0	Lecturer Services Processors: 1 Problems: 0	Schedule Services Processors: 0 Problems: 0
Retention Services Processors: 0 Problems: 0	Blackboard Services Processors: 0 Problems: 0	Edusoft Services Processors: 0 Problems: 0
IUOSS Services Processors: 1 Problems: 0	Mail Services Processors: 0 Problems: 0	Certification Services Processors: 0 Problems: 0

Certification Services
Processors: 0
Problems: 0
Description
Certification services for managing certifications in the university
Processors (0)

Figure 62: Certification Service group added

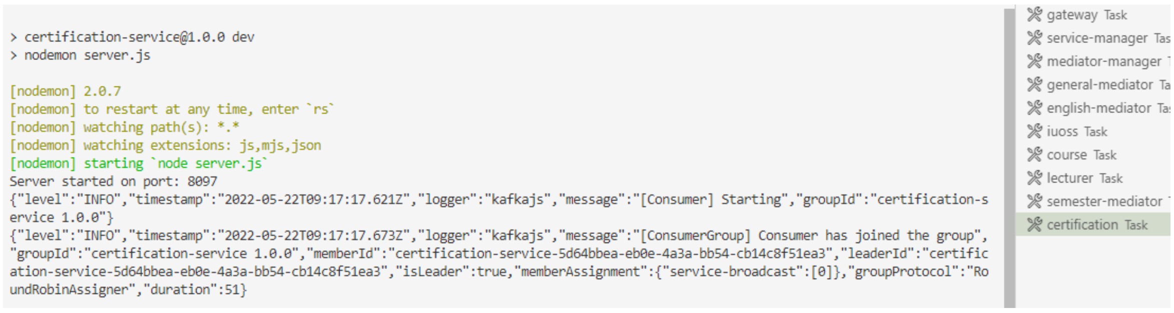
Next, to connect to an actual certification service, it needs to have *kafka.js* and *service-registry.js* to communicate with the service manager through Kafka. The *service-group-id* is configured to match to the ID of the recent added group.



```
certification-service
  node_modules
  src
    controllers
    kafka
      kafka.js
      service-registry.js
    routes
      routes.js
  package-lock.json
  package.json
  server.js
```

Figure 63: Certification service folder

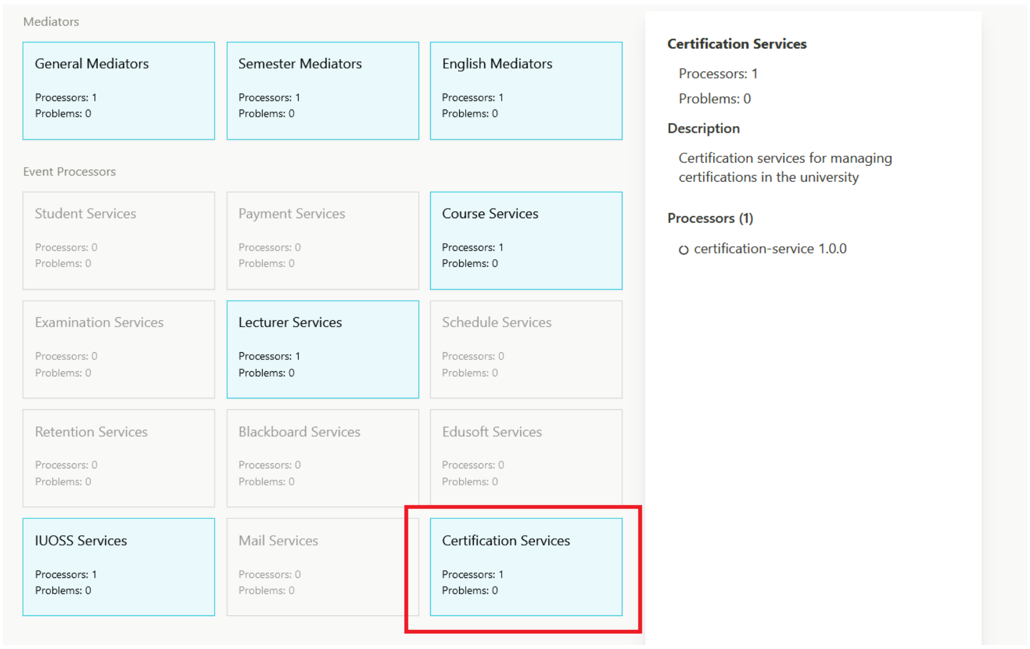
Finally, run server.js and it will automatically reach out and registered itself to the server.



```
> certification-service@1.0.0 dev
> nodemon server.js

[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node server.js`
Server started on port: 8097
{"level":"INFO","timestamp":"2022-05-22T09:17:17.621Z","logger":"kafkajs","message":"[Consumer] Starting","groupId":"certification-s
ervice 1.0.0"}
{"level":"INFO","timestamp":"2022-05-22T09:17:17.673Z","logger":"kafkajs","message":"[ConsumerGroup] Consumer has joined the group",
"groupId":"certification-service 1.0.0","memberId":"certification-service-5d64bbea-eb0e-4a3a-bb54-cb14c8f51ea3","leaderId":"certific
ation-service-5d64bbea-eb0e-4a3a-bb54-cb14c8f51ea3","isLeader":true,"memberAssignment":{"service-broadcast":[0]},"groupProtocol":"Ro
undRobinAssigner","duration":51}
```

Figure 64: Certification service running

We can see the Certification Services rectangle is now lighting up and ready to process requests:



Mediators

General Mediators
Processors: 1
Problems: 0

Semester Mediators
Processors: 1
Problems: 0

English Mediators
Processors: 1
Problems: 0

Event Processors

Student Services
Processors: 0
Problems: 0

Payment Services
Processors: 0
Problems: 0

Course Services
Processors: 1
Problems: 0

Examination Services
Processors: 0
Problems: 0

Lecturer Services
Processors: 1
Problems: 0

Schedule Services
Processors: 0
Problems: 0

Retention Services
Processors: 0
Problems: 0

Blackboard Services
Processors: 0
Problems: 0

Edusoft Services
Processors: 0
Problems: 0

IUOSS Services
Processors: 1
Problems: 0

Mail Services
Processors: 0
Problems: 0

Certification Services
Processors: 1
Problems: 0

Certification Services

Processors: 1

Problems: 0

Description

Certification services for managing certifications in the university

Processors (1)

○ certification-service 1.0.0

Figure 65: Certification Services working

Next, let's follow the general mediator scenario to demonstrate the parallel execution ability as well as to test out the newly connected certificate service:

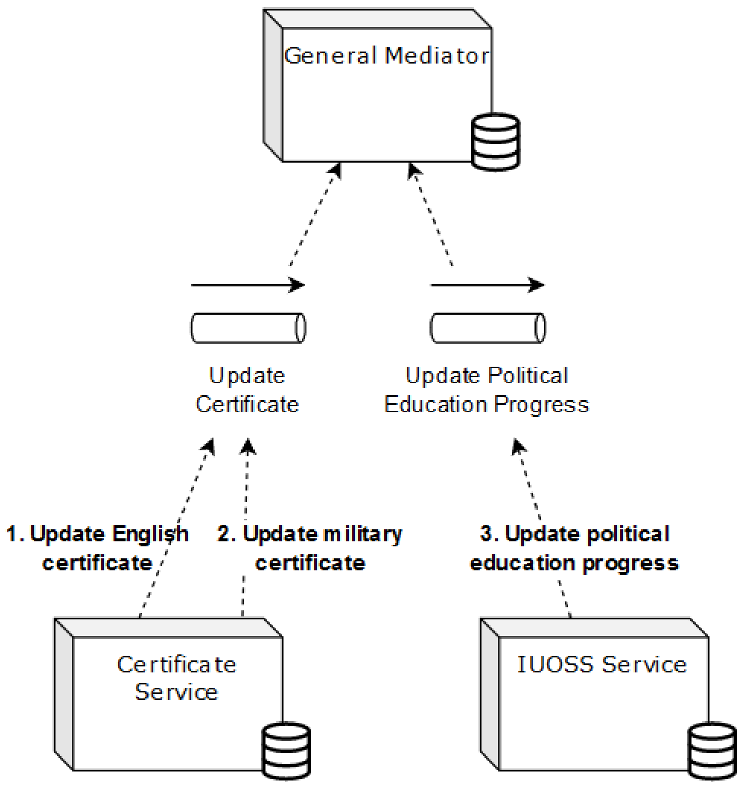
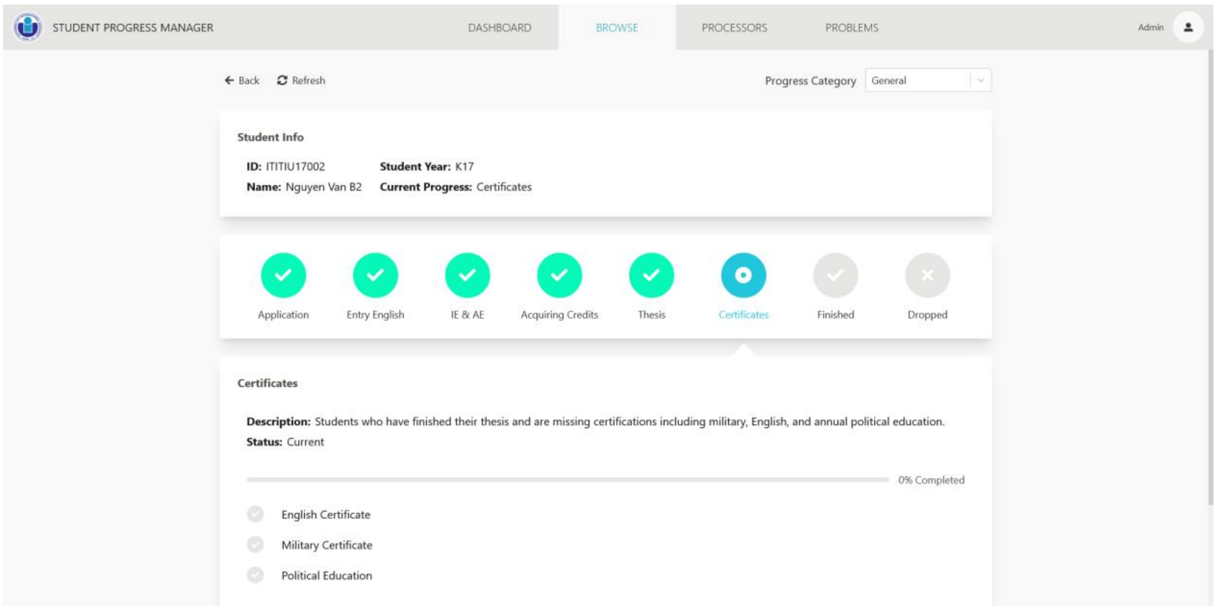

General Mediator
Update Certificate
Update Political Education Progress
1. Update English certificate
2. Update military certificate
3. Update political education progress
Certificate Service
IUOSS Service

Figure 66: General mediator scenario

This scenario focuses on the *Certificates* step of the student, there are three requirements need to be met before proceeding to the next step: an English certificate, a military certificate, and political education. The order of them does not matter because they can happen in parallel.

To get started, select a student at *Certificates* step in *General Mediator*:



Progress Category: General

Student Info

ID: ITITIU17002 **Student Year:** K17
Name: Nguyen Van B2 **Current Progress:** Certificates

Certificates

Description: Students who have finished their thesis and are missing certifications including military, English, and annual political education.
Status: Current

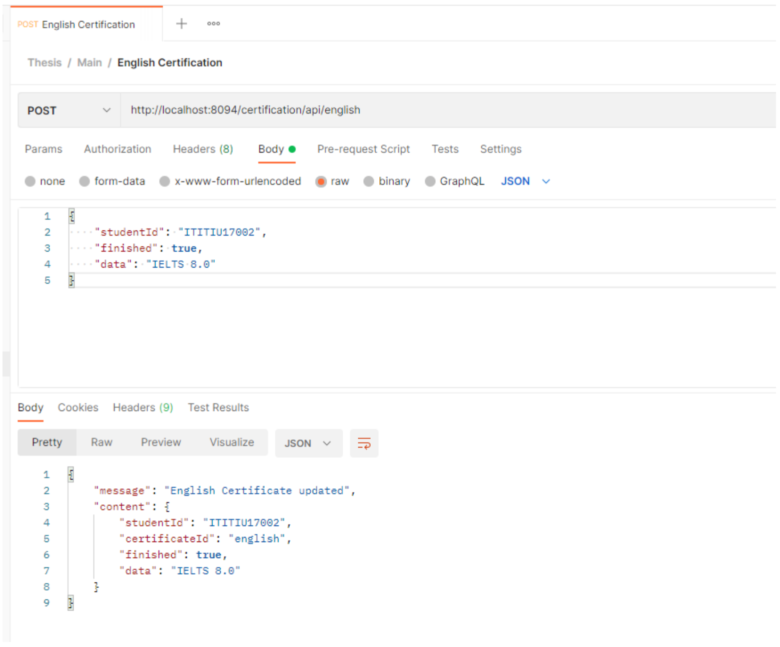
0% Completed

English Certificate
Military Certificate
Political Education

Figure 67: Student progress at certificates step

We can see there are three items under the step descriptions, that are three types of certificates.

Like semester mediator, we can use Postman to send requests to certification service one by one:



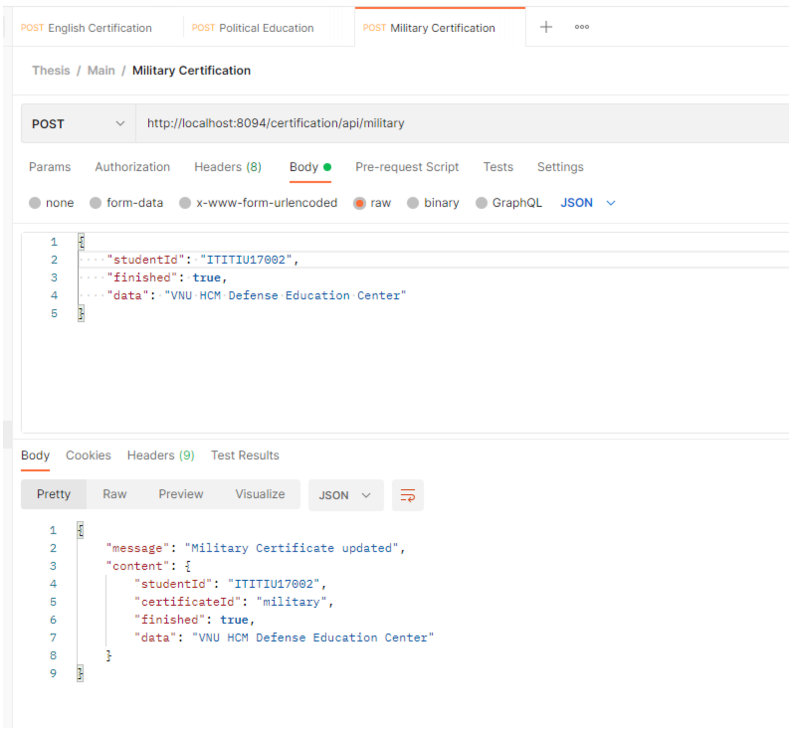
POST http://localhost:8094/certification/api/english

```
{
    "studentId": "ITITIU17002",
    "finished": true,
    "data": "IELTS 8.0"
}
```

Response Body: JSON

```
{
    "message": "English Certificate updated",
    "content": {
        "studentId": "ITITIU17002",
        "certificateId": "english",
        "finished": true,
        "data": "IELTS 8.0"
    }
}
```

Figure 68: Updating english certificate



POST http://localhost:8094/certification/api/military

Body: raw, JSON

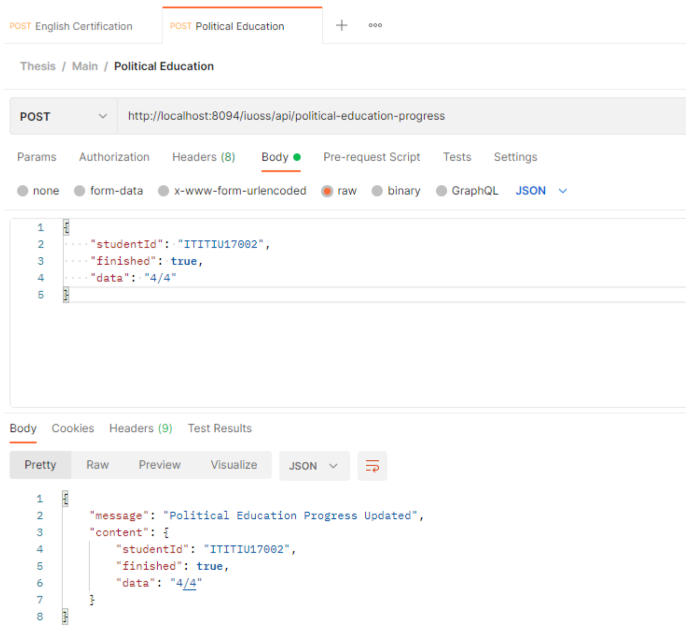
```
{
    "studentId": "ITITIU17002",
    "finished": true,
    "data": "VNU HCM Defense Education Center"
}
```

Response (Pretty, JSON):

```
{
    "message": "Military Certificate updated",
    "content": {
        "studentId": "ITITIU17002",
        "certificateId": "military",
        "finished": true,
        "data": "VNU HCM Defense Education Center"
    }
}
```

Figure 69: Updating military certificate

The third certificate item (political education progress) belongs to the iuoss service, so do it similarly:



POST http://localhost:8094/iuoss/api/political-education-progress

Body: raw, JSON

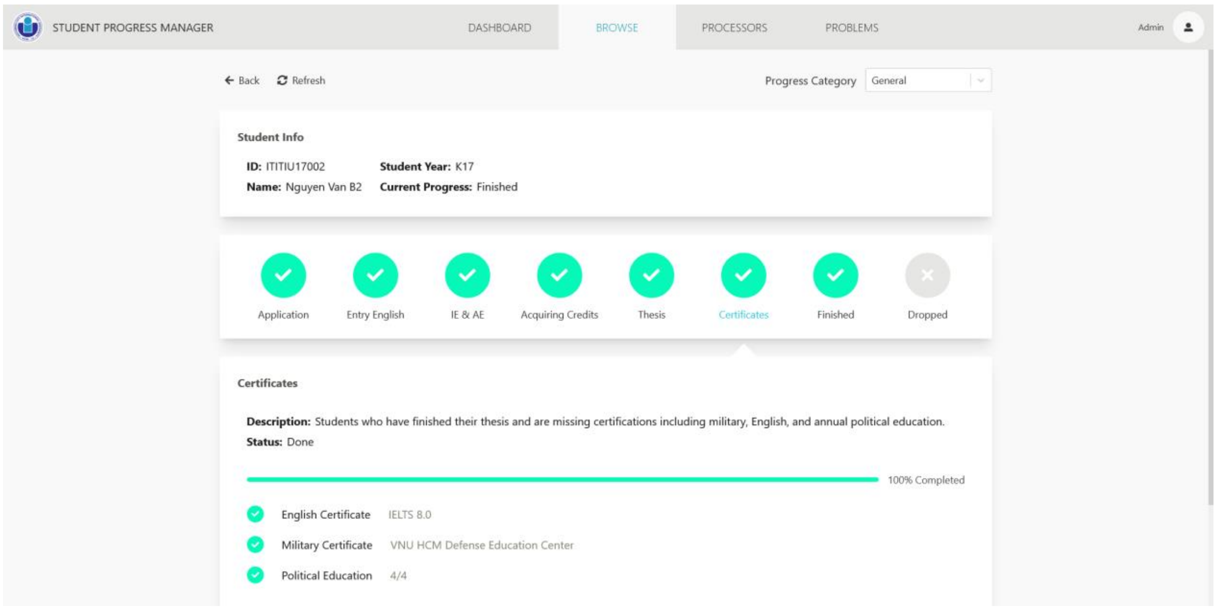
```
{
    "studentId": "ITITIU17002",
    "finished": true,
    "data": "4/4"
}
```

Response (Pretty, JSON):

```
{
    "message": "Political Education Progress Updated",
    "content": {
        "studentId": "ITITIU17002",
        "finished": true,
        "data": "4/4"
    }
}
```

Figure 70: Updating politcal education

And this is the result, the *Certification* step is finished:


STUDENT PROGRESS MANAGER

Student Info

ID: ITITIU17002 **Student Year:** K17
Name: Nguyen Van B2 **Current Progress:** Finished

Certificates

Description: Students who have finished their thesis and are missing certifications including military, English, and annual political education.
Status: Done

100% Completed

English Certificate	IELTS 8.0
Military Certificate	VNU HCM Defense Education Center
Political Education	4/4

Figure 71: Certification step completed

This is the logs of general mediator and it has received all three events:

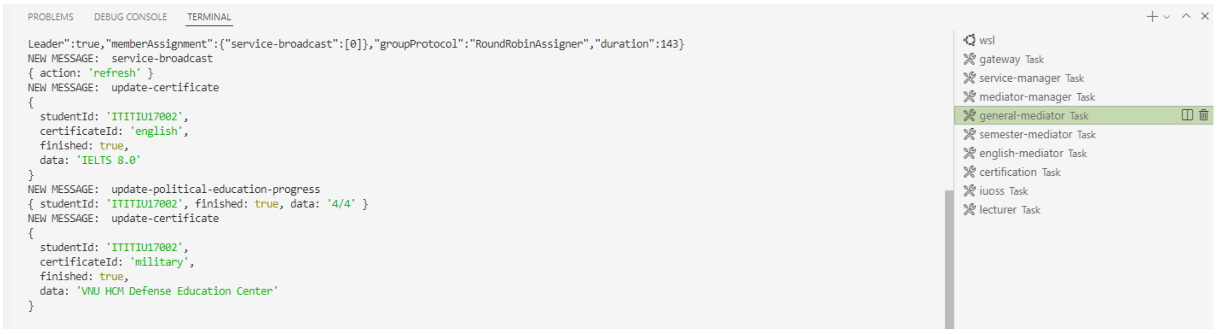

```
Leader":true,"memberAssignment":{"service-broadcast":[0]},"groupProtocol":"RoundRobinAssigner","duration":143}
NEW MESSAGE:  service-broadcast
{ action: 'refresh' }
NEW MESSAGE:  update-certificate
{
  studentId: 'ITITIU17002',
  certificateId: 'english',
  finished: true,
  data: 'IELTS 8.0'
}
NEW MESSAGE:  update-political-education-progress
{ studentId: 'ITITIU17002', finished: true, data: '4/4' }
NEW MESSAGE:  update-certificate
{
  studentId: 'ITITIU17002',
  certificateId: 'military',
  finished: true,
  data: 'VNU HCM Defense Education Center'
}
```

Figure 72: General mediator logs

Next is the demonstration on error handling by server manager.

4.2.4. Error handling

Now, to illustrate the capability of handling errors in the system, let’s revisit the diagram from earlier that was about the case of course registration:

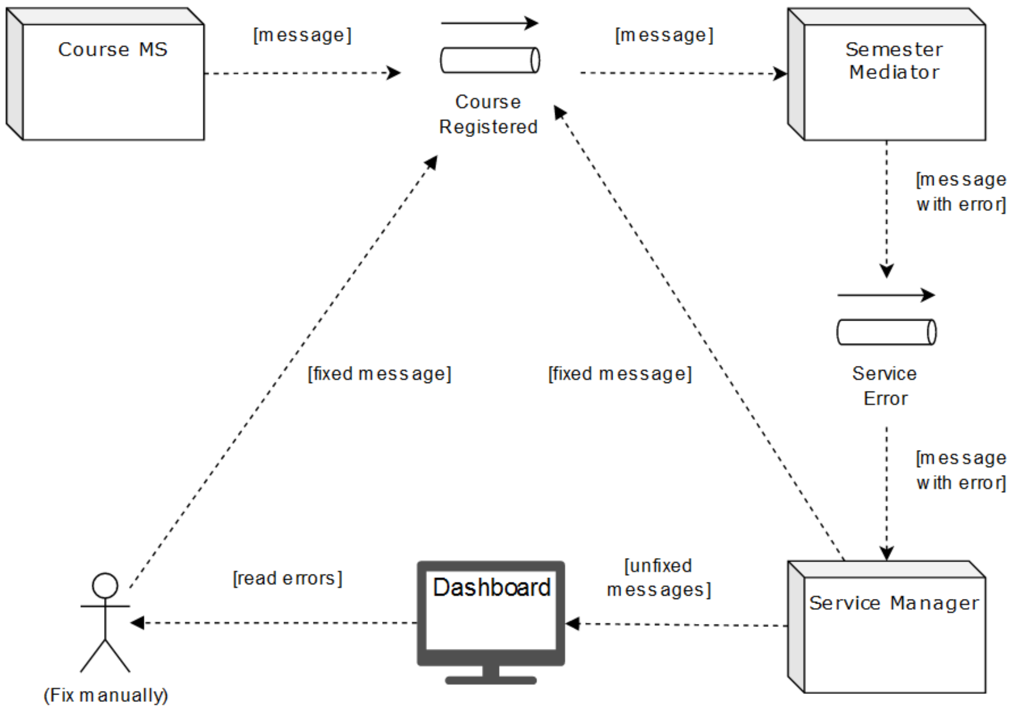


Figure 73: Error handling flow

When the *Course MS* publishes to *Course Registered* and *Semester Mediator* finds error in the message, the message will be moved to *Service Error* topic. When the *Service Manager* handles the error, there are two scenarios. One, the *Service Manager* can resolve the error automatically if the error is predicted and can programmatically be fixed. And two, when it is not, then the system operator can detect it on the dashboard and fix it manually. The following demonstration will cover each of the cases.

Resolving errors automatically

The flow is marked on this diagram below:

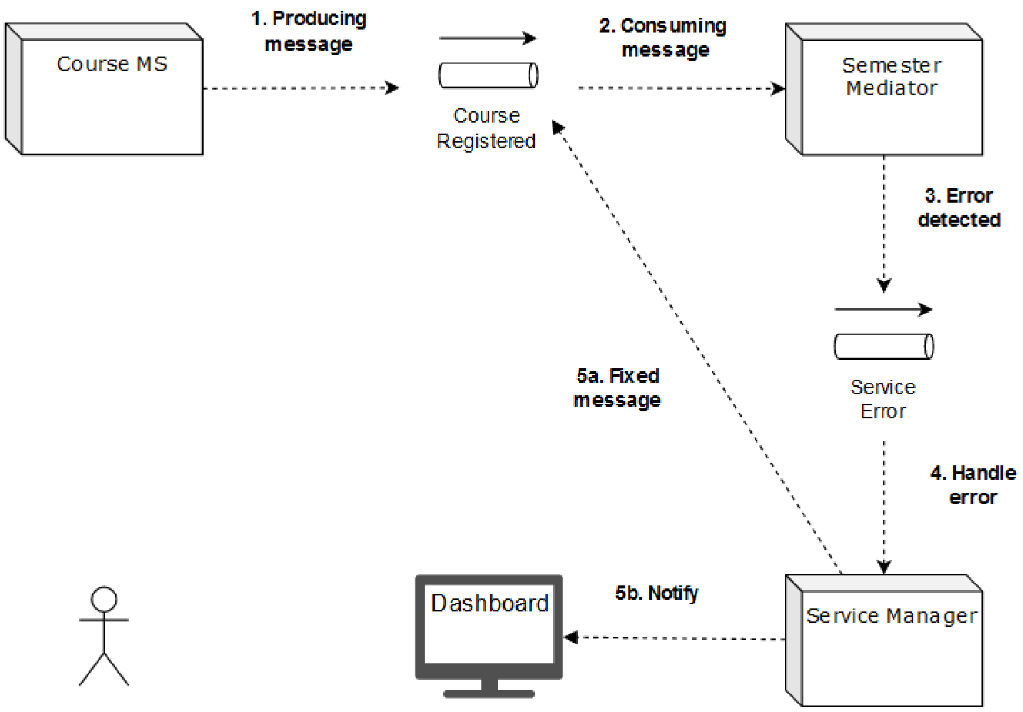
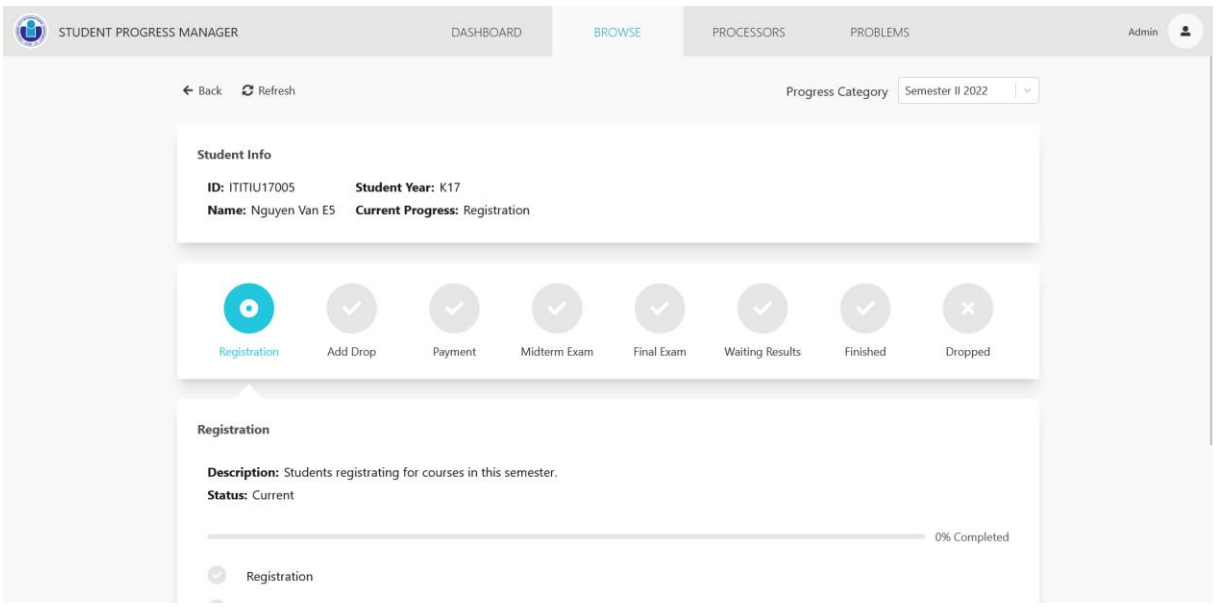


Figure 74: Handling error automatically

Starting at step one. this is a student in the registration step of semester mediator flow:



Progress Category: Semester II 2022

Student Info

ID: ITITIU17005 **Student Year:** K17
Name: Nguyen Van E5 **Current Progress:** Registration

Registration

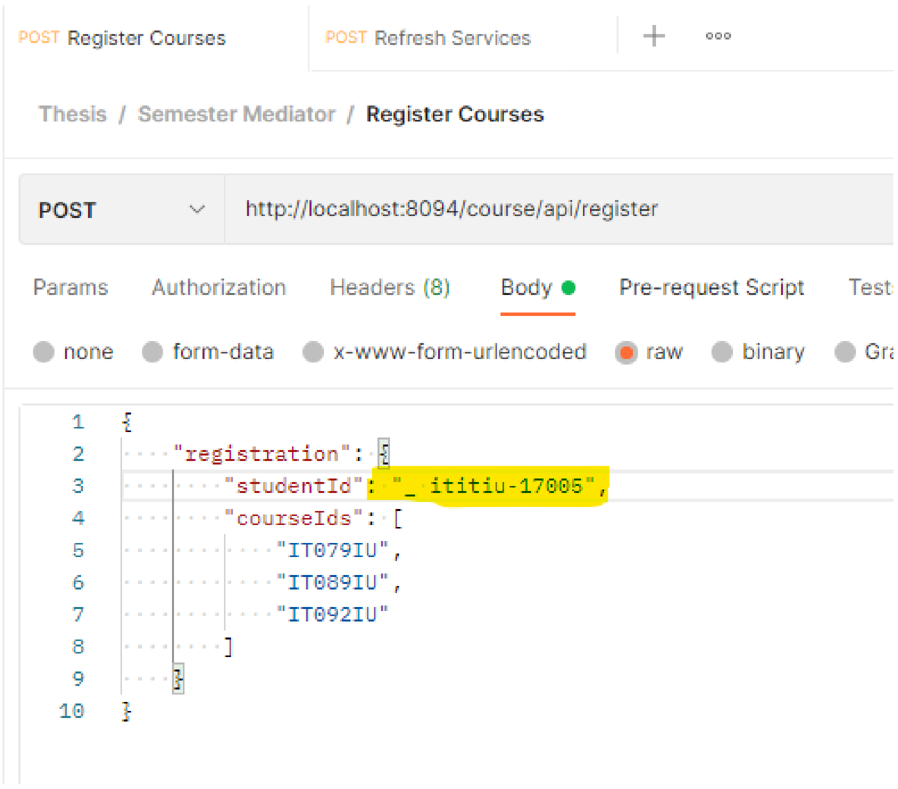
Description: Students registrating for courses in this semester.
Status: Current

0% Completed

Registration

Figure 75: Registration step

In this scenario, we try something different – instead of putting the student's id the correct format, we convert them to lowercase, add some noise characters in like dashes and spaces:



POST Register Courses | POST Refresh Services

Thesis / Semester Mediator / Register Courses

POST http://localhost:8094/course/api/register

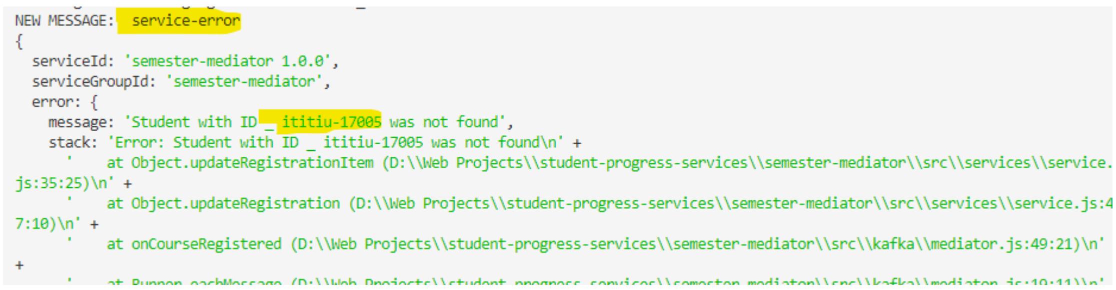
Params Authorization Headers (8) Body Pre-request Script Test

none form-data x-www-form-urlencoded raw binary Gr

```
{
    "registration": {
        "studentId": "_ ititiu-17005",
        "courseIds": [
            "IT079IU",
            "IT089IU",
            "IT092IU"
        ]
    }
}
```

Figure 76: Student ID wrong format

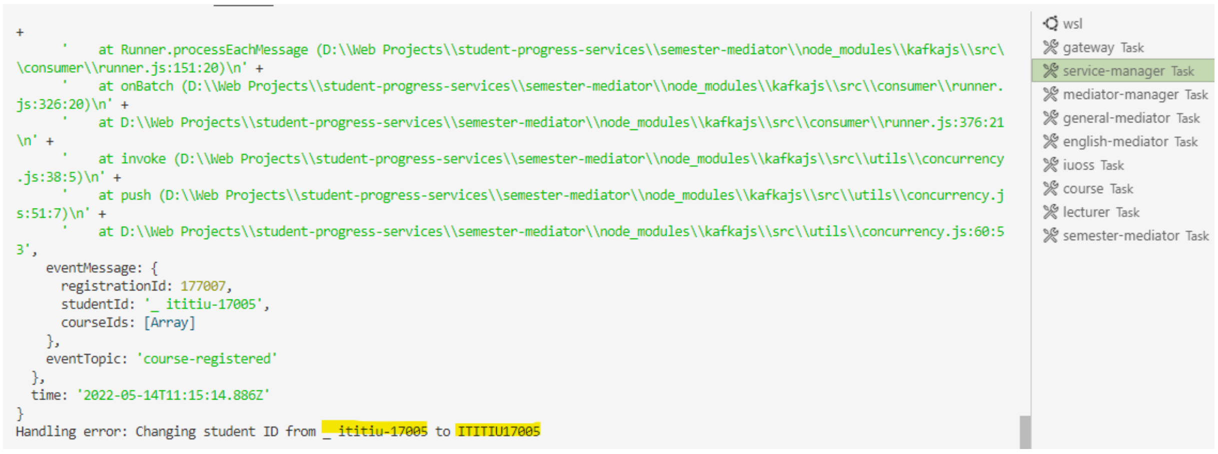
After sending the wrong formatted payload, assuming that the course service could not detect the error and publish them to the topic *course-registered* and when the semester mediator reads it, it cannot be processed and an error is thrown to the *service-error* topic. After that, the service manager is the one that handles the error:



```
NEW MESSAGE:  service-error
{
  serviceId: 'semester-mediator 1.0.0',
  serviceGroupId: 'semester-mediator',
  error: {
    message: 'Student with ID _ ititiu-17005 was not found',
    stack: 'Error: Student with ID _ ititiu-17005 was not found\n' +
      '    at Object.updateRegistrationItem (D:\\Web Projects\\student-progress-services\\semester-mediator\\src\\services\\service.
js:35:25)\n' +
      '    at Object.updateRegistration (D:\\Web Projects\\student-progress-services\\semester-mediator\\src\\services\\service.js:4
7:10)\n' +
      '    at onCourseRegistered (D:\\Web Projects\\student-progress-services\\semester-mediator\\src\\kafka\\mediator.js:49:21)\n'
+
      '    at Runner.eachMessage (D:\\Web Projects\\student-progress-services\\semester-mediator\\src\\kafka\\mediator.js:19:11)\n'
```

Figure 77: Receiving error (service manager)

And again, assuming that this error is well-known, therefore the manager knows how to fix it. In this case, it changes the student ID from `_ ititiu-17005` to `ITITIU17005`:

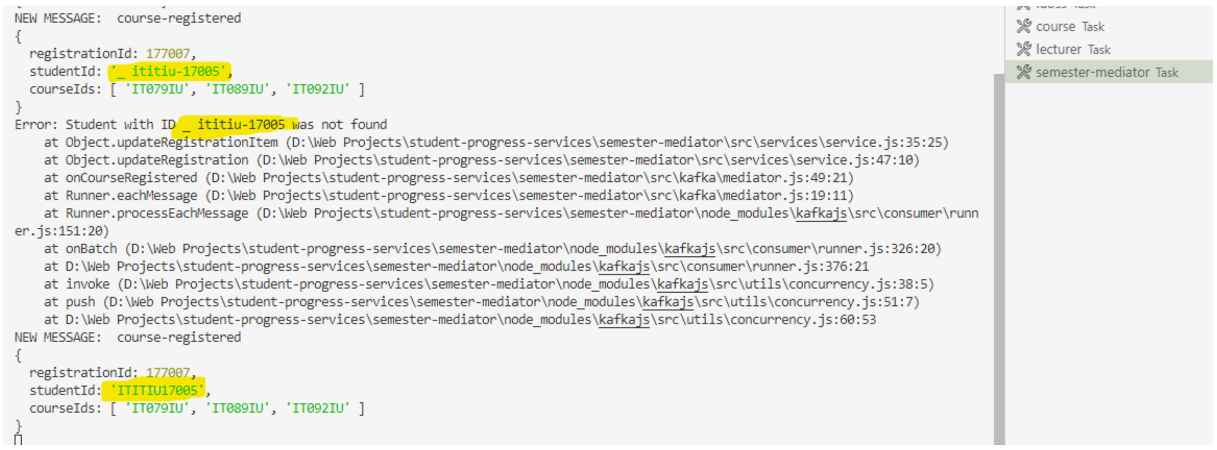


```
+
      '    at Runner.processEachMessage (D:\\Web Projects\\student-progress-services\\semester-mediator\\node_modules\\kafkajs\\src\
\consumer\\runner.js:151:20)\n' +
      '    at onBatch (D:\\Web Projects\\student-progress-services\\semester-mediator\\node_modules\\kafkajs\\src\\consumer\\runner.
js:326:20)\n' +
      '    at D:\\Web Projects\\student-progress-services\\semester-mediator\\node_modules\\kafkajs\\src\\consumer\\runner.js:376:21
\n' +
      '    at invoke (D:\\Web Projects\\student-progress-services\\semester-mediator\\node_modules\\kafkajs\\src\\utils\\concurrency
.js:38:5)\n' +
      '    at push (D:\\Web Projects\\student-progress-services\\semester-mediator\\node_modules\\kafkajs\\src\\utils\\concurrency.j
s:51:7)\n' +
      '    at D:\\Web Projects\\student-progress-services\\semester-mediator\\node_modules\\kafkajs\\src\\utils\\concurrency.js:60:5
3',
    eventMessage: {
      registrationId: 177007,
      studentId: '_ ititiu-17005',
      courseIds: [Array]
    },
    eventTopic: 'course-registered'
  },
  time: '2022-05-14T11:15:14.886Z'
}
Handling error: Changing student ID from _ ititiu-17005 to ITITIU17005
```

Figure 78: Fixing message (service manager)

This followed by a re-push the fixed message to the topic *course-registered* by the manager. And it is then consumed by the mediator and the process is finished.

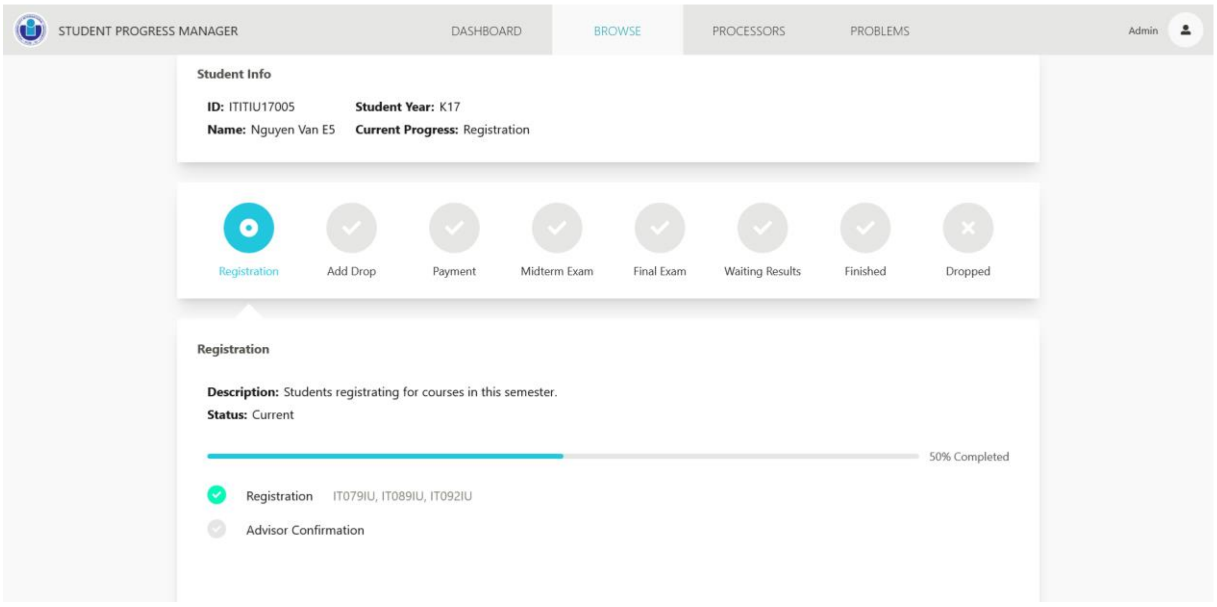
This is the logs of the semester mediator, it shows that the mediator has received two messages, the raw one and the later fixed one.



```
NEW MESSAGE:  course-registered
{
  registrationId: 177007,
  studentId: '_ ititiu-17005',
  courseIds: [ 'IT079IU', 'IT089IU', 'IT092IU' ]
}
Error: Student with ID _ ititiu-17005 was not found
    at Object.updateRegistrationItem (D:\Web Projects\student-progress-services\semester-mediator\src\services\service.js:35:25)
    at Object.updateRegistration (D:\Web Projects\student-progress-services\semester-mediator\src\services\service.js:47:10)
    at onCourseRegistered (D:\Web Projects\student-progress-services\semester-mediator\src\kafka\mediator.js:49:21)
    at Runner.eachMessage (D:\Web Projects\student-progress-services\semester-mediator\src\kafka\mediator.js:19:11)
    at Runner.processEachMessage (D:\Web Projects\student-progress-services\semester-mediator\node_modules\kafkajs\src\consumer\runn
er.js:151:20)
    at onBatch (D:\Web Projects\student-progress-services\semester-mediator\node_modules\kafkajs\src\consumer\runner.js:326:20)
    at D:\Web Projects\student-progress-services\semester-mediator\node_modules\kafkajs\src\consumer\runner.js:376:21
    at invoke (D:\Web Projects\student-progress-services\semester-mediator\node_modules\kafkajs\src\utils\concurrency.js:38:5)
    at push (D:\Web Projects\student-progress-services\semester-mediator\node_modules\kafkajs\src\utils\concurrency.js:51:7)
    at D:\Web Projects\student-progress-services\semester-mediator\node_modules\kafkajs\src\utils\concurrency.js:60:53
NEW MESSAGE:  course-registered
{
  registrationId: 177007,
  studentId: 'ITITIU17005',
  courseIds: [ 'IT079IU', 'IT089IU', 'IT092IU' ]
}
```

Figure 79: Mediator's logs

If we were to open the web, we would see the information has been updated:


STUDENT PROGRESS MANAGER | DASHBOARD | BROWSE | PROCESSORS | PROBLEMS | Admin

Student Info

ID: ITITIU17005 **Student Year:** K17
Name: Nguyen Van E5 **Current Progress:** Registration

Registration | Add Drop | Payment | Midterm Exam | Final Exam | Waiting Results | Finished | Dropped

Registration

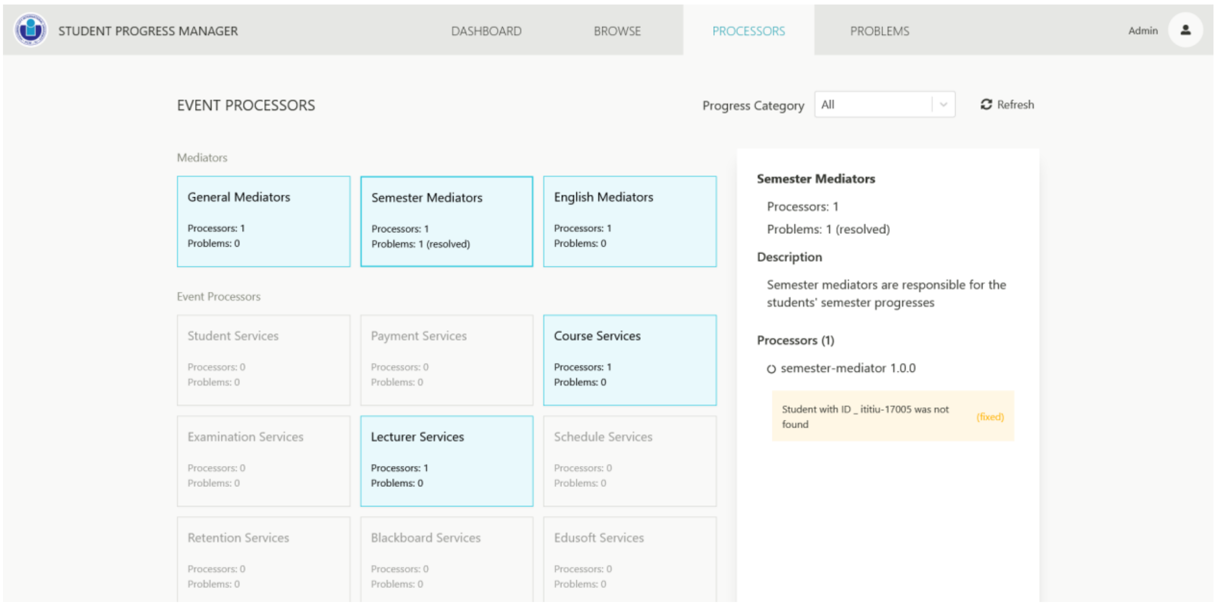
Description: Students registrating for courses in this semester.
Status: Current

50% Completed

✓ Registration IT079IU, IT089IU, IT092IU
Advisor Confirmation

Figure 80: Registration item finished

And navigating to the processors page, the error count (problems) is also updated with the error message as well as the flag that says “(fixed”):


STUDENT PROGRESS MANAGER | DASHBOARD | BROWSE | PROCESSORS | PROBLEMS | Admin

EVENT PROCESSORS

Progress Category: All | Refresh

Mediators

General Mediators	Semester Mediators	English Mediators
Processors: 1 Problems: 0	Processors: 1 Problems: 1 (resolved)	Processors: 1 Problems: 0

Event Processors

Student Services	Payment Services	Course Services
Processors: 0 Problems: 0	Processors: 0 Problems: 0	Processors: 1 Problems: 0
Examination Services	**Lecturer Services**	**Schedule Services**
Processors: 0 Problems: 0	Processors: 1 Problems: 0	Processors: 0 Problems: 0
Retention Services	**Blackboard Services**	**Edusoft Services**
Processors: 0 Problems: 0	Processors: 0 Problems: 0	Processors: 0 Problems: 0

Semester Mediators

Processors: 1
Problems: 1 (resolved)

Description

Semester mediators are responsible for the students' semester progresses

Processors (1)

semester-mediator 1.0.0

Student with ID _ ititiu-17005 was not found (fixed)

Figure 81: Error fixed display

Resolving errors manually

That is one way the service manager can handle the exception. What if the payload is not well-known and impossible to fix automatically? In this second scenaio, the error is handled manually following this diagram:

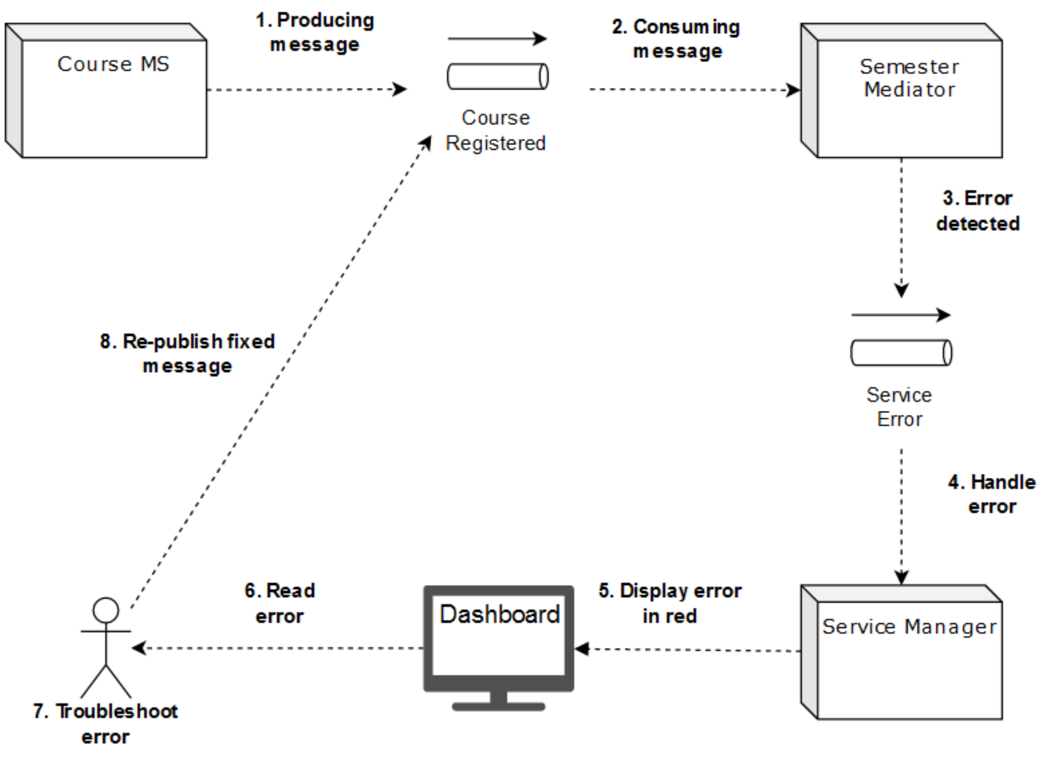
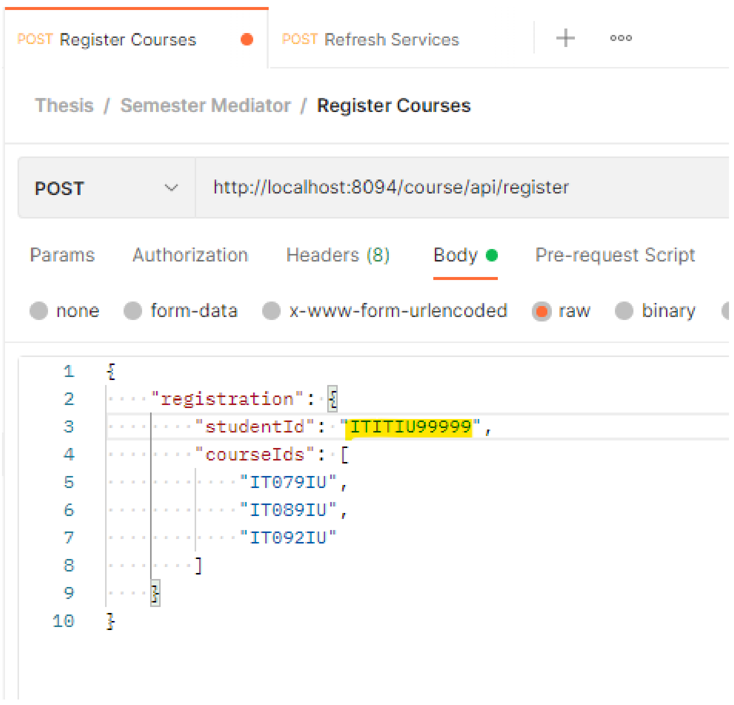


Figure 82: Handling errors manually

To demonstrate this case, we change the ID to something that does not exist in the database:

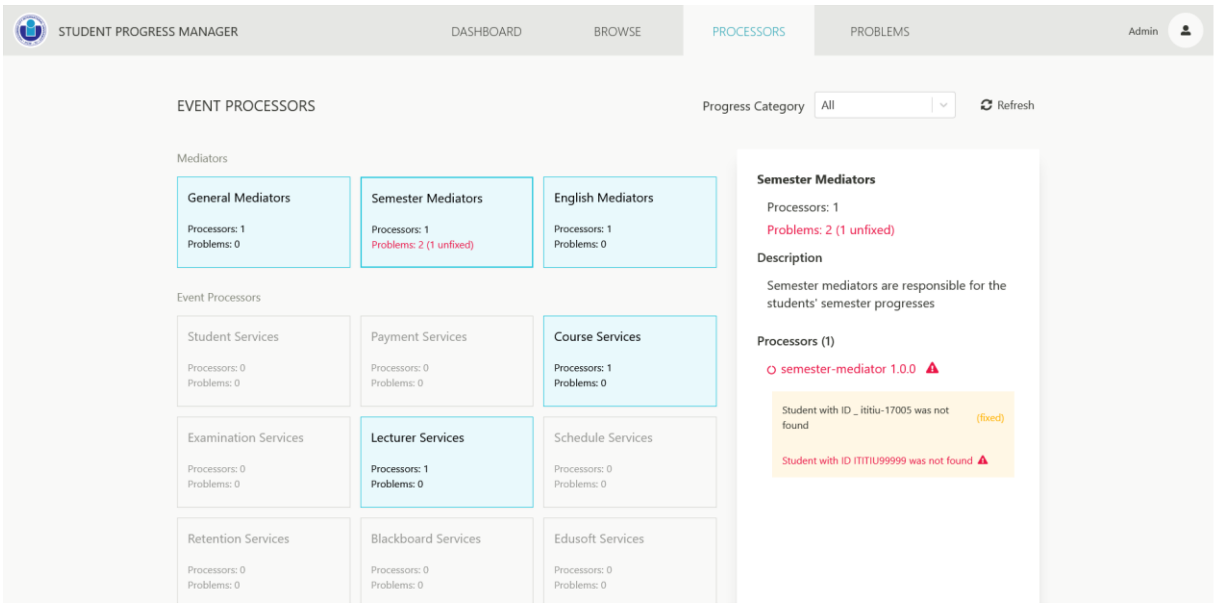


http://localhost:8094/course/api/register

```
{
    "registration": {
        "studentId": "ITITIU99999",
        "courseIds": [
            "IT079IU",
            "IT089IU",
            "IT092IU"
        ]
    }
}
```

Figure 83: Invalid student ID

And assume that the course service does not detect the error, the event is passed to the *course-registered* topic, and the semester mediator picks up the message and forwards the error to *service-error* topic. At this point, the service manager cannot fix the error by itself. The error is now highlighted red on the user interface like the below figure:



General Mediators
Processors: 1
Problems: 0

Semester Mediators
Processors: 1
Problems: 2 (1 unfixed)

English Mediators
Processors: 1
Problems: 0

Student Services
Processors: 0
Problems: 0

Payment Services
Processors: 0
Problems: 0

Course Services
Processors: 1
Problems: 0

Examination Services
Processors: 0
Problems: 0

Lecturer Services
Processors: 1
Problems: 0

Schedule Services
Processors: 0
Problems: 0

Retention Services
Processors: 0
Problems: 0

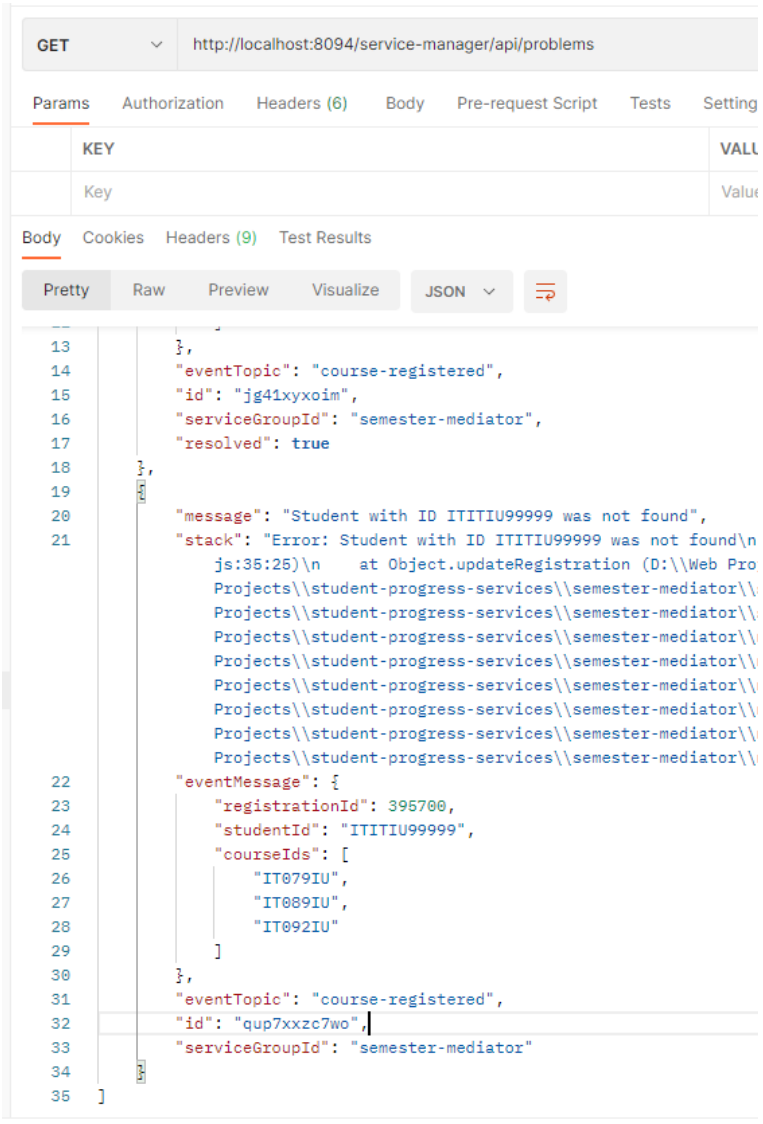
Blackboard Services
Processors: 0
Problems: 0

Edusoft Services
Processors: 0
Problems: 0

Semester Mediators
Processors: 1
Problems: 2 (1 unfixed)
Description
Semester mediators are responsible for the students' semester progresses
Processors (1)
semester-mediator 1.0.0
Student with ID _ ititiu-17005 was not found (fixed)
Student with ID ITITIU99999 was not found

Figure 84: Error displayed red

Now, when the system operator is notified about this error, the maintainance team can investigate the problem and resolve it. They can get all errors by using the get request:

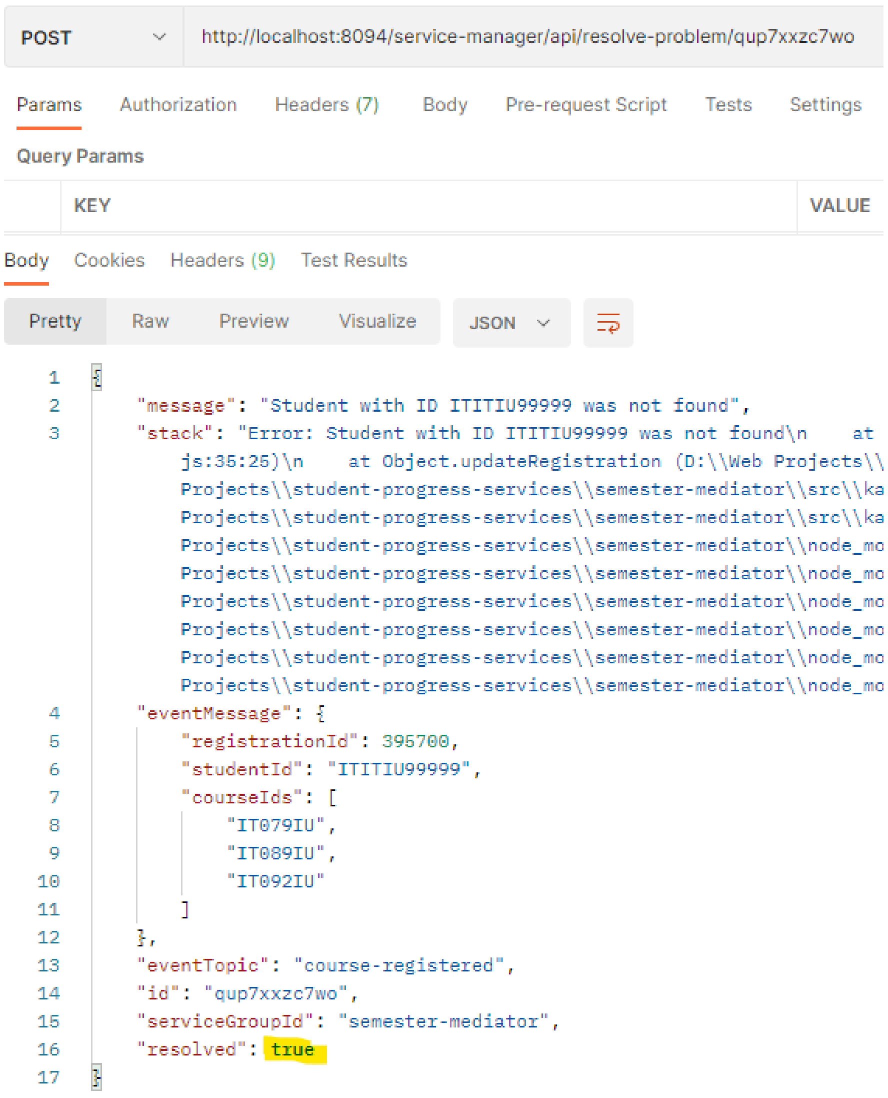


GET http://localhost:8094/service-manager/api/problems

```
        },
        "eventTopic": "course-registered",
        "id": "jg41xyxoim",
        "serviceGroupId": "semester-mediator",
        "resolved": true
    },
    {
        "message": "Student with ID ITITIU99999 was not found",
        "stack": "Error: Student with ID ITITIU99999 was not found\n
            js:35:25)\n    at Object.updateRegistration (D:\\Web Pro
            Projects\\student-progress-services\\semester-mediator\\
            Projects\\student-progress-services\\semester-mediator\\
            Projects\\student-progress-services\\semester-mediator\\
            Projects\\student-progress-services\\semester-mediator\\
            Projects\\student-progress-services\\semester-mediator\\
            Projects\\student-progress-services\\semester-mediator\\
            Projects\\student-progress-services\\semester-mediator\\
            Projects\\student-progress-services\\semester-mediator\\
        "eventMessage": {
            "registrationId": 395700,
            "studentId": "ITITIU99999",
            "courseIds": [
                "IT079IU",
                "IT089IU",
                "IT092IU"
            ]
        },
        "eventTopic": "course-registered",
        "id": "qup7xxzc7wo",
        "serviceGroupId": "semester-mediator"
    }
]
```

Figure 85: Getting errors

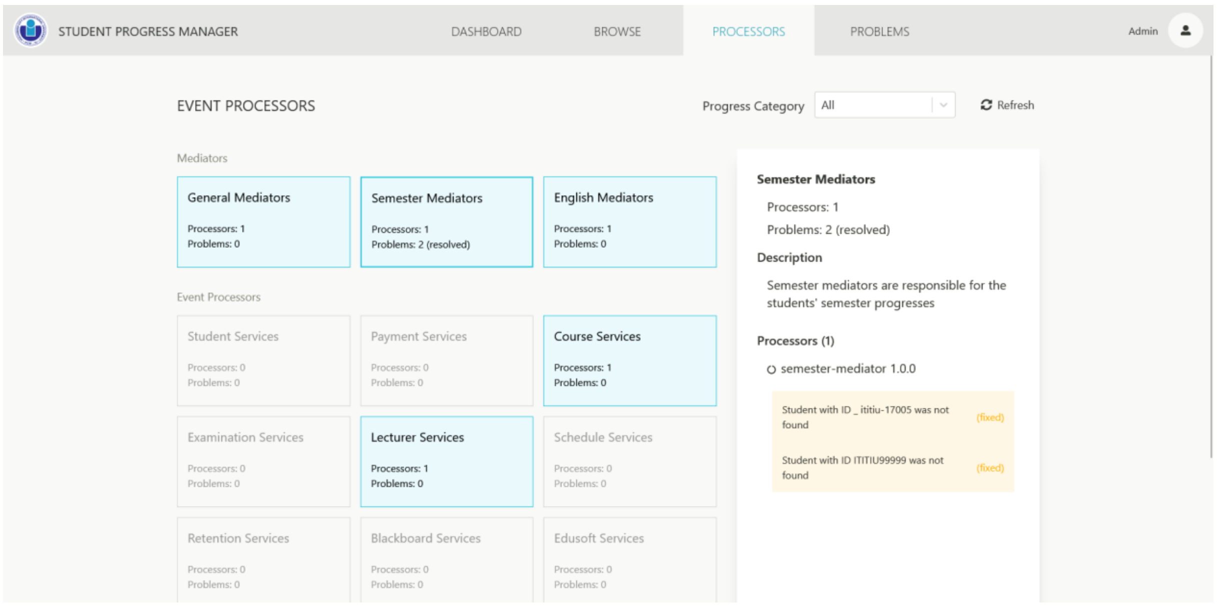
After that, to tell the system that the error has been handled, we use the post request to mark the error with specific ID as fixed:



```
{
    "message": "Student with ID ITITIU99999 was not found",
    "stack": "Error: Student with ID ITITIU99999 was not found\n    at
        js:35:25)\n    at Object.updateRegistration (D:\\Web Projects\\
        Projects\\student-progress-services\\semester-mediator\\src\\ka
        Projects\\student-progress-services\\semester-mediator\\src\\ka
        Projects\\student-progress-services\\semester-mediator\\node_mo
        Projects\\student-progress-services\\semester-mediator\\node_mo
        Projects\\student-progress-services\\semester-mediator\\node_mo
        Projects\\student-progress-services\\semester-mediator\\node_mo
        Projects\\student-progress-services\\semester-mediator\\node_mo
        Projects\\student-progress-services\\semester-mediator\\node_mo
    "eventMessage": {
        "registrationId": 395700,
        "studentId": "ITITIU99999",
        "courseIds": [
            "IT079IU",
            "IT089IU",
            "IT092IU"
        ]
    },
    "eventTopic": "course-registered",
    "id": "qup7xxzc7wo",
    "serviceGroupId": "semester-mediator",
    "resolved": true
}
```

Figure 86: Mark error as fixed

And finally, the user interface can now display them normally without the red warning indications:



EVENT PROCESSORS

Progress Category: All

Mediators

General Mediators
Processors: 1
Problems: 0

Semester Mediators
Processors: 1
Problems: 2 (resolved)

English Mediators
Processors: 1
Problems: 0

Event Processors

Student Services
Processors: 0
Problems: 0

Payment Services
Processors: 0
Problems: 0

Course Services
Processors: 1
Problems: 0

Examination Services
Processors: 0
Problems: 0

Lecturer Services
Processors: 1
Problems: 0

Schedule Services
Processors: 0
Problems: 0

Retention Services
Processors: 0
Problems: 0

Blackboard Services
Processors: 0
Problems: 0

Edusoft Services
Processors: 0
Problems: 0

Semester Mediators
Processors: 1
Problems: 2 (resolved)

Description
Semester mediators are responsible for the students' semester progresses

Processors (1)
semester-mediator 1.0.0

Student with ID _ ititiu-17005 was not found (fixed)

Student with ID ITITIU99999 was not found (fixed)

Figure 87: Error is now fixed

This is the end of the error handling demonstration.

CHAPTER 5

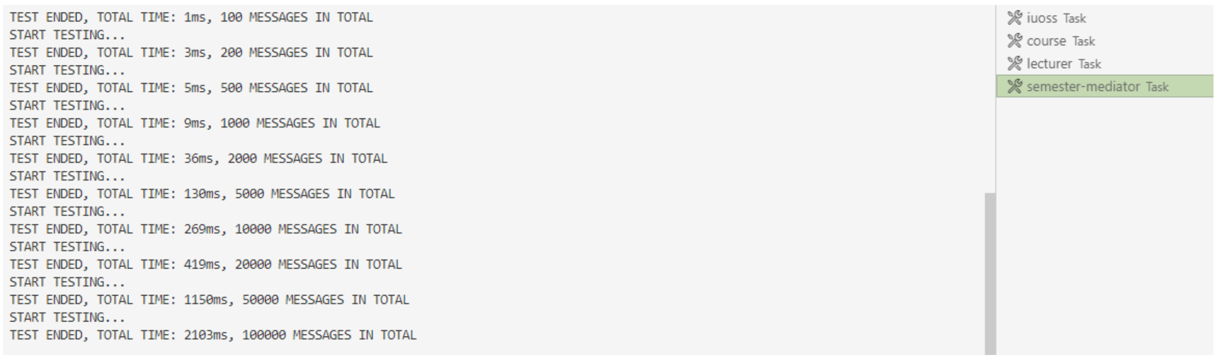
EVALUATION AND DISCUSSION

5.1. Evaluation

This evaluation section will test two properties of using event-driven as well as the use of mediators inside the system. The first test is throughput test, means how much time it takes to handle a large number of requests. The second is latency test, comparing the delay time between using and not using mediators in a particular communication between two services.

Throughput test

To know if the process speed is efficient, a simulation is run with from 100 to 100,000 requests of course registrations to test the time it takes to receive all of the messages. The respective event consumer is the semester mediator and the times between receiving the first to the last messages are measured. This experiment does not include the side processing time that mediator do its jobs including updating data and publishing to other topics. This is the result:



```
TEST ENDED, TOTAL TIME: 1ms, 100 MESSAGES IN TOTAL
START TESTING...
TEST ENDED, TOTAL TIME: 3ms, 200 MESSAGES IN TOTAL
START TESTING...
TEST ENDED, TOTAL TIME: 5ms, 500 MESSAGES IN TOTAL
START TESTING...
TEST ENDED, TOTAL TIME: 9ms, 1000 MESSAGES IN TOTAL
START TESTING...
TEST ENDED, TOTAL TIME: 36ms, 2000 MESSAGES IN TOTAL
START TESTING...
TEST ENDED, TOTAL TIME: 130ms, 5000 MESSAGES IN TOTAL
START TESTING...
TEST ENDED, TOTAL TIME: 269ms, 10000 MESSAGES IN TOTAL
START TESTING...
TEST ENDED, TOTAL TIME: 419ms, 20000 MESSAGES IN TOTAL
START TESTING...
TEST ENDED, TOTAL TIME: 1150ms, 50000 MESSAGES IN TOTAL
START TESTING...
TEST ENDED, TOTAL TIME: 2103ms, 100000 MESSAGES IN TOTAL
```

Figure 88: Testing performance

Number of messages	Time (ms)	
100	1	100
200	3	66
500	5	100
1,000	9	111
2,000	36	55
5,000	130	38
10,000	269	37.2
20,000	419	47.7
50,000	1150	43
100,000	2103	47

Table 8: Time for different number of messages

Overall, the time for all of them is very fast, as for 100,000 messages, it is around two seconds. And predictably, the pattern of the data shows there is a linear correlation between the time and the message count.

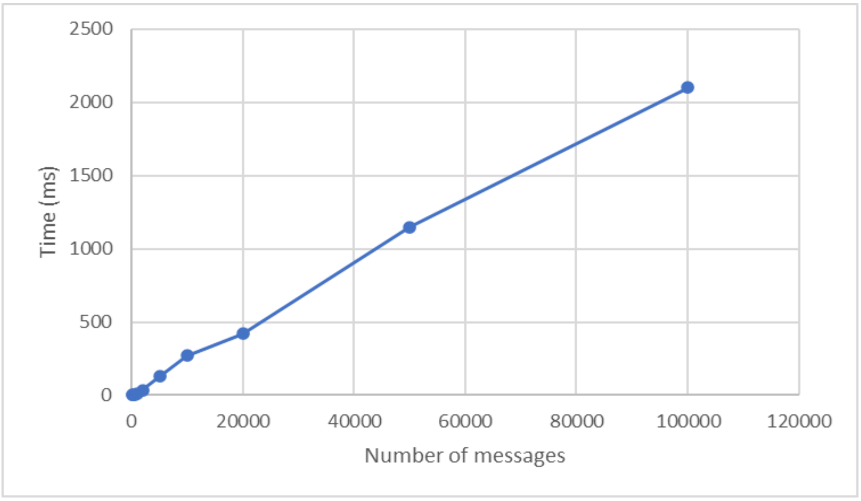


Figure 89: Time for different number of messages

Latency test

In this test, we want to find out the time difference between communication with mediator and without mediator. The time being measured is how long it takes since a course registration is done by the course service to when the lecturer service receives the event.

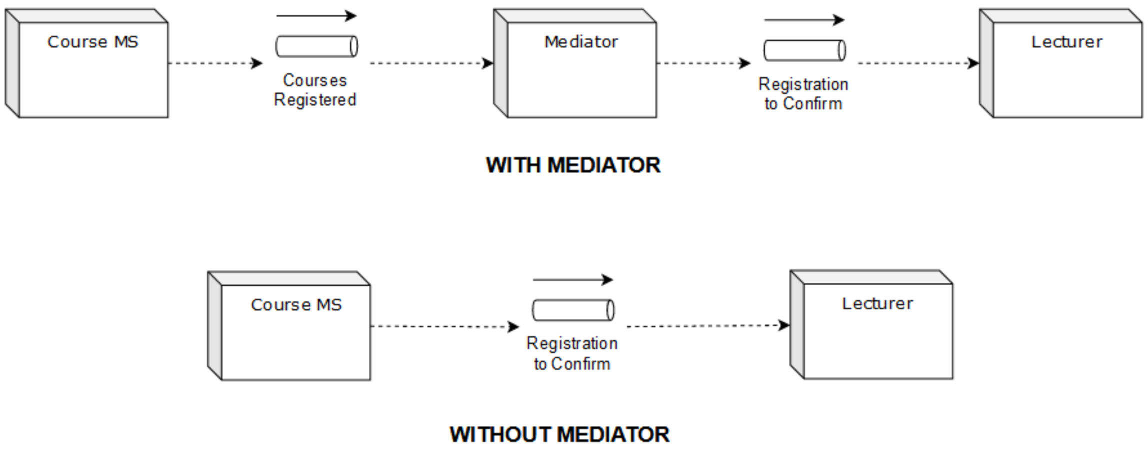


Figure 90: Latency test with and without mediator

For the case that the mediator is present, the message first goes to *Courses Registered* topic, then through the mediator and the second topic *Registration to Confirm*, and then finally to the lecturer service. In the case without mediator, there is only one queue that the message travel through that is *Registration to Confirm* and the lecturer service consumes it directly.

This is the result of the test after five tries for both cases:

	Mediator	**Direct**

#1	5 ms	2 ms
#2	4 ms	0 ms
#3	4 ms	1 ms
#4	5 ms	1 ms
#5	7 ms	0 ms
#6	8 ms	4 ms
#7	7 ms	2 ms
#8	8 ms	1 ms
#9	9 ms	1 ms
#10	9 ms	2 ms
Average	*6.6 ms*	*1.4 ms*

Table 9: Latency test between mediator and direct

The mediator's measurements are longer than that of the direct communication by a number of milliseconds (5.2 ms). The reason why mediator's time is not just a double of the other but far higher might be the result of the time it takes for the mediator to wait for the connection of its producer to Kafka. Therefore, although mediators are really great at the orchestration level, they are not optimal at run time and should be used with consideration when applying this pattern in practice.

5.2. Discussion

This thesis has many improvements on the event-driven way of communicating between microservices which is Hoang Tu's thesis. He has dealt with the asynchronous way of processing requests by introducting Kafka as an event broker, that has made the system more responsive and powerful. In contrast to that, this work by me has put another layer of abstraction on top – the mediator pattern, services managing, error handling mechanism – and apply them to managing student learning progress.

The first advantage of this work is to separate the services from their unrelated topics, which is decoupling. Second, the overall flow of the services is easier to setup and change especially the scenarios that have multiple steps. Another problem arises when scaling the system up regarding the different services to be look after and when errors occur, that was also solved by the service manager and the error handling pattern in event-driven. Finally, a user interface has been created to bring all of the abstract logic to the front-end for better visualization.

However, as tested in the evaluation section, this implementation of mediator can be a drawback for a low-latency required system as the system scales, small difference in milliseconds can turn to something more dramatic. Therefore, it is not ideal to apply this pattern in any situation.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1. Conclusion

From the methodology, I have implemented and demonstrated the mediator pattern working in the student learning progress management system as well as the ways of managing service with an error handling mechanism and they are all displayed on the user interface. Together, this thesis has shown the next step of system integration in event-driven architecture. It is important and very useful for a large system with abundance of events we wish to manage, specifically, the scenarios with numbers of steps that involve multiple event processors. Applying this to the current project, we can now keep track of the students' progresses in any facet of the university and have the potential to extend the system further by integrating new services or by connecting them to legacy systems.

6.2. Future work

The mediator implementation in this thesis programmed in the mediator in codes. When more and more mediators are requested, it is a good time to research and leverage Apache ODE[7] and Oracle BPEL[8], the existing frameworks to manage flow inside the system by interacting on the GUI.

Another space to further look into is to have more several topics from microservices listened by a single central event manager. From that we can create an all-in-one app, that is, every user-concerned event is gathered into one single app for student and lecturer when the course is registered, examination score update, student request responded or class cancelled, the users can see notification and the history of them.

REFERENCES

[1] "An Introduction to Event Driven Microservices," *Developer.Com*, Nov. 25, 2021. https://www.developer.com/design/event-driven-microservices/

[2] "Benefits of Migrating to Event-Driven Architecture," *Amazon Web Services*, May 09, 2022. https://aws.amazon.com/blogs/compute/benefits-of-migrating-to-event-driven-architecture/

[3] Neal Ford, Mark Richards, *Fundamentals of Software Architecture*. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc., 2020.

[4] Nguyen Phan Hoang Tu, "System Integration for Student Academic Affairs Management with Event-driven Architecture," VNU HCM International University, 2021.

[5] "Introduction to Apache Kafka," *Apache Kafka*. https://kafka.apache.org/intro

[6] Dreamix Group, "Building Microservices with Netflix OSS, Apache Kafka and Spring Boot - Part 2: Message Broker and User Service," 7 Nov. 20217. https://dreamix.eu/blog/java/building-microservices-with-netflix-oss-apache-kafka-and-spring-boot-part-2

[7] "An Introduction to Apache ODE." https://www.infoq.com/articles/paul-brown-ode/

[8] "Introduction to Oracle BPEL Process Manager," *Oracle*. https://docs.oracle.com/cd/B14099_19/integrate.1012/b14448/introbpel.htm