Documentation: Middleware-Based Student Data Integration

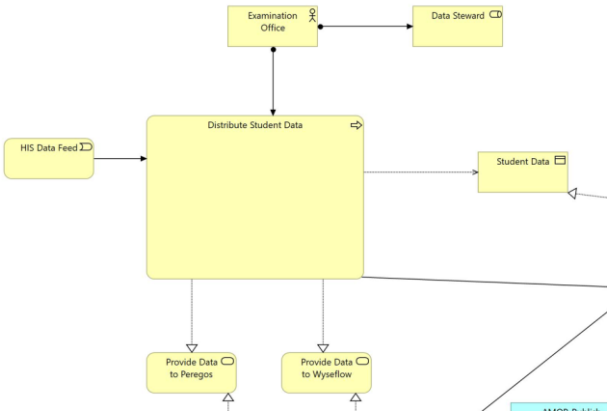
1. Introduction

As part of the *Architecture and Integration* course, a middleware was implemented to automatically distribute student data from the HIS system to two target systems: **Peregos** (examination board) and **WyseFlow** (final thesis management). The goal was to avoid manual duplication of work and improve data consistency. The solution is based on a **message-based integration pattern** using **RabbitMQ**.

2. Architectural Overview

The architecture of the solution was modeled using ArchiMate and describes the key components involved in the automated student data distribution process. It includes the business, application, and technology layers as required by the project. The architecture ensures clear separation of concerns and supports reliable and scalable integration between HIS, Peregos, and WyseFlow.

2.1 Business Layer



At the business level, the process of distributing student data is initiated by the **Examination Office** and handled by the **Data Steward**. The process is modeled as a business process called **"Distribute Student Data"**, which receives input from the **HIS Data Feed** and outputs data to both **Peregos** and **WyseFlow**.

This layer also includes two business services:

- **Provide Data to Peregos**
- **Provide Data to WyseFlow**

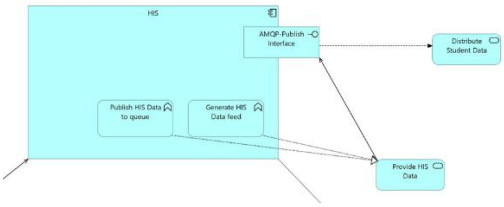
These services are realized by application functions in the application layer and support the overall data delivery process.

2.2 Application Layer

The application layer models the behavior and interaction of the systems involved in the integration.

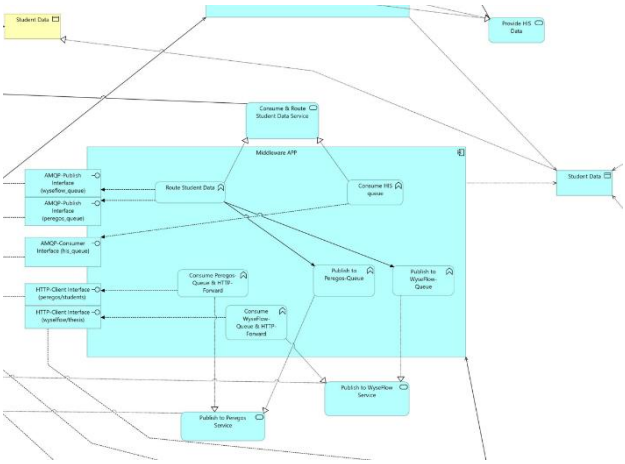
Applications:

HIS is the source system for student data. It contains application functions such as:



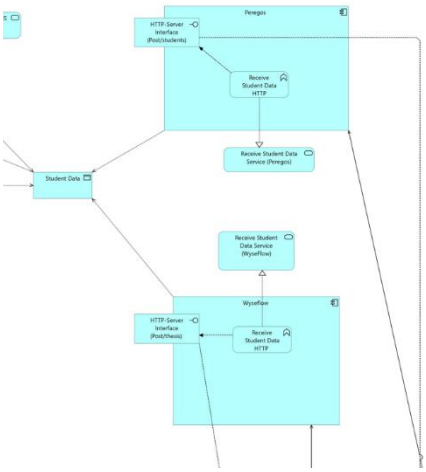
- ❖ Generate HIS Data Feed
- ❖ Publish HIS Data to Queue
- ❖ These functions are exposed via an **AMQP Publish Interface**, which connects to the middleware.

Middleware Application handles all routing and processing logic. Its key functions include:



- ❖ Consume HIS Queue
- ❖ Route Student Data
- ❖ Publish to Peregos Queue and Publish to WyseFlow Queue
- ❖ Consumer Peregos Queue & HTTP Forward
- ❖ Consumer WyseFlow Queue & HTTP Forward

Peregos and **WyseFlow** are target systems. Each contains:



- ❖Receive Student Data Service
- ❖Exposed through **HTTP Client Interfaces** such as /students and /thesis.

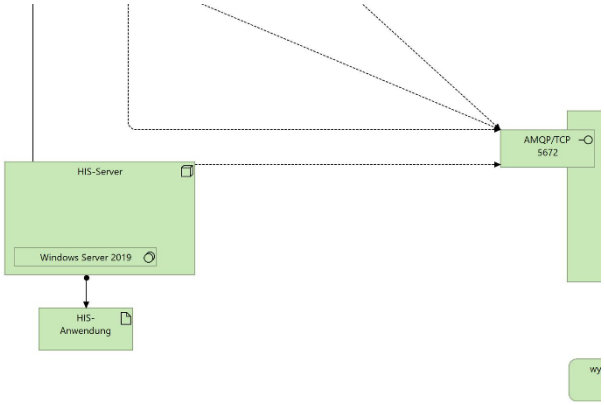
All these systems are connected through **data objects** like Student Data, which flows from HIS through the middleware to the target systems.

2.3 Technology Layer

The technology layer shows the physical infrastructure and execution environments that support the application layer.

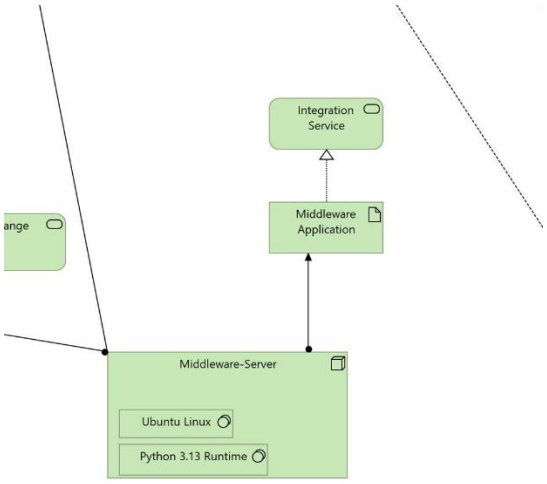
Servers and Nodes:

HIS Server



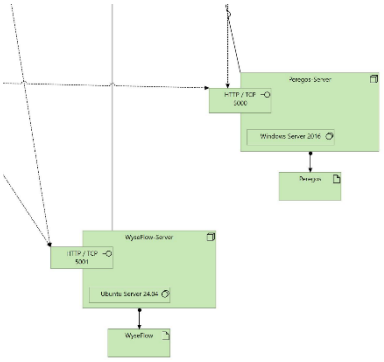
- Runs the HIS application on a **Windows Server 2019** node.
- Generates student data records in JSON format.
- Publishes messages to RabbitMQ using the routing key ***studentdata.his***

Middleware Server



- Hosts the Python-based middleware application on **Ubuntu Linux** with **Python 3.11 Runtime**.
- Consumes messages from ***his_queue***
- Routes data to ***peregos_queue*** and ***wyseflow_queue*** using defined routing keys.

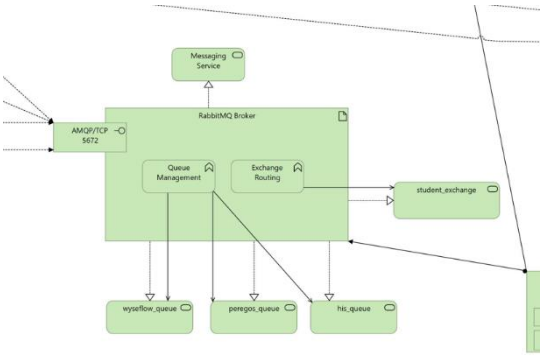
Peregos Server and **WyseFlow Server**



- Simulate the target systems, each running on separate **Ubuntu** or **Windows** environments and exposing HTTP interfaces (/students and /thesis).

Integration Infrastructure:

RabbitMQ Broker Acts as the messaging middleware, deployed as a separate node. It includes:

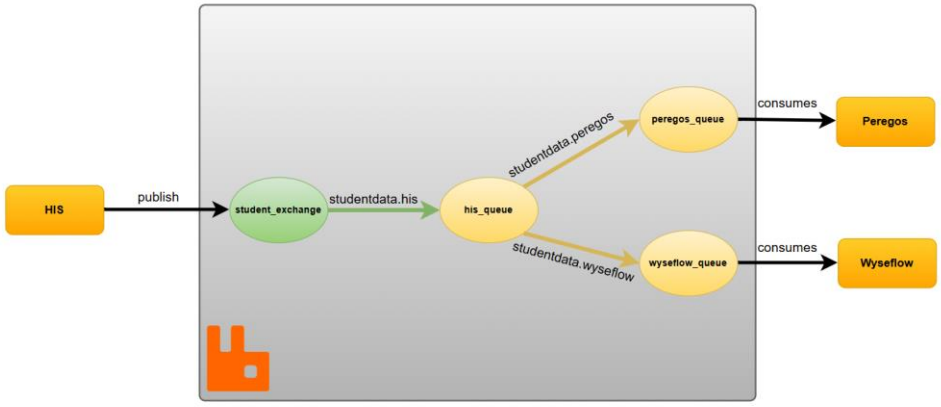


- ❖ **Exchange Routing (student_exchange)**
- ❖ **Queue Management** with:
- ❖ his_queue
- ❖ peregos_queue
- ❖ wyseflow_queue

RabbitMQ communicates over **AMQP/TCP 5672**, and each consumer connects via its respective queue interface. This infrastructure enables asynchronous and decoupled communication between the systems.

3. How the Middleware Works

The following diagram illustrates the overall message flow architecture implemented in the middleware



The middleware follows a message-based data flow using **RabbitMQ** as the central message broker. The workflow is organized in three main steps:

3.1 HIS sends student data

The script **run_his.py** simulates the HIS system and sends sample student records to RabbitMQ using the routing key for the **his_queue**. This queue acts as a **buffer**, decoupling the data generation from the routing logic.

publish_message(QUEUE_HIS, student)

The message is not sent directly to Peregos or WyseFlow because it must first pass through the middleware's routing logic. This enables **central control**, **future filtering**, and **format conversion** before sending the data to the target systems.

3.2 Middleware routes the data

The **his_consumer.py** listens to the his_queue and passes each student message to the **route_student()** function, which sends copies to both **peregos_queue** and **wyseflow_queue** using routing keys.

publish_message(ROUTING_KEY_PEREGOS, data)

publish_message(ROUTING_KEY_WYSEFLOW, data)

Messages are sent to **peregos_queue** and **wyseflow_queue**

3.3 Peregos & WyseFlow receive the data

Each target system has its own consumer script, **peregos_consumer.py** and **wyseflow_consumer.py**

These consumers read from their respective queues and first call /health to check whether the target system is online. Only then do they send a POST request to the corresponding endpoint.

3.4 Error handling and reliability

If a system is **offline** or returns an **HTTP error**, the message is not lost. Instead, a **warning** is logged, the message is **requeued**, the consumer **waits** before retrying

4. Conclusion

The implemented middleware shows how student data can be shared reliably and efficiently between different systems using a message-based approach. By using RabbitMQ, the systems are loosely connected, making the data flow more stable and easier to manage.

The structure of the solution is flexible, so it can be extended in the future — for example, by adding new systems, checking data before sending, or adding monitoring.