ONLINE EXAMINATION SYSTEM

(Event-Driven Microservices)

Software Design Project

CS5721

Lecturer: Mr. J.J Collins

Submitted By:

Wang Chonghuan – 22117989

Rohini Govindarajan Konar - 22020616

Deepika Pandian - 22163514

CONTENT (with index)

1. Project description

1.1 Overview

Online Examination System is a web-based system where online exams can be conducted for students by the teachers. It was during pandemic when there was a huge demand for such systems and a lot of the universities utilized such web- based application to conduct the examinations. This system provides an easy mode of examination at any point in time and at any place possible. The system will allow the students to select a department and then register for a course. Once they sign in, they will have access to the application and will have to register for a course. After registration is confirmed, the student will be able to sit for the exam.

The system will have a teacher module wherein a teacher will be able to accept the registration of a student for a course. The teacher can set the exam questions which can be of various types i.e., Multiple choice, Single Choice, Short Answer and Long Answer. The teacher will have access to the question bank which will have a set of questions which might be huge in number. The teacher can choose the questions from the question bank and accordingly set the question paper for his subject. At any point in time the teacher can add, modify or delete a question from the question bank. The Teacher will be able to view the results of the students in different formats, they will have an option to choose.

The system will have an admin module wherein the main activities that an admin can perform would be adding or removing any teacher along with this he will have the authority to provide a teacher access to the subjects. He can add or remove the course on which the examination can be taken. He will have the same access as a teacher to add, modify or remove questions from the question bank.

1.2 Case tools

IntelliJ IDEA is an integrated development environment written in Java for developing computer software written in Java, Kotlin, Groovy, and other JVM-based languages. It is developed by JetBrains and is available as an Apache 2 Licensed community edition, and in a proprietary commercial edition.

DataGrip is a database management environment for developers. It is designed to query, create, and manage databases. Databases can work locally on a server, or in the cloud. Supports MySQL, PostgreSQL, Microsoft SQL Server, Oracle, and more.

StarUML is a software engineering tool for system modeling using the Unified Modeling Language, as well as Systems Modeling Language, and classical modeling notations.
Diagrams.net diagrams.net is a free and open-source cross-platform graph drawing software developed in HTML5 and JavaScript.

GitHub is an Internet hosting service for software development and version control using Git. It provides the distributed version control of Git plus access control, bug tracking, software feature requests, task management, continuous JIRA integration, and wikis for every project.

SonarCloud is a cloud-based code analysis service designed to detect code quality issues in 25 different programming languages, continuously ensuring the maintainability, reliability and security of your code.

2. Software development process

Software Development Lifecycle is the application of standard business practices to building software applications. It is basically divided into different steps namely: Planning, Analysis, Design, Implementation, Test &Integration and Maintenance. There are a lot of different software development models available like Waterfall Model, V-Shaped Model, Prototype Model, Spiral Model, Iterative Incremental Model, Big Bang Model, Agile Model.

Agile methodology is an iterative process which allows us to revisit any stage if we think we need to make a change or in case any improvement is needed. At the end of every iteration feedback will be provided by the customer and user. Based on the various discussions with the customer and the users, the feedback can be incorporated in the next iteration. It also allows us to communicate on a regular basis, which provides us with a clear picture of the overall system that we need to build and there will not be any communication gaps. The overall development time of the application is cut down when agile model is used. For this project we did a lot of brainstorming on the model we should go ahead with and given the system we are going to build we have chosen Agile Model because of flexibility, easily handling sudden changes anytime, Facilitates effective communication between teams and clients and cuts overall development time.

3. Project plan

Sr. no	Deliverable/Tasks	Owner	Target completion week
1	Front Cover	Deepika	2
2	Business Scenario	Rohini	2
3	Lifecycle adopted	Rohini	2
4	Project Plan and Roles	Rohini	3
5	Requirements and Use Cases	Rohini	3
6	Use Case Diagram	Deepika	4
7	Use Case Description	Rohini	4
8	Quality Attributes	Rohini	4
9	GUI Prototypes	Deepika	5
10	MVC Architectural Patterns	Wang	5
11	System Architecture	Wang	5
12	List of candidate objects	Wang	6
13	Class Diagram	Wang	6
14	Sequence Diagram	Deepika	6
15	Entity Relationship Diagram	Wang	6
16	State chart	Wang	6
17	Transparency and Traceability	Rohini	7
18	Code snippets	Wang	8
19	Design Pattern	Team	8
20	Automation Testing	Deepika	9
21	Version control snippets	Rohini	9
22	Recovered architecture	Wang	9
23	Design Blueprints	Wang	10
24	Critique of quality of the design and implementation	Wang	10
25	Reflection	Team	11
26	References	Deepika	10

3.1 Roles and responsibility of each team member

Sr. No	Role	Description	Designated Team Member
1	Project Manager	Sets up group meetings, gets agreement on the project plan, and tracks progress.	Rohini
2	Documentation Manager	Responsible for sourcing relevant supporting documentation from each team member and composing it in the report.	Deepika

3	Business Analyst / Reqs. Engineer	Responsible for section 6 - Requirements.	Deepika
4	Architect	Defines system architecture	Wang
5	Systems Analysts	Creates conceptual class mode	Rohini
6	Designer	Responsible for recovering design time blueprints from implementation	Wang
7	Technical Lead	Leads the implementation effort	Wang
8	Programmers	Each team member develops at least 1 package in the architecture	All
9	Tester	Coding of automated test cases	Deepika
10	Devops	Must ensure that each team member is competent with development infrastructure, e.g., GitHub, etc.	Rohini

3.2 Industry experience

Student ID	Name	Experience	Domain/Programming Language/ Framework
22117989	Wang Chonghuan	6 years	C++, PHP and Python
22020616	Rohini Govindarajan Konar	7 years	Data Testing and Automation Testing
22163514	Deepika Pandian	5 years	UX & UI designing (HTML, Figma)

4. Requirements

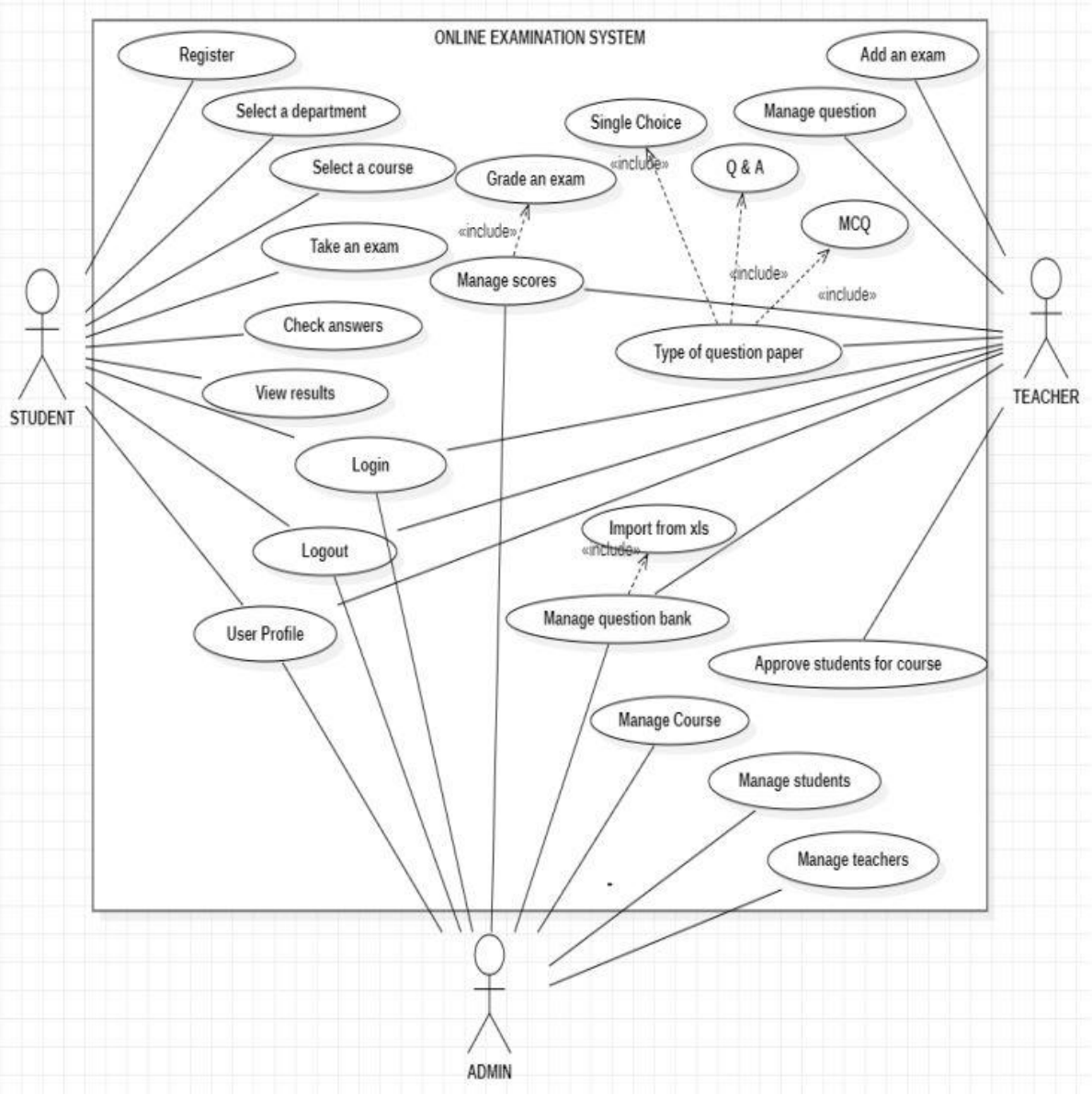
4.1 Functional requirements

For Student:
1.A student can register using his email address, phone number, address, first name and last name.
2.Students can select a department under which the courses would be available.
3.Students need to select a course and register for it.
4.Students can take exams for different courses they have registered for.
5.Students can check the results on the application and download a copy of the results.
6.Students can check for past exam results.
7.Students can check the average score of all the courses they have opted for.

For Teacher:
1.Teachers can approve the registration of a student for a course.
2.Teachers can add an exam for a course.
3.Teachers can manage the question for the exam – add, modify or delete a question.
4.Teachers can manage the question bank – add, modify or delete a question from question bank.
5.Teachers can grade the student.
6.Teacher can check the average score of all the students in a course.

For Admin:
1.Admin can manage Teachers i.e., can perform crud operations.
2.Admin can manage Students.
3.Admin can manage the question bank.
4.Admin can manage the scores of the student.

4.1.1 Use case diagram



4.1.2 Use case descriptions

4.1.2.1 Use case: Taking an exam

Use Case 1	Taking an exam
Goal in Context	The student registers for a course and then takes the exam.
Scope & Level	System, Summary

Preconditions	The student should be logged into the system. The student should have registered for the course under the selected department. The teacher should have approved the registration of the student.
Success End Conditions	The student takes the exam for the course and would receive a grade for the exam from the teacher which can be checked on the application. The student can download the results.
Failed End Condition	Student is unable to take the exam due disapproval from the teacher.
Primary, Secondary, Actors	Student
Trigger	The student registered for a course.
Description	1. The student logs into the system 2. The student clicks on the exam section where the exam link would be provided. 3. The student starts the exam after going through the instructions. 4. The student submits the exam.
Extensions	4a. If the student does not submit the exam before the duration of the exam the system will submit automatically.
Variations	None
Priority	Top
Open Issues	What if the student is not notified about the disapproval of the course from the teacher?
Due Date	Week 7

4.1.2.2 Use case: Setting an exam

Use Case 2	Setting an exam
Goal in Context	The Teacher sets the question paper for the exam.
Scope & Level	System, Summary
Preconditions	The Teacher should be logged into the system. The Teacher should have the access to set a exam

Success End Conditions	The Teacher will be able to set an exam for the students who have registered for the course and students will be able to take the exam.
Failed End Condition	The students registered for the course will not be able to take the exam without an examination being set by the Teacher.
Primary, Secondary, Actors	Teacher and Student
Trigger	The Teacher has the access to set the exam for a course
Description	1. The teacher selects the course for which exam needs to be set. 2. The teacher decides in the examination pattern. 3. The teacher selects the questions from the question bank for the exam. 4. The teacher uploads the exam into the system. 5. The student takes the exam once it is made available in the system by the teacher. 6. Student will get a notification on the previous day of the exam.
Extensions	2a. The teacher decides on the instruction for the exam, time, total score and pass mark for the exam 3a. Based on the type of exam the teacher will select True/False, Long answer, Short Answer, Multiple choice questions from the question bank.
Variations	None
Priority	Top
Open Issues	What if the teacher does not make the exam available for the students to take it?
Due Date	Week 7

4.1.2.3 Use case: Grading an exam

Use Case 3	Providing grade to Student
Goal in Context	The teacher grades the exam given by the student
Scope & Level	System, Summary

Preconditions	The teacher must have logged into the system. The student should have given the exam for the course
Success End Conditions	After the student takes the exam grade will be provided by the teacher for long and short answer questions.
Failed End Condition	Student will not be getting the grade for a course if the teacher does not check the answers given by the student.
Primary,Secondary, Actors	Teacher and Student
Trigger	Student have given the exam for a course
Description	1. The teacher logs into the system. 2. The teacher searches for the exam using exam ID or course for grading. 3. The teacher checks the answer written by all the students and provide grade. 4. Auto grading is done by the application for MCQ. 5. Bonus marks can be added by the Teacher for a student. 6. The teacher uploads the grade on the application. 7. The student can view the results i.e., grade and download a copy of it
Extensions	3a. The teacher will check the answers himself if it is short/long answers. 3b. For other type of examination, the grade will be provided by the system.
Variations	None
Priority	Top
Open Issues	What if the teacher doesn't upload the grade on the system?
Due Date	Week 7

4.3 Quality attributes

Usability - Usability is a measure of how easy it is for users to use a software program. A software program with good usability is easy to learn and use, and it is intuitive and efficient for the user to accomplish their tasks. The usability of a software program can be improved through user testing and user-centered design, which focus on the needs and abilities of the user. A software program with good usability can help users to be more productive and

satisfied with their work.

Availability - Availability is a measure of how reliable and accessible a software program is for users. A software program with good availability is always accessible and usable for the user, even under challenging conditions such as high workloads or network outages. The availability of a software program can be improved through techniques such as redundant servers, backup systems, and load balancing. A software program with good availability can help users to be more productive and satisfied with their work.

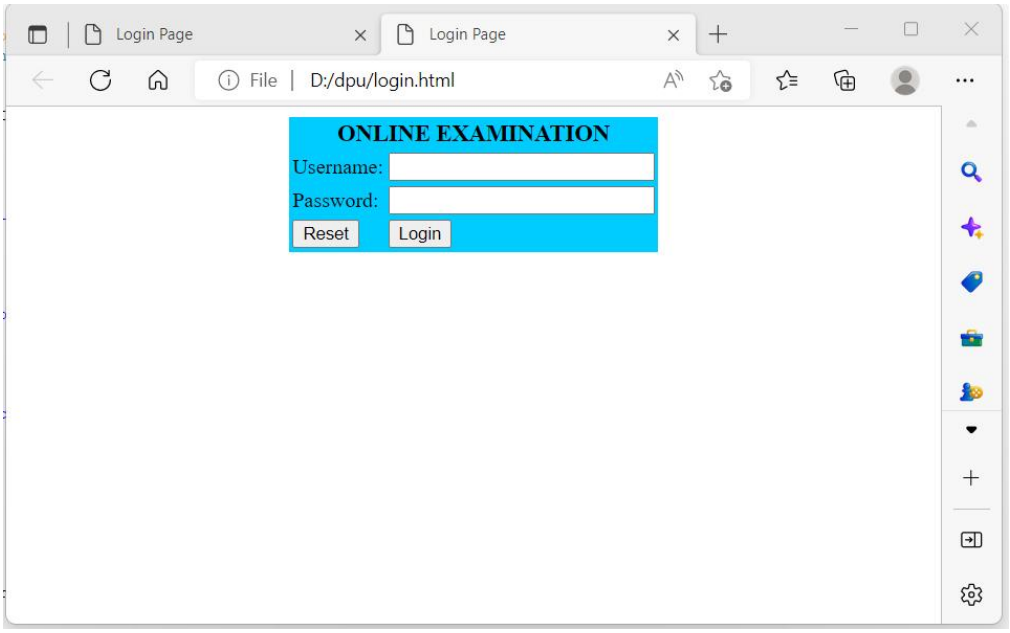
Flexibility - Flexibility is a measure of how easily a software program can be adapted to changes in the user's needs or the environment in which it is used. A software program with good flexibility is easy to modify and extend, and it can support a wide range of users and use cases. The flexibility of a software program can be improved through modular design and the use of open standards and APIs. A software program with good flexibility can help users to be more productive and satisfied with their work.

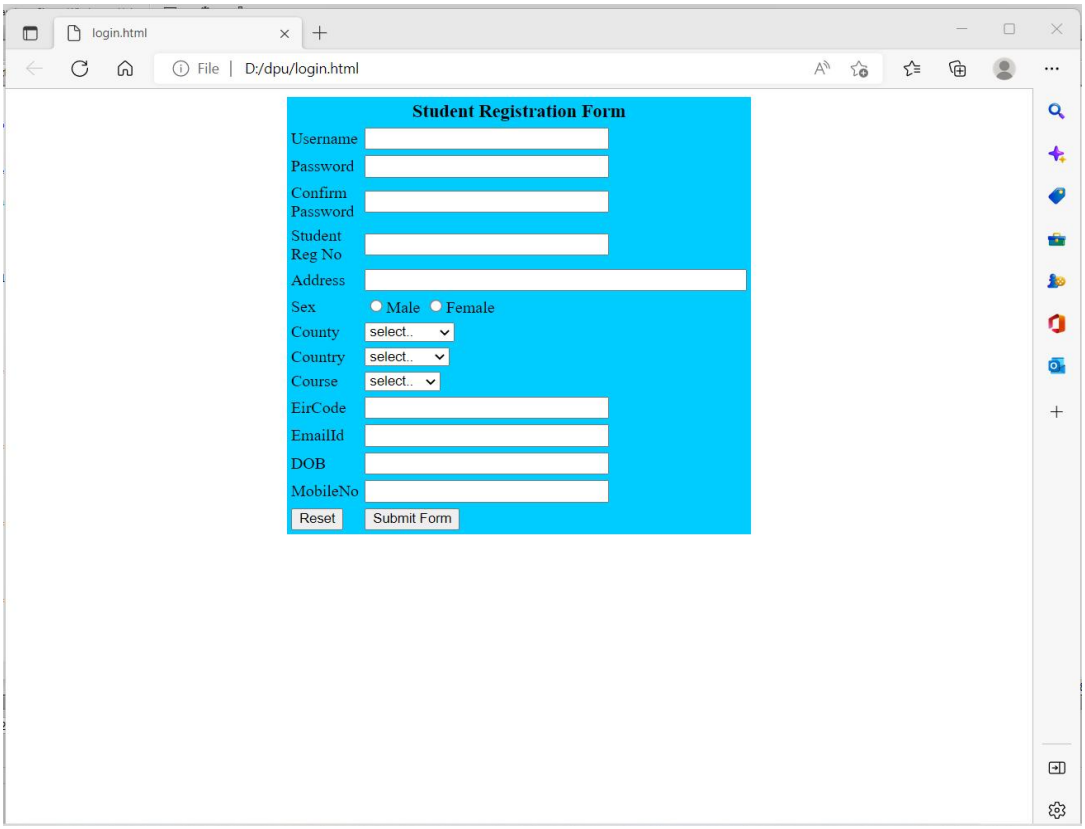
Maintainability - Maintainability is a measure of how easily a software program can be modified or maintained over time. A software program with good maintainability is easy to update, fix, and optimize, and it can be maintained by a small team of developers without significant disruptions to the user. The maintainability of a software program can be improved through good software design and documentation, and by following established software development standards and practices. A software program with good maintainability can help users to be more productive and satisfied with their work.

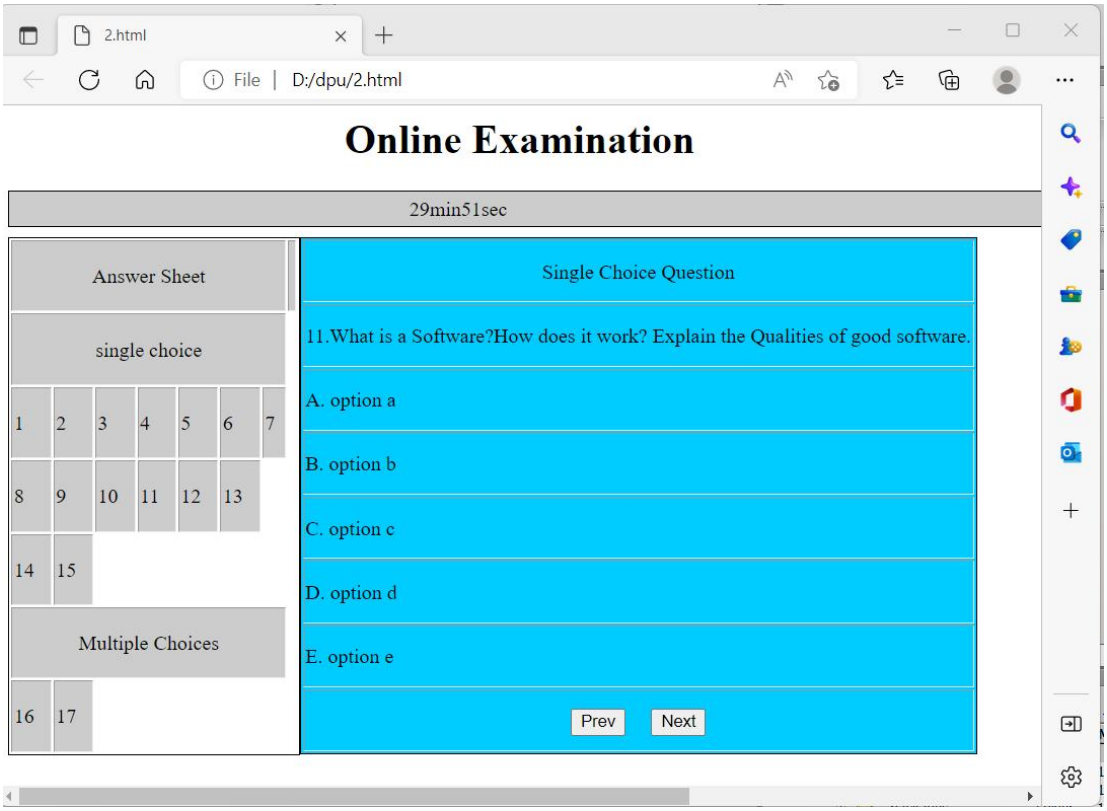
Portability - Portability is a measure of how easily a software program can be transferred or used on different hardware or software platforms. A software program with good portability is easy to install and run on different types of computers or operating systems, and it can be used by a wide range of users. The portability of a software program can be improved through the use of portable languages and development frameworks, and by following established software development standards and practices. A software program with good portability can help users to be more productive and satisfied with their work.

4.4 GUI Prototypes

4.4.1 Screenshot of the pages

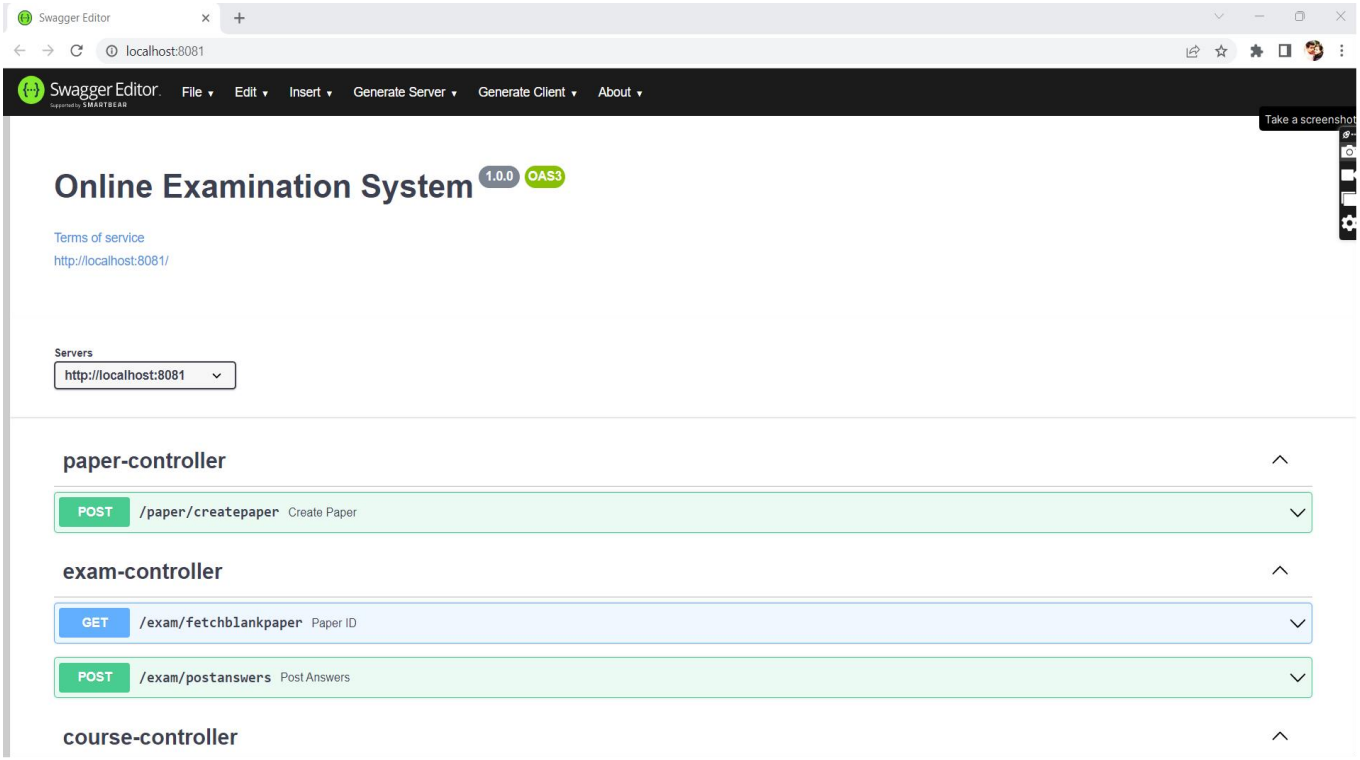


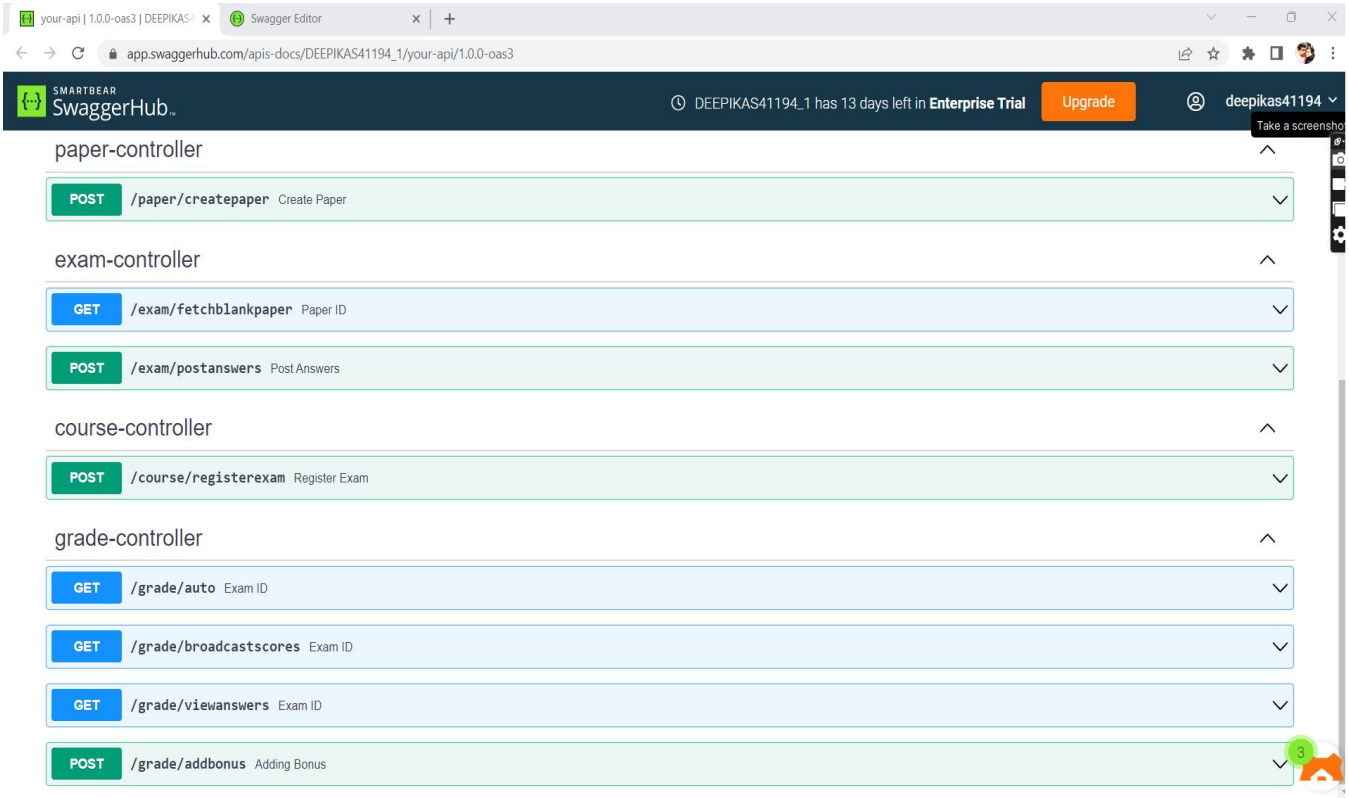




4.4.2 Swagger

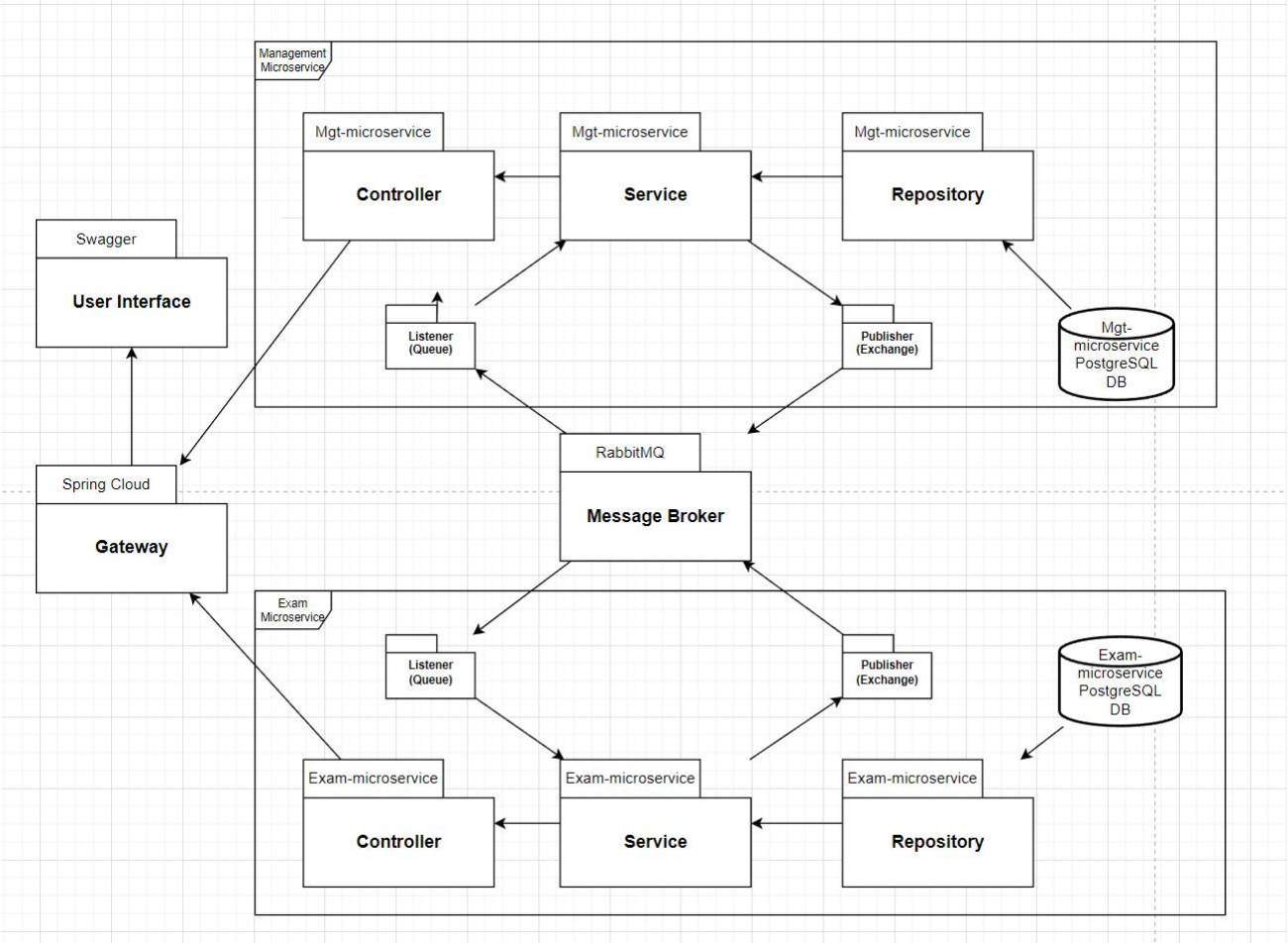
Swagger UI offers a web-based UI that provides information about the service, using the generated OpenAPI specification. Both Swashbuckle and NSwag include an embedded version of Swagger UI. Swagger project (Hosting) is a middleware for API documentation which has been configured in our project. The swagger UI could be accessed using the link http://localhost:8081/ and we have used it for testing out the software.


Swagger Editor
localhost:8081
Swagger Editor
File
Edit
Insert
Generate Server
Generate Client
About
Online Examination System
1.0.0
OAS3
Terms of service
http://localhost:8081/
Servers
http://localhost:8081
paper-controller
POST /paper/createpaper Create Paper
exam-controller
GET /exam/fetchblankpaper Paper ID
POST /exam/postanswers Post Answers
course-controller


your-api | 1.0.0-oas3 | DEEPIKAS
Swagger Editor
app.swaggerhub.com/apis-docs/DEEPIKAS41194_1/your-api/1.0.0-oas3
SMARTBEAR
SwaggerHub
DEEPIKAS41194_1 has 13 days left in Enterprise Trial
Upgrade
deepikas41194
paper-controller
POST /paper/createpaper Create Paper
exam-controller
GET /exam/fetchblankpaper Paper ID
POST /exam/postanswers Post Answers
course-controller
POST /course/registerexam Register Exam
grade-controller
GET /grade/auto Exam ID
GET /grade/broadcastscores Exam ID
GET /grade/viewanswers Exam ID
POST /grade/addbonus Adding Bonus

5. System Architecture

5.1 System Architecture with package diagram



This is the architecture diagram of the examination system, based on the event-driven microservice architecture. The system is divided into two microservices, one is the management microservice that manages the relationship between courses, teachers, and students. The other one is the examination microservice that handles test paper generation, examination, and scoring.

For each microservice, we use Spring Boot as the development framework, use MVC architecture, and have an individual database. Instead of using HTTP to communicate directly between microservices, we use RabbitMQ event-based communication to decouple the system and increase scalability.

Spring Cloud Gateway is used as a gateway to the whole system, providing a unified interface to the UI and hiding the system implementation.

5.2 Architecture patterns considered for the system

5.3.1 MVC Architecture

The Model-View-Controller (MVC) architecture is a design pattern that separates an application into three main components: the model, the view, and the controller.
The model represents the data and business logic of the application. It manages the behavior and data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller).
The view is the user interface of the application. It presents the model data to the user and provides a way for the user to interact with the data and the application.
The controller is the middleman between the model and the view. It receives user input, sends commands to the model to update the model's state, and sends commands to the view to change the view's presentation of the model.
In summary, the MVC architecture separates an application into three main components: the model, the view, and the controller. This separation of concerns allows for more modular and maintainable code.

5.3.2 Event-driven architecture pattern (Independent research)

Event-driven architecture is a design pattern in which the application is built as a series of event-driven components that communicate with each other by producing and consuming events. In this architecture, an event is a user-defined message that describes something that has happened in the system. When an event is produced, it is sent to an event queue, where it is processed by the appropriate event-driven component.
Event-driven architecture has several benefits. It allows for asynchronous communication between components, which can improve the performance and scalability of the application. It also promotes loose coupling between components, which makes the application easier to maintain and extend.
In summary, event-driven architecture is a design pattern in which the application is built as a series of event-driven components that communicate with each other by producing and consuming events. This architecture allows for asynchronous communication and promotes loose coupling between components.

5.3.3 Microservices architecture (Independent research)

Microservices architecture is a design pattern in which a large application is built as a suite of independently deployable, small, modular services. Each service runs in its own process and communicates with other services through well-defined interfaces, typically using a lightweight mechanism such as an HTTP resource API. This architecture allows for greater flexibility, scalability, and maintainability compared to a monolithic architecture, where the entire application is built as a single, integrated unit.

5.3 Tech stack

Spring Boot
Spring Boot is a Java-based framework that provides a pre-configured set of features for building production-ready Spring-based applications.

Spring MVC
Spring MVC is a request-based framework for building web applications that uses the Model-View-Controller design pattern. It is a part of the Spring Framework and is used to create web applications that can handle complex requests, rendering dynamic content, and integrating with other web frameworks and technologies.

Spring JPA
Spring JPA is a part of the Spring Framework that provides a simple and consistent approach to data access. It enables developers to build robust and efficient persistence layers for their applications, using the Java Persistence API (JPA) and other related technologies.

PostgreSQL
PostgreSQL is a powerful, open-source object-relational database system with a strong reputation for reliability, feature robustness, and performance. It is used by many organizations and individuals as a primary data store for a wide range of applications, including web-based applications, data warehousing, and business intelligence.

Spring AMQP
Spring AMQP is a Spring-based framework that simplifies the development of applications that use the Advanced Message Queuing Protocol (AMQP) for asynchronous messaging. It provides a familiar Spring-like programming model, using dependency injection and template-based messaging to reduce the complexity of implementing AMQP-based messaging solutions.

Spring Cloud Gateway
Spring Cloud Gateway is a reactive Gateway built on top of the Spring Framework and Spring Boot 2. It provides a simple, yet effective way to route, monitor, and secure your microservice applications. It is built on the Project Reactor, which provides a non-blocking API for building reactive microservice applications. This allows the gateway to handle high volumes of traffic with low latency, making it an ideal choice for building cloud-native applications.

RabbitMQ
RabbitMQ is a popular open-source message broker that implements the Advanced Message Queuing Protocol (AMQP) and other messaging protocols. It is designed to be fast, efficient, and easy to use, and is used by many organizations to enable the exchange of data between different applications and systems.

Lombok
Lombok provides annotations and tools to reduce boilerplate code in Java programs. The Lombok plugin integrates with Java IDEs to provide a seamless experience for developers using Lombok in their projects.

Fasterxml.Jackson
FasterXML.Jackson is a popular Java library for serializing and deserializing Java objects to and from JSON. It is highly customizable and offers advanced features, making it a popular choice for many Java developers. FasterXML.Jackson is part of the FasterXML family of JSON libraries.

Maven
Maven is a build automation tool used primarily for Java projects. It provides a uniform build system and is supported by a large ecosystem of plugins and integrations. Maven is known for its simplicity and flexibility and is widely used to automate the build and deployment of software projects.

Swagger
Swagger is a popular tool for designing, documenting, and consuming RESTful APIs. It provides a simple and interactive way to test and debug APIs, and helps developers to easily understand and use APIs. Swagger is supported by a large ecosystem of tools, libraries, and integrations, and is widely used by developers and organizations to design and build APIs.

Sonar Cloud
SonarCloud is a cloud-based code analysis platform for projects using Git. It provides static analysis, unit test coverage, and code duplication detection for over 20 programming languages. SonarCloud is free for open source projects, and offers paid plans for private projects. It integrates with popular continuous integration tools and offers a web-based interface for developers to review and discuss code quality issues.

Github
GitHub is a web-based platform for hosting and collaborating on software projects. It provides version control, bug tracking, and project management features, and is widely used by individuals and organizations to host and share code. GitHub is built on top of the Git version control system, and offers a variety of tools and integrations for developers to collaborate on projects and build software.

6. Sketches

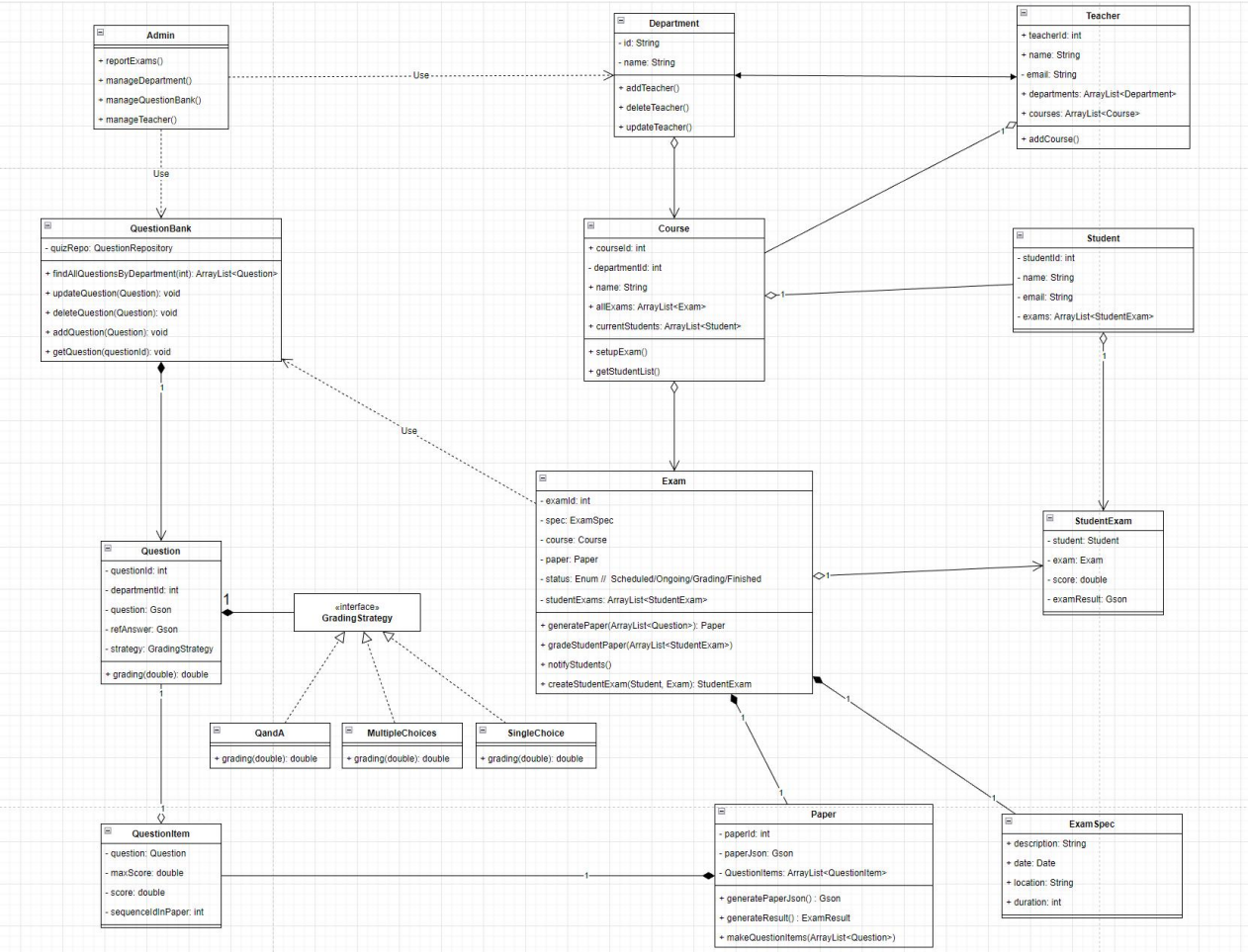
6.1 List of candidate objects

exammsv/command/AddBonusCommand
exammsv/command/AssemblePaperCommand
exammsv/command/BroadcastScoreCommand
exammsv/command/CalculateScoreCommand
exammsv/command/CheckGradeStateCommand
exammsv/command/GradeCommand
exammsv/command/GradeCommandChainBuilder
exammsv/command/ManuallyGradeCommand
exammsv/command/SubmitCommand
exammsv/configuration/AMQPConfiguration
exammsv/configuration/JsonConfiguration
exammsv/controller/GradeController
exammsv/controller/PaperController
exammsv/domain/Blankpaper
exammsv/domain/Question
exammsv/domain/StudentExamResult
exammsv/dto/Answer
exammsv/dto/AnswerDTO
exammsv/dto/AssembledAnswer
exammsv/dto/AssembledAnswerDTO
exammsv/dto/AssembledAnswerResultDecorator
exammsv/dto/BroadcastDTO
exammsv/dto/ManuallyGradeDTO
exammsv/dto/ManullyGrade
exammsv/dto/QuestionPOJO
exammsv/dto/QuestionSetting
exammsv/dto/QuestionSettingDTO
exammsv/dto/QuestionSettingQuestionDecorator
exammsv/ExammsvApplication
exammsv/interpreter/AddInterpreter
exammsv/interpreter/BonusCalculator
exammsv/interpreter/ExpressionParser
exammsv/interpreter/Interpreter
exammsv/interpreter/MultiplyInterpreter
exammsv/interpreter/NumberInterpreter
exammsv/interpreter/OperatorUtil
exammsv/mq/AbstractPublisher
exammsv/mq/event/Score
exammsv/mq/event/ScoreEvent
exammsv/mq/event/StudentExamEvent

exammsv/mq/QueuesListener
exammsv/mq/ScorePublisher
exammsv/repository/BlankpaperRepository
exammsv/repository/QuestionRepository
exammsv/repository/StudentExamResultRepository
exammsv/service/GradeService
exammsv/service/PaperService
exammsv/strategy/MultipleChoiceStrategy
exammsv/strategy/SingleChoiceStrategy
exammsv/strategy/Strategy
exammsv/strategy/StrategyFactory
exammsv/strategy/WritingStrategy
mgtmsv/configuration/AMQPConfiguration
mgtmsv/configuration/JsonConfiguration
mgtmsv/controller/CourseController
mgtmsv/controller/ReportController
mgtmsv/controller/RoleController
mgtmsv/domain/Course
mgtmsv/domain/CourseRepository
mgtmsv/domain/dto/StudentsExamDTO
mgtmsv/domain/Exam
mgtmsv/domain/ExamRepository
mgtmsv/domain/Student
mgtmsv/domain/StudentRepository
mgtmsv/domain/Teacher
mgtmsv/domain/TeacherRepository
mgtmsv/mq/AbstractPublisher
mgtmsv/mq/event/BlankpaperEvent
mgtmsv/mq/event/Score
mgtmsv/mq/event/ScoreEvent
mgtmsv/mq/event/StudentsExamEvent
mgtmsv/mq/QueuesListener
mgtmsv/mq/StudentExamPublisher
mgtmsv/service/CourseService

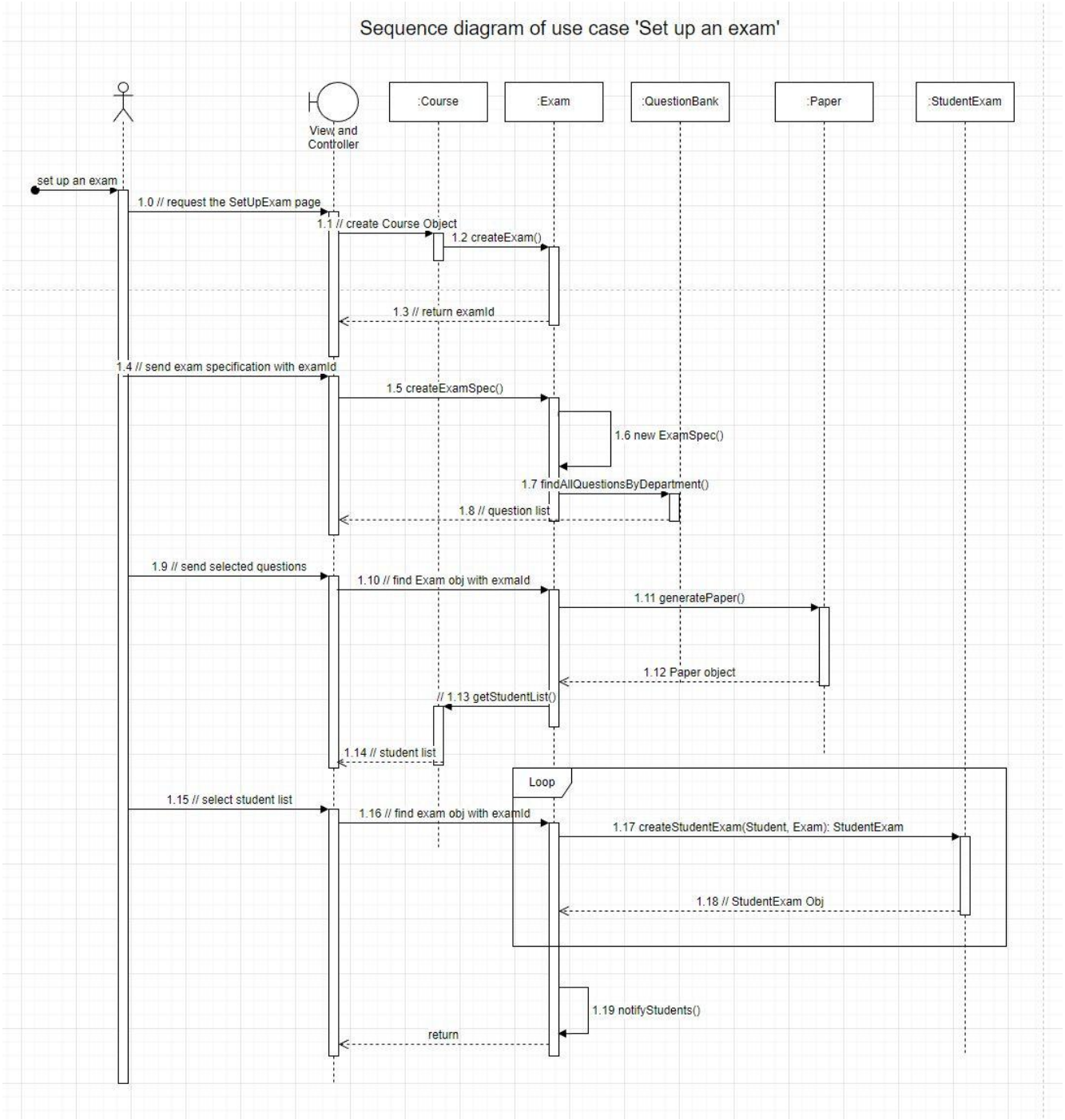
6.2 Analysis-time class diagram

This is the class diagram of the analyisis time, which is similar to what we implemented at last, but most of the classes are different. So this one is abandoned, please refer to section 10.2 to check the latest one.



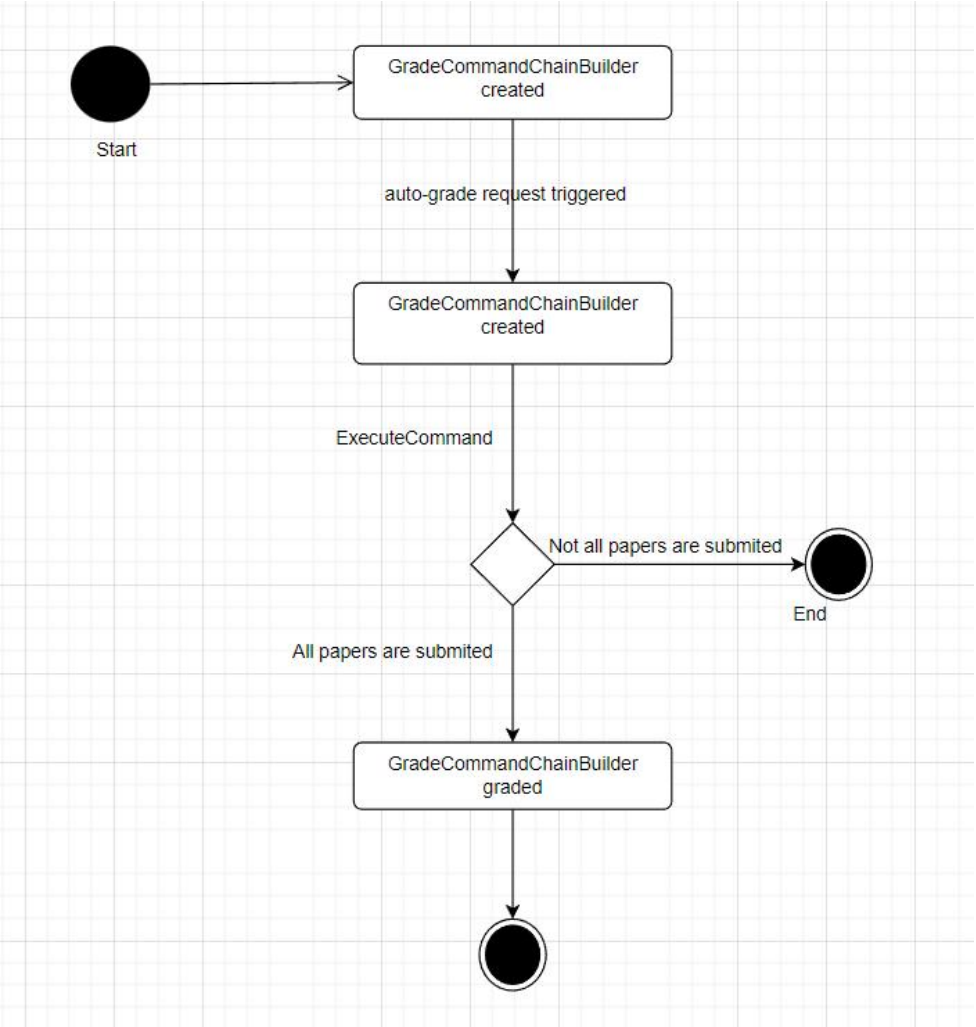
6.3 Analysis-time sequence diagram

This is the sequence diagram of analysis time, which is similar to what we implemented at last, but most of the classes are different. So this one is abandoned, please refer to section 10.3 to check the latest one.



6.4 State chart diagram

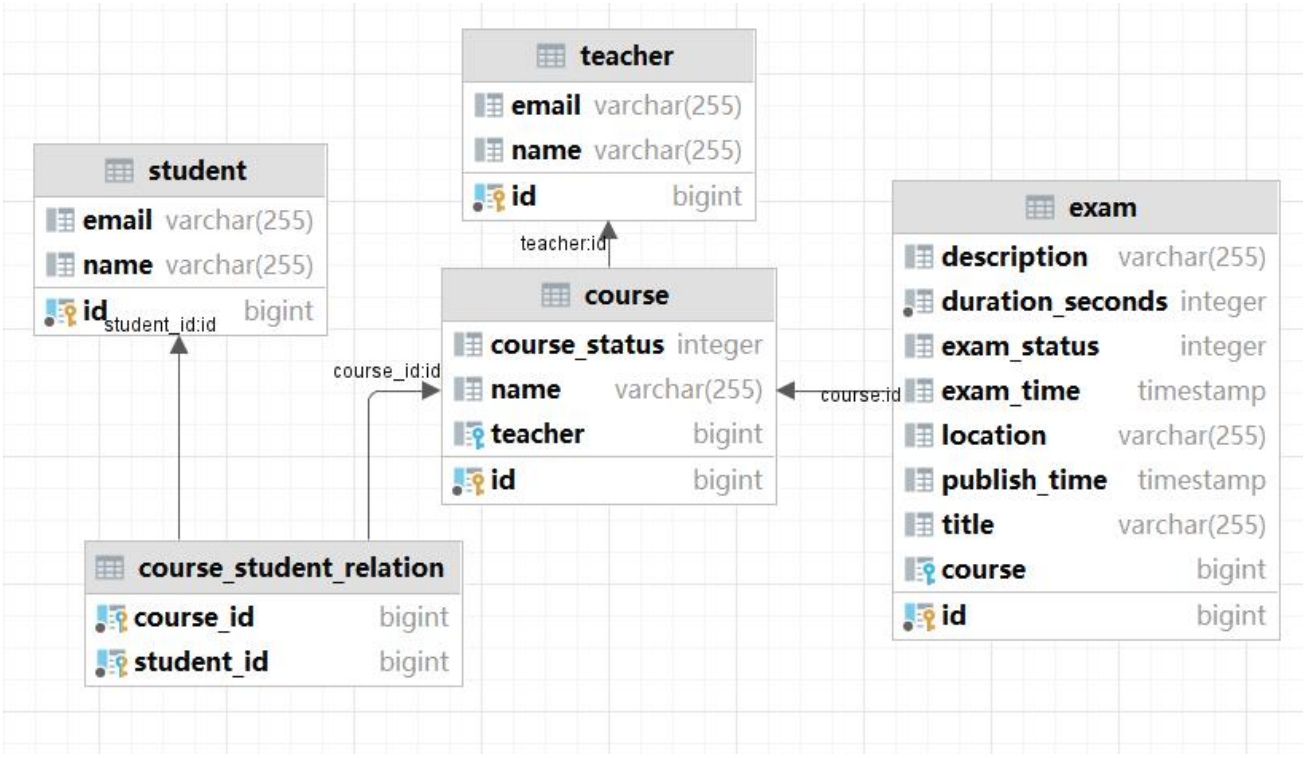
This is the state chart diagram of class GradeCommandChainBuilder in sequence diagram auto-grade(refer to section 10.3)

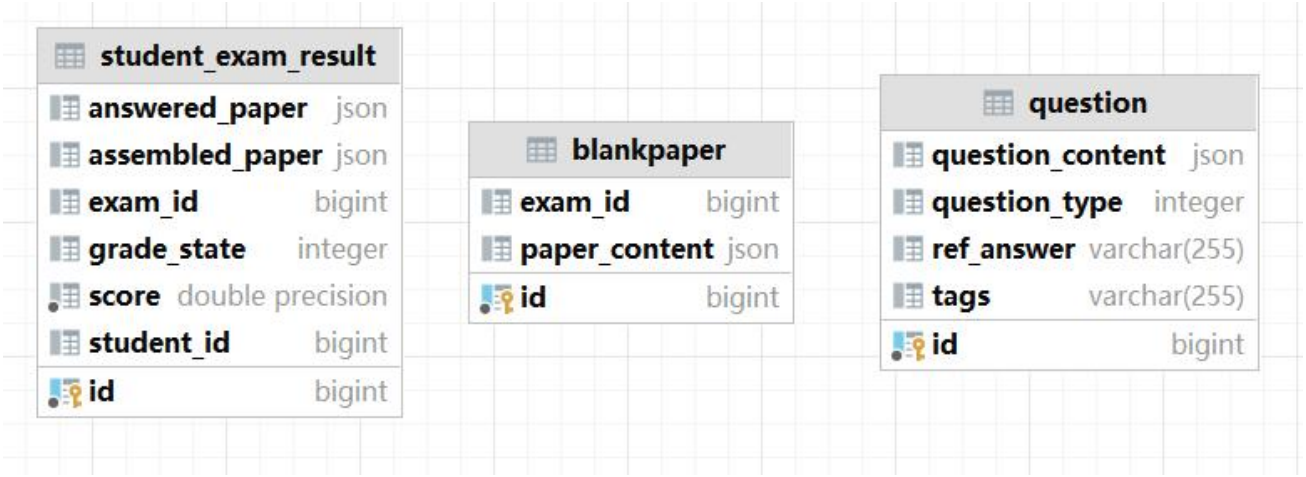


6.5 Entity relationship diagram

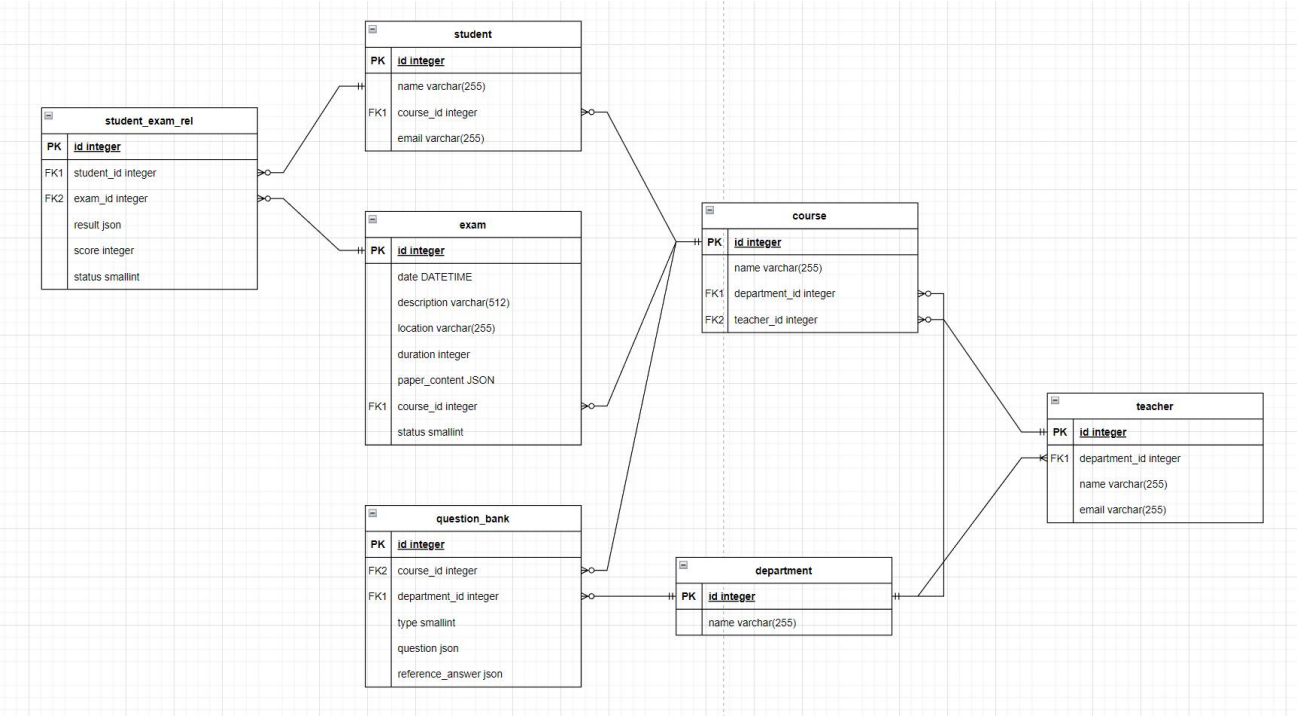
6.5.1 Design-time E-R diagram

We use two databases for two microservices. They have different schema. The first diagram is the schema of Mgt-Microservice. The second diagram if the schema of Exam-Microservice, which has no foreign keys, because it is mainly used for persistence JSON format data, so it can be migrated to a NoSQL database easily. You can compare this section with next section, to see how we split the database by refactoring the monolith to microservices.


teacher
email varchar(255)
name varchar(255)
id bigint
teacher:id
student
email varchar(255)
name varchar(255)
id bigint
student_id:id
course
course_status integer
name varchar(255)
teacher bigint
id bigint
course_id:id
course:id
exam
description varchar(255)
duration_seconds integer
exam_status integer
exam_time timestamp
location varchar(255)
publish_time timestamp
title varchar(255)
course bigint
id bigint
course_student_relation
course_id bigint
student_id bigint


student_exam_result
answered_paper json
assembled_paper json
exam_id bigint
grade_state integer
score double precision
student_id bigint
id bigint
blankpaper
exam_id bigint
paper_content json
id bigint
question
question_content json
question_type integer
ref_answer varchar(255)
tags varchar(255)
id bigint

6.5.2 Analysis-time E-R diagram



7. Transparency and traceability

7.1 A table listing the classes in each package, authors, and lines

lines	Author	package/classname
22	Wang	exammsv/command/AddBonusCommand.java
48	Wang	exammsv/command/AssemblePaperCommand.java
28	Wang	exammsv/command/BroadcastScoreCommand.java
12	Wang	exammsv/command/CalculateScoreCommand.java
25	Wang	exammsv/command/CheckGradeStateCommand.java
14	Wang	exammsv/command/GradeCommand.java
64	Wang	exammsv/command/GradeCommandChainBuilder.java
28	Wang	exammsv/command/ManuallyGradeCommand.java
27	Wang	exammsv/command/SubmitCommand.java
62	Wang	exammsv/configuration/AMQPConfiguration.java
14	Wang	exammsv/configuration/JsonConfiguration.java
48	Deepika	exammsv/controller/GradeController.java
56	Deepika	exammsv/controller/PaperController.java
42	Wang	exammsv/domain/Blankpaper.java
49	Wang	exammsv/domain/Question.java
59	Wang	exammsv/domain/StudentExamResult.java
14	Wang	exammsv/dto/Answer.java
15	Wang	exammsv/dto/AnswerDTO.java
16	Wang	exammsv/dto/AssembledAnswer.java
18	Wang	exammsv/dto/AssembledAnswerDTO.java
55	Rohini	exammsv/dto/AssembledAnswerResultDecorator.java
16	Wang	exammsv/dto/BroadcastDTO.java
18	Wang	exammsv/dto/ManuallyGradeDTO.java
15	Wang	exammsv/dto/ManullyGrade.java
19	Wang	exammsv/dto/QuestionPOJO.java
13	Wang	exammsv/dto/QuestionSetting.java
14	Wang	exammsv/dto/QuestionSettingDTO.java
26	Rohini	exammsv/dto/QuestionSettingQuestionDecorator.java
31	Deepika	exammsv/ExammsvApplication.java
17	Wang	exammsv/interpreter/AddInterpreter.java
22	Wang	exammsv/interpreter/BonusCalculator.java
24	Wang	exammsv/interpreter/ExpressionParser.java
5	Wang	exammsv/interpreter/Interpreter.java
17	Wang	exammsv/interpreter/MultiplyInterpreter.java
17	Wang	exammsv/interpreter/NumberInterpreter.java
17	Wang	exammsv/interpreter/OperatorUtil.java
22	Wang	exammsv/mq/AbstractPublisher.java

12	Wang	exammsv/mq/event/Score.java
12	Wang	exammsv/mq/event/ScoreEvent.java
11	Wang	exammsv/mq/event/StudentExamEvent.java
38	Wang	exammsv/mq/QueuesListener.java
16	Wang	exammsv/mq/ScorePublisher.java
24	Deepika	exammsv/repository/BlankpaperRepository.java
35	Deepika	exammsv/repository/QuestionRepository.java
26	Deepika	exammsv/repository/StudentExamResultRepository.java
27	Wang	exammsv/service/GradeService.java
65	Rohini	exammsv/service/PaperService.java
23	Deepika	exammsv/strategy/MultipleChoiceStrategy.java
13	Deepika	exammsv/strategy/SingleChoiceStrategy.java
5	Deepika	exammsv/strategy/Strategy.java
22	Deepika	exammsv/strategy/StrategyFactory.java
8	Deepika	exammsv/strategy/WritingStrategy.java
61	Wang	test/exammsv/ControllerTests.java
13	Rohini	test/exammsv/ExammsvApplicationTests.java
120	Deepika	../exammsv/pom.xml
89	Wang	../gateway/pom.xml
13	Wang	gateway/GatewayApplication.java
28	Wang	../gateway/src/main/resources/application.yml
13	Wang	gateway/GatewayApplicationTests.java
60	Wang	mgtmsv/configuration/AMQPConfiguration.java
14	Wang	mgtmsv/configuration/JsonConfiguration.java
53	Deepika	mgtmsv/controller/CourseController.java
31	Deepika	mgtmsv/controller/ReportController.java
25	Deepika	mgtmsv/controller/RoleController.java
59	Rohini	mgtmsv/domain/Course.java
10	Rohini	mgtmsv/domain/CourseRepository.java
11	Rohini	mgtmsv/domain/dto/StudentsExamDTO.java
55	Rohini	mgtmsv/domain/Exam.java
7	Rohini	mgtmsv/domain/ExamRepository.java
34	Rohini	mgtmsv/domain/Student.java
10	Rohini	mgtmsv/domain/StudentRepository.java
31	Rohini	mgtmsv/domain/Teacher.java
10	Rohini	mgtmsv/domain/TeacherRepository.java
59	Rohini	mgtmsv/InitData.java
30	Deepika	mgtmsv/MgtmsvApplication.java
121	Rohini	../mgtmsv/pom.xml
22	Wang	mgtmsv/mq/AbstractPublisher.java
9	Wang	mgtmsv/mq/event/BlankpaperEvent.java
10	Wang	mgtmsv/mq/event/Score.java
10	Wang	mgtmsv/mq/event/ScoreEvent.java

11	Wang	mgtmsv/mq/event/StudentsExamEvent.java
29	Wang	mgtmsv/mq/QueuesListener.java
16	Wang	mgtmsv/mq/StudentExamPublisher.java
18	Wang	mgtmsv/service/CourseService.java
29	Deepika	test/mgtmsv/ControllerTests.java
13	Rohini	test/mgtmsv/MgtmsvApplicationTests.java
35	Rohini	../workflow/build.yml

7.2 A table listing total lines of code implemented by each team member.

Total	2505
Wang	1312
Deepika	579
Rohini	614

7.3 A table that shows who did what with respect to patterns, testing, CI/CD, etc.

Wang: Architecture design, System design, E-R design, Event-driven pattern, Command pattern, Interpreter pattern, Observer pattern, MVC pattern.

Rohini: Decorator pattern, Implementation of Controller and repository layer, CI/CD, Testing, UI layer implementation.

Deepika: Strategy pattern, Factory pattern, Implementation of Controller and repository layer, Testing, UI design.

8. Code Snippets and brief descriptions

All the code can be checked at Github :
https://github.com/wang-chonghuan/ExamMicroservices.git

8.1 MVC architecture pattern

The Model-View-Controller (MVC) architecture is a design pattern that separates the application into three main components: the model, the view, and the controller.
The model represents the data and business logic of the application. It manages the behavior and data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller).
The view is the user interface of the application. It presents the model data to the user and provides a way for the user to interact with the data and the application.
The controller is the middleman between the model and the view. It receives user input, sends commands to the model to update the model's state, and sends commands to the view to change the view's presentation of the model.
The service layer is an optional component of the MVC architecture. It sits between the controller and the model, and provides additional logic and functionality for the application.
The repository is another optional component of the MVC architecture. It provides a way for the model to access the data stored in a database or other persistent storage.
In summary, the MVC architecture separates an application into three main components: the model, the view, and the controller. The service and repository layers may also be included as part of the architecture.

```java
package com.wang.exammsv.controller;

import ...

@Slf4j
@RequiredArgsConstructor
@RestController
@RequestMapping("/grade")
public class GradeController {

    3 usages
    @Autowired
    private GradeService gradeService;

    @RequestMapping(value="/auto", method= RequestMethod.GET)
    public ResponseEntity<?> auto(@RequestParam long examId) throws Exception {
        gradeService.autoGrade(examId);
        return ResponseEntity.status(HttpStatus.OK).build();
    }

    // 1. add bonus(or "") to all scores
    // 2. send score to mgt-sv to update the scores there
    // modify all the scores of an exam, with an expression like: "10 + 1.2 * 1 +"
    // apply this to 50 : (50+10)*1.2+1=73
    @RequestMapping(value="/broadcastscores", method= RequestMethod.POST)
    public ResponseEntity<?> broadcastScoreWithBonus(@RequestBody BroadcastDTO broadcastDTO) throws Exception {
        gradeService.broadcastScoreWithBonus(broadcastDTO);
        return ResponseEntity.status(HttpStatus.OK).build();
    }

    // todo this one is trivial, leave it unimplemented, we default that all the question are graded automatically
    @RequestMapping(value="/manual", method= RequestMethod.POST)
    public ResponseEntity<?> manual(@RequestBody ManuallyGradeDTO dto) throws Exception {
        gradeService.manuallyGrade(dto);
        return ResponseEntity.ok().body("");
    }
}
```

This is a component of the Controller layer, which defines the Rest-APIs and URL, and pass the request to Service layer.

```java
package com.wang.exammsv.service;

import ...

2 usages
@Slf4j
@Service
public class GradeService {
    3 usages
    @Autowired
    private GradeCommandChainBuilder gradeCommandChainBuilder;

    1 usage
    public void broadcastScoreWithBonus(BroadcastDTO dto) {
        gradeCommandChainBuilder.bonusAndBroadcastProcess(dto.getExpression()).executeCommands(dto.getExamId());
    }

    1 usage
    public void autoGrade(long examId) { gradeCommandChainBuilder.autoGradeProcess().executeCommands(examId); }

    1 usage
    public void manuallyGrade(ManuallyGradeDTO dto) {
        gradeCommandChainBuilder.manuallyGradeProcess(dto).executeCommands(dto.getExamId());
    }
}
```

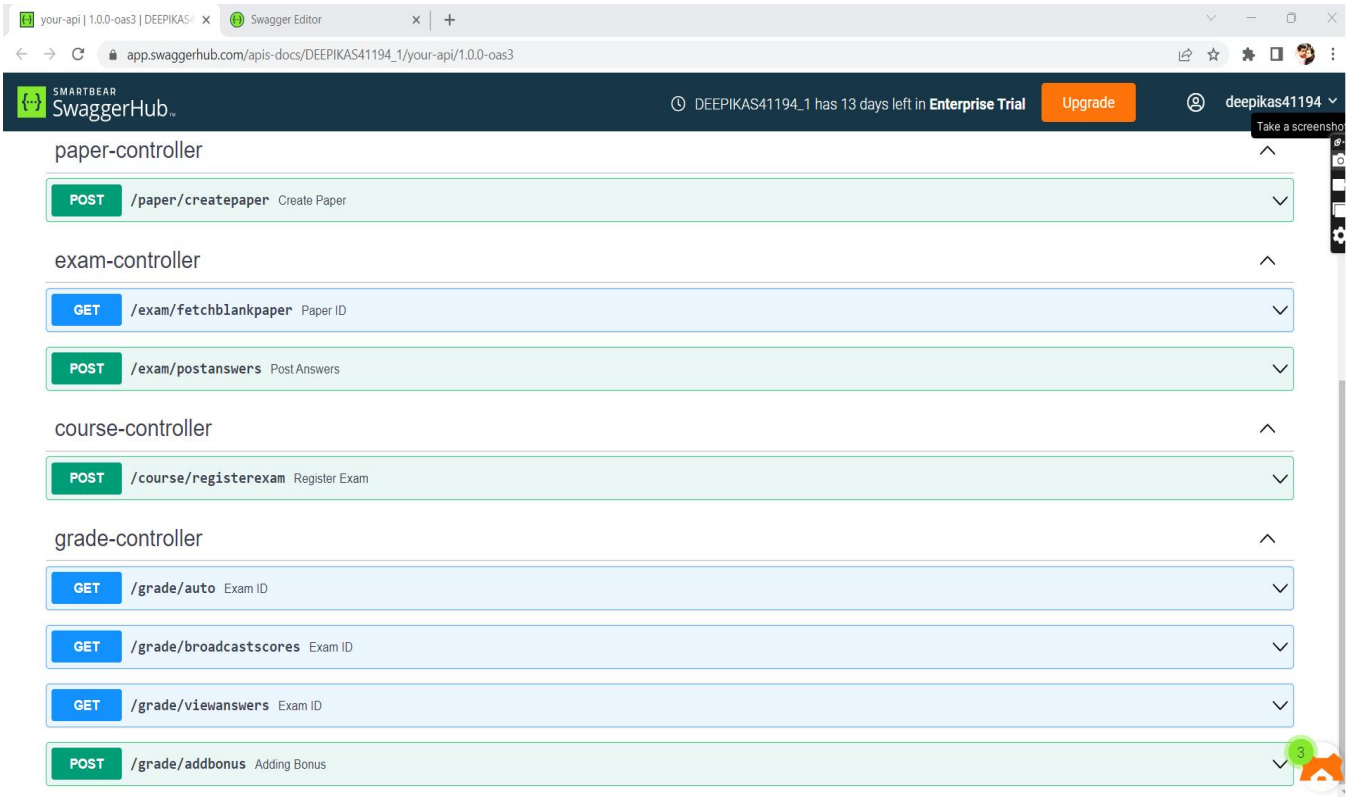
This is one of the components of the Sevice layer, which contains most of the business logic, and use design patterns to organize the logic, and contains the Repository layer to deal with the data persistence with database.

```java
package com.wang.exammsv.repository;

import ...

13 usages
public interface StudentExamResultRepository extends CrudRepository<StudentExamResult, Long> {
    1 usage
    List<StudentExamResult> findByExamId(long examId);

    3 usages
    List<StudentExamResult> findByStudentIdAndExamId(@Param("student_id")long studentId, @Param("exam_id")long examId);

    1 usage
    default void updateAnsweredPaper(StudentExamResult result) {
        var oldResult : Optional<StudentExamResult> = findByStudentIdAndExamId(
                result.getStudentId(), result.getExamId()).stream().findFirst();
        // bug if a student resubmit his paper, the state should not be changed to submit, bcz other students may be all fullygraded
        if(oldResult.isPresent()) {
            oldResult.get().setAnsweredpaper(result.getAnsweredpaper());
            oldResult.get().setGradeState(result.getGradeState());
            save(oldResult.get());
        } else {
            save(result);
        }
    }
}
```

This is the Repository layer. It is simple because we learn and use Spring JPA to make it simple. It undertakes all the CRUD operation with database.

```java
package com.wang.exammsv.domain;

import ...

16 usages
@TypeDefs({@TypeDef(name = "json", typeClass = JsonType.class)})
@Entity
@Data
@AllArgsConstructor
@NoArgsConstructor
public class StudentExamResult {
    @Id
    @GeneratedValue
    private Long id;
    2 usages
    private Long studentId; // fk of Student in mqtmsv
    2 usages
    private Long examId; // fk of Exam in mqtmsv
    1 usage
    private double score;

    public void setScore(double score) { this.score = Double.parseDouble(new DecimalFormat( pattern: "#.00").format(score)); }

    1 usage
    @Type(type = "json")
    @Column(name = "answered_paper", columnDefinition = "json")
    private Map<String, Object> answeredpaper = new HashMap<>();

    @Type(type = "json")
    @Column(name = "assembled_paper", columnDefinition = "json")
    private Map<String, Object> assembledpaper = new HashMap<>();

    2 usages
    @Enumerated(EnumType.ORDINAL)
    private GradeCommand.GradeState gradeState;

    // used when post answered paper
    1 usage
    public StudentExamResult(Long studentId, Long examId, Map<String, Object> answeredpaper, GradeCommand.GradeState gradeState) {
        this.studentId = studentId;
        this.examId = examId;
        this.answeredpaper = answeredpaper;
        this.gradeState = gradeState; // first create the result, the student hasnt taken the exam
    }
```

This is the domain/model/database definition in our java project. This Entity class will be directly mapped to a table in database.



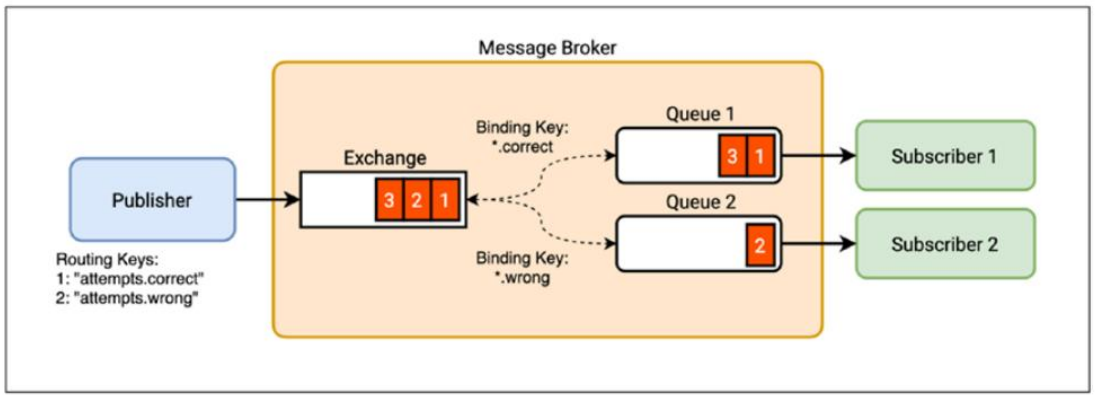
This is the View layer, we use Swagger to simulate the view layer. View is mainly for representing and interacting with the REST-APIs.

8.2 Event-driven architecture pattern

Even-driven architecture pattern is based on the Publish-Subscribe pattern. Instead of addressing the data to a specific destination, publishers classify and send events without any knowledge of the parts of the system that receive them, the subscribers. These event consumers don't need to be aware of the publishers' logic either. This change of paradigm makes our system loosely coupled and scalable.

In this project, we have two microservices, one of them is mgt-microservice, which targets on teachers and students user information management and the CRUD operations of the courses and exams. the other is exam-microservice, which targets on the exam paper creation and dealing with the process of taking and grading exams. The communication between them can be a REST-API synchronous call or even asynchronous call, but either of them will make the two microservices more coupled.

So we choose the even-driven architecture to handle inter-service communication. When the teacher set up an exam in the mgt-microservice, it will send an event of the list of students that registered to this exam to the exam-microservice. When the teacher finish grading an exam, the exam-microservice will send an event of the scores of all the students taken the exam to the mgt-microservice.



This diagram is how Message broker works in a microservices system. Notice the Exchange and Queue, they are key concepts in the Event architecture system.

```
# mgt-microservice
amqp.exchange.mgt=mgt.topic
amqp.queue.score=score.queue
amqp.routingkey.score=score.routingkey
# exam-microservice
amqp.exchange.exam=exam.topic
amqp.queue.studentexam=studentexam.queue
amqp.routingkey.studentexam=studentexam.routingkey
```

This is how to configure the three main components in properties.application file: exchang, queue, routingkey in each microservice. Only if the names are configured correctly, the rabbitMQ can work properly.

```java
package com.wang.exammsv.configuration;

import ...

@Configuration
public class AMQPConfiguration {

    // student doesn't need the switch, but it is created
    // so that it doesn't have to wait for the teacher to start before it can start
    // Switches are created idempotently, so it doesn't matter if they are created multiple times
    @Bean
    public TopicExchange mgtTopicExchange(@Value("mgt.topic") final String exchangeName) {
        return ExchangeBuilder.topicExchange(exchangeName).durable( isDurable: true).build();
    }

    @Bean
    public TopicExchange examTopicExchange(@Value("exam.topic") final String exchangeName) {
        return ExchangeBuilder.topicExchange(exchangeName).durable( isDurable: true).build();
    }

    @Bean
    public Queue studentExamQueue(@Value("studentexam.queue") final String queueName) {
        return QueueBuilder.durable(queueName).build();
    }

    @Bean
    public Binding studentExamBinding(final Queue studentExamQueue, final TopicExchange mgtTopicExchange) {
        return BindingBuilder.bind(studentExamQueue).to(mgtTopicExchange).with( routingKey: "studentexam.routingkey");
    }

    @Bean
    public RabbitListenerConfigurer rabbitListenerConfigurer(
            final MessageHandlerMethodFactory messageHandlerMethodFactory) {
        return c -> c.setMessageHandlerMethodFactory(messageHandlerMethodFactory);
    }
```

This is how to configure the Spring AMQP which is an adapter of RabbitMQ in Exam-Microservice. Here we can see all the Exchanges and Queues are created in Beans. And the Events represented by their names are binding to the Exchanges and Queues.

```java
package com.wang.exammsv.mq.event;

import ...

2 usages
@Data
public class StudentExamEvent {
    private long examId;
    private List<Long> studentIdList;
}
```

This is the definition of the Even StudentExamEvent, which will be passes from Mgt-Microservice to Exam-Microservice.

```java
package com.wang.mgtmsv.mq.event;

import ...

2 usages
@Data
public class ScoreEvent {
    private List<Score> scoreList;
    1 usage
    public static class Score {
        private long studentId;
        private long examId;
        private double score;
    }

}
```

This is how the ScoreEvent is defined. It will be passed from Exam-Microservice to Mgt-Microservice.
Please check the next section 8.3 to see how this message broker interact with listeners and publishers.

8.3 Observer pattern

Observer design pattern comes under behavioural design patterns. Observer design pattern states that define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

In our project, as described in 8.2, we apply observer pattern for the event-driven architecture.

```
# mgt msv
amqp.exchange.mgt=mgt.topic
amqp.queue.score=score.queue
amqp.routingkey.score=score.routingkey
# exam msv
amqp.exchange.exam=exam.topic
amqp.queue.studentexam=studentexam.queue
amqp.routingkey.studentexam=studentexam.routingkey
```

Code Snippet in properties.application
This is how the names of the exchange, queue and routingKey are configured in properties.application, and these names will be visible in both the message broker RabbitMQ and our Spring project. For example, the mgt-microservice has an exchange named mgt.topic, and a queue named score.queue. The exam-microservice has an exchange named exam.topic, and a queue named studentexam.queue.

```java
    @Bean
    public Binding scoreBinding(final Queue scoreQueue, final TopicExchange examTopicExchange) {
        return BindingBuilder.bind(scoreQueue).to(examTopicExchange).with( routingKey: "score.routingkey");
    }
```

Code snippet in com.wang.mgtmsv.configuration.AMQPConfiguration
This is a configuration that binds the scoreQueue of the mgt-microservice to the examTopicExchange of the exam-microservice with a routingKey score.routingkey. When the exam-microservice sends an event with a routingkey score.routingkey to the scoreQueue, the mgt-microservice will receive the event, because it has the scoreQueue.

```java
package com.wang.exammsv.mq;

import ...

@Slf4j
@Service
@RequiredArgsConstructor
public class QueuesListener {
    2 usages
    @Autowired
    private StudentExamResultRepository resultRepository;

    // when @RabbitListener is on class level, use @RabbitHandler to annotate the method
    @RabbitListener(queues="studentexam.queue")
    void handleStudentExamEvent(final StudentExamEvent event) {
        try {
            log.info("handleStudentExamEvent, examId {}, content {}", event.getExamId(), event.getStudentIdList());
            event.getStudentIdList().forEach(studentId -> {
                var result :Optional<StudentExamResult> =
                        resultRepository.findByStudentIdAndExamId(studentId, event.getExamId()).stream().findFirst();
                if(result.isEmpty()) {
                    resultRepository.save(new StudentExamResult(studentId, event.getExamId()));
                }
            });
        } catch (final Exception e) {
            log.error("error when trying to process handleStudentExamEvent", e);
            throw new AmqpRejectAndDontRequeueException(e);
        }
    }
}
```

This is how the exam-microservice handle the StudentExamEvent as a listener. When it receives the event, which contains the list of the studentIds that registered to a specific exam, it saves the student-exam-relations to the database.

```java
package com.wang.exammsv.mq;

import org.springframework.amqp.core.AmqpTemplate;


1 usage  1 inheritor
public abstract class AbstractPublisher {
    2 usages
    private final AmqpTemplate amqpTemplate;
    2 usages
    private final String topicExchangeName;
    2 usages
    private final String routingKey;

    1 usage
    public AbstractPublisher(final AmqpTemplate amqpTemplate,
                             final String topicExchange,
                             final String routingKey) {
        this.amqpTemplate = amqpTemplate;
        this.topicExchangeName = topicExchange;
        this.routingKey = routingKey;
    }

    1 usage
    public void publish(final Object event) { amqpTemplate.convertAndSend(topicExchangeName, routingKey, event); }
}
```

```java
package com.wang.exammsv.mq;

import ...

5 usages
@Slf4j
@Service
public class ScorePublisher extends AbstractPublisher {
    public ScorePublisher(final AmqpTemplate amqpTemplate,
                          @Value("exam.topic") final String topicExchange,
                          @Value("score.routingkey") final String routingKey) {
        super(amqpTemplate, topicExchange, routingKey);
    }
}
```

This is how the exam-microservice publishes the ScoreEvnet to the mgt-microservice. Actually, it broadcasts the ScoreEvent so that all the other microservices can receive it if there are any. It uses the API provided by the Spring AMQP which is an adapter of the RabbitMQ to send the event. The code is simple, but the learning and configuring is hard. Noticed that we specify the exam.topic and score.routingkey when sending the event.

The same observer pattern is also implemented in the mgt-microservice to make the two microservices and communicate each other in this way. Theses microcoservices don't know the existence of each other, well decoupled.

8.4 Command pattern and Chain of Responsibility Pattern (similar)

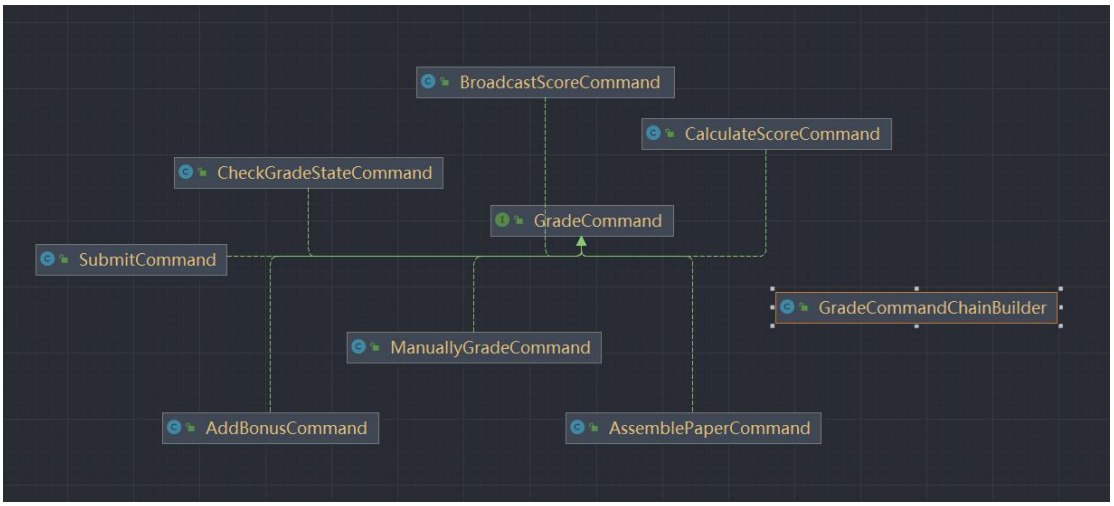
Command pattern is a data driven design pattern and falls under behavioral pattern category. A request is wrapped under an object as command and passed to invoker object. Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.

The chain of responsibility pattern creates a chain of receiver objects for a request. This pattern decouples sender and receiver of a request based on type of request. This pattern comes under behavioral patterns.

In our project, we combine these two patterns to make the main structure of how to grade an exam, which is the most important business logic in the project.
When the teacher is grading an exam, the system will first check if all the papers are submitted, then assemble each question with it's reference answer and store it into database for scoring and reviewing, then calculate the score of each question and sum them to the score of the exam of one student, at last it will save the score to the database. Then it will broadcast the score to other microservices like mgt-microservice or some statistics microservice.
If there are some writing questions in the paper, the teacher will have to grade it manually, which makes the chain above more complex. So we use command to represent each action above, and combine the commands to different chains to meet different requirement.

The code is in package com.wang.exammsv.service.command.
Check the code here:

https://github.com/wang-chonghuan/ExamMicroservices/tree/main/exammsv/src/main/java/com/wang/exammsv/service/command



This is the class diagram. There are many command subclasses, each of them represent an action. There superclass is GradeCommand. And the chain is built in the GradeCommandChainBuilder.

```
7 implementations
public interface GradeCommand {
    7 implementations
    public void execute(long examId, List<ResultGradeDecorator> resultDecoratorList) throws Exception;

    12 usages
    public enum GradeState {
        1 usage
        pending, submitted, autograded, fullygraded, broadcasted
    }
}
```

This is the interface of the Command Classes. It has only one method which is execute, which deals with all the papers students submitted in one exam.

```java
@Service
public class GradeCommandChainBuilder {
    1 usage
    public GradeCommandChainBuilder autoGradeProcess() {
        gradeCommandList.clear();
        gradeCommandList.addAll(List.of(
                new CheckGradeStateCommand(GradeCommand.GradeState.submitted),
                new AssemblePaperCommand(questionRepository, resultRepository),
                new CalculateScoreCommand(),
                new SubmitCommand(GradeCommand.GradeState.autograded, resultRepository)));
        return this;
    }

    1 usage
    public GradeCommandChainBuilder bonusAndBroadcastProcess(String expression) {
        gradeCommandList.clear();
        gradeCommandList.addAll(List.of(
                new CheckGradeStateCommand(GradeCommand.GradeState.fullygraded),
                new AddBonusCommand(expression),
                new BroadcastScoreCommand(scorePublisher),
                new SubmitCommand(GradeCommand.GradeState.broadcasted, resultRepository)));
        return this;
    }

    2 usages
    public void executeCommands(long examId) {
        List<StudentExamResult> resultList = resultRepository.findByExamId(examId);
        List<ResultGradeDecorator> resultDecoratorList = new ArrayList<>();
        resultList.forEach(result -> {
            resultDecoratorList.add(new ResultGradeDecorator(result));
        });
        gradeCommandList.forEach(command -> {
            try {
                command.execute(examId, resultDecoratorList);
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        });
    }
```

This is the chain of commands builder. When the teacher wants to automatically grade the paper, the system will build a autoGradProcess, which composites the four commands into a chain. Then execute the method executeCommands to call each command in a foreach loop. If there are more requirements, this chain builder and deal with it easily.

```java
1 usage
public class BroadcastScoreCommand implements GradeCommand {

    // Here the injection with Autowire will be a null pointer,
    // because the class is new at runtime and cannot be injected automatically.
    // The object to be injected automatically must be assembled at server startup
    2 usages
    private final ScorePublisher publisher;

    @Override
    public void execute(long examId, List<ResultGradeDecorator> resultDecoratorList) throws Exception {
        publisher.publish(new ScoreEvent(resultDecoratorList.stream() Stream<ResultGradeDecorator>
                .map(r -> new Score(r.getExamId(), r.getStudentId(), r.getScore())) Stream<Score>
                .collect(Collectors.toList())));
    }

    1 usage
    public BroadcastScoreCommand(ScorePublisher publisher) { this.publisher = publisher; }
}
```

This is the publish command which should be added at the end of the chain. When we calculate the score the save it to the database, this command can broadcast the list of scores in this exam to the mgt-microservice to do make the exam report and ranking.

```java
3 usages
public class SubmitCommand implements GradeCommand {
    2 usages
    private final GradeState gradeState;
    2 usages
    private final StudentExamResultRepository resultRepository;

    @Override
    public void execute(long examId, List<ResultGradeDecorator> resultDecoratorList) throws Exception {
        this.resultRepository.saveAll(resultDecoratorList.stream().map(r -> {
            r.getResult().setGradeState(gradeState);
            return r.getResult();
        }).collect(Collectors.toList()));
    }

    3 usages
    public SubmitCommand(GradeState gradeState, StudentExamResultRepository resultRepository) {
        this.gradeState = gradeState;
        this.resultRepository = resultRepository;
    }


}
```

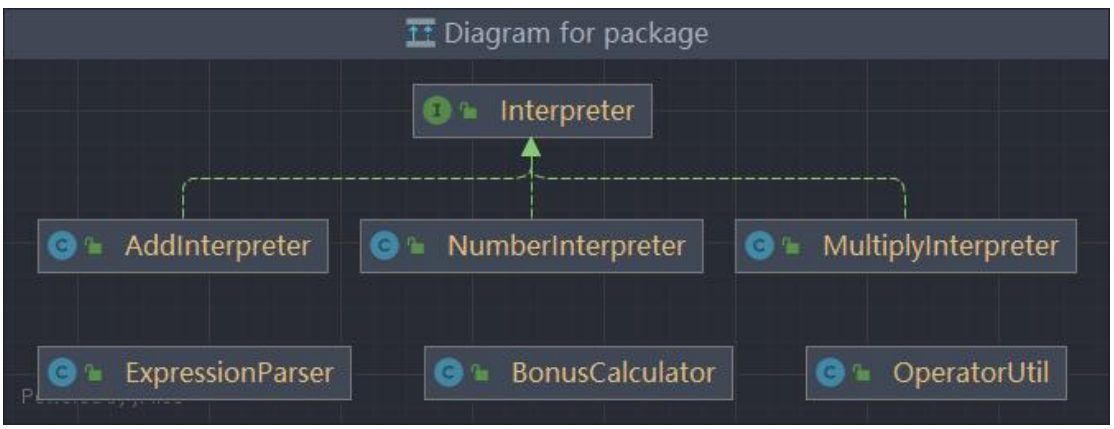
This is the submit command for saving the score and assembled paper to the database. When every previous action is done successfully , this command will be executed to keep the data integrity.

8.5 Interpreter pattern

Java interpreter design pattern comes under behavioural design patterns. It is used to provide a way to evaluate language grammar and provides an interpreter to deal with this grammar.

In our project, considering a situation that after the all the scores of an exam are calculated, what if the teacher wants to multiply each person's grade by a percentage, or add a bonus point? At this point he can pass in an expression string through the REST-api interface, and the interpreter pattern parser analyzes this expression to modify each person's test score. For example, the expression "10 + 1.2 * 1 +" is applied to a score of 50, that is (50 + 10) * 1.2 + 1 = 73, and the student's score is modified to 73.

The code is in this package com.wang.exammsv.service.interpreter;



This is the class diagram of this pattern in our project

```java
package com.wang.exammsv.service.interpreter;

public interface Interpreter {
    6 usages  3 implementations
    double interpret();
}
```

This is the interface of Interpreter

```java
package com.wang.exammsv.service.interpreter;

public class AddInterpreter implements Interpreter {

    2 usages
    private final Interpreter firstInterpreter;
    2 usages
    private final Interpreter secondInterpreter;

    1 usage
    public AddInterpreter(Interpreter firstInterpreter, Interpreter secondInterpreter) {
        this.firstInterpreter = firstInterpreter;
        this.secondInterpreter = secondInterpreter;
    }

    6 usages
    @Override
    public double interpret() { return firstInterpreter.interpret() + secondInterpreter.interpret(); }
}
```

This is the AddInterpreter which interprets its two interpreters with an ADD operation.

```java
package com.wang.exammsv.service.interpreter;

public class MultiplyInterpreter implements Interpreter {

    2 usages
    private final Interpreter firstInterpreter;
    2 usages
    private final Interpreter secondInterpreter;

    1 usage
    MultiplyInterpreter(Interpreter firstInterpreter, Interpreter secondInterpreter) {
        this.firstInterpreter = firstInterpreter;
        this.secondInterpreter = secondInterpreter;
    }

    6 usages
    @Override
    public double interpret() { return firstInterpreter.interpret() * secondInterpreter.interpret(); }
}
```

This is the MultiplyInterpreter which interprets two interpreters with an MULTIPLY operation.

```java
package com.wang.exammsv.service.interpreter;

public class NumberInterpreter implements Interpreter {

    3 usages
    private final double number;
    1 usage
    NumberInterpreter(double number) { this.number = number; }
    1 usage
    NumberInterpreter(String number) { this.number = Double.parseDouble(number); }

    6 usages
    @Override
    public double interpret() { return this.number; }
}
```

This is the terminal interpreter, when it deals with number, it just returns it, doing no more operations.

```
1 usage
public class ExpressionParser {
    5 usages
    private static final LinkedList<Interpreter> queue = new LinkedList<Interpreter>();

    1 usage
    public static double parse(String oriStr) {
        String[] splitStrList = oriStr.split( regex: " ");
        for(String curStr: splitStrList) {
            if(OperatorUtil.isSymbol(curStr)) {
                Interpreter firstInterpreter = queue.pop();
                Interpreter secondInterpreter = queue.pop();
                Interpreter expressionObject = OperatorUtil.getExpressionObject(
                        firstInterpreter, secondInterpreter, curStr);
                assert expressionObject != null;
                queue.push(new NumberInterpreter(expressionObject.interpret()));
            } else {
                queue.push(new NumberInterpreter(curStr));
            }
        }
        return queue.pop().interpret();
    }
}
```

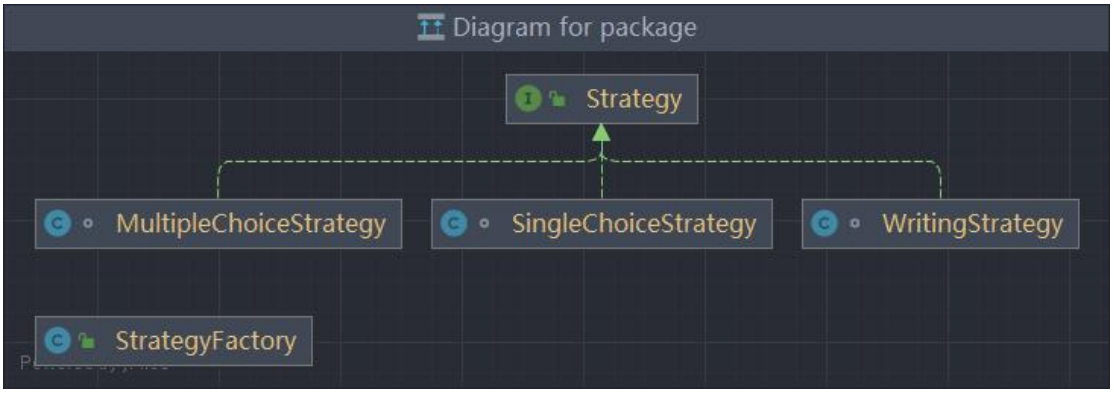
This is how to parse an expression. Create an interpreter by the character of the string and push it into a LIFO queue. When it's an operator interpreter, push the result into the queue, otherwise, just push the number interpreter. When it ends traversing the characters of the string, we get the value of this expression.

8.6 Strategy pattern

Java strategy design pattern comes under behavioural design patterns. Strategy design pattern states that define multiple algorithms for a specific task and let the client application choose the algorithm to be used.

In our project, we have three types of questions, they are single choice, multiple choices, and writing. They have different ways of calculation. So we will apply strategy pattern to deal with different strategies.

The code is in package com.wang.exammsv.service.strategy.



This is the class diagram of this pattern implemented in our project.

```java
public interface Strategy {
    1 usage   3 implementations
    double grade(String answer, String refAnswer, double mark);
}
```

This is the interface of the strategy. It needs the answer and reference answer to calculate to score.

```java
class SingleChoiceStrategy implements Strategy {

    1 usage
    @Override
    public double grade(String answer, String refAnswer, double mark) {
        if (answer.equals(refAnswer)) {
            return mark;
        } else {
            return 0.0;
        }
    }
}
```

This is the strategy for single choice question.

```java
1 usage
class MultipleChoiceStrategy implements Strategy {
    1 usage
    @Override
    public double grade(String answer, String refAnswer, double mark) {
        String[] answerArray = answer.split( regex: ",");
        List<String> refList = Arrays.asList(refAnswer.split( regex: ","));
        double eachMark = mark / (double) refList.size();
        double finalScore = 0.0;
        for (String eachAnswer : answerArray) {
            if (refList.contains(eachAnswer)) { // one right, accumulate one choice's mark
                finalScore += eachMark;
            } else { // one wrong, final score is 0
                finalScore = 0.0;
                break;
            }
        }
        return finalScore;
    }
}
```

This is the strategy for multiple choice question which has a totally different calculation of the score.

```java
    1 usage
    public void calculateScore() {
        assembledAnswerDTO.getAssembledAnswerList().forEach(assembledAnswer -> {
            Strategy strategy = StrategyFactory.create(assembledAnswer.getQuestionPOJO().getQuestionType());
            var score :double  = strategy.grade(
                    assembledAnswer.getAnswer().getAnswer(),
                    assembledAnswer.getQuestionPOJO().getRefAnswer(),
                    assembledAnswer.getAnswer().getMark());
            assembledAnswer.setScore(score);
            log.info("score {} examId {} studentId {} questionId {}",
                    score, assembledAnswerDTO.getExamId(), assembledAnswerDTO.getStudentId(),
                    assembledAnswer.getQuestionPOJO().getQuestionId());
        });
        result.setScore(assembledAnswerDTO.getAssembledAnswerList().stream()
                .mapToDouble(AssembledAnswer::getScore).sum());
    }
```

This is how to use this strategy. Create a strategy, call the interface, no need to care about the implementation.

8.7 Factory pattern

Factory pattern is one of the most used design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.
In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

In our project, when we create the strategy, we use a StrategyFactory, based on the type of the question to create it's corresponding Strategy

```
2 usages
public class StrategyFactory {

    1 usage
    public static Strategy create(Question.QuestionType qtype) {
        switch (qtype) {
            case SINGLE -> {
                return new SingleChoiceStrategy();
            }
            case MULTIPLE -> {
                return new MultipleChoiceStrategy();
            }
            case WRITING -> {
                return new WritingStrategy();
            }
        }
        return new SingleChoiceStrategy();
    }

}
```

This factory class returns the interface of the strategy without letting its user know which specific strategy it is.

8.8 Decorator pattern

Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class. This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

In our project, we need to combine the answer and the reference answer to calculate the score of this question. But we can's and any methods to these two methods. Because they are domain and DTO classes which should be POJO for serializing. So I use the decorator pattern to add a calculateScore method to the StudentExamResult class.

```java
@Slf4j
@RequiredArgsConstructor
public class ResultGradeDecorator {
    8 usages
    private final StudentExamResult result;
    5 usages
    private AssembledAnswerDTO assembledAnswerDTO; // forbid other class to read/write this DTO

    // dto should only be used here
    1 usage
    public void calculateScore() {
        assembledAnswerDTO.getAssembledAnswerList().forEach(assembledAnswer -> {
            Strategy strategy = StrategyFactory.create(assembledAnswer.getQuestionPOJO().getQuestionType());
            var score :double = strategy.grade(
                    assembledAnswer.getAnswer().getAnswer(),
                    assembledAnswer.getQuestionPOJO().getRefAnswer(),
                    assembledAnswer.getAnswer().getMark());
            assembledAnswer.setScore(score);
            log.info("score {} examId {} studentId {} questionId {}",
                    score, assembledAnswerDTO.getExamId(), assembledAnswerDTO.getStudentId(),
                    assembledAnswer.getQuestionPOJO().getQuestionId());
        });
        result.setScore(assembledAnswerDTO.getAssembledAnswerList().stream()
                .mapToDouble(AssembledAnswer::getScore).sum());
    }

    1 usage
    public void setAssembledAnswerDTO(AssembledAnswerDTO assembledAnswerDTO) {
        this.assembledAnswerDTO = assembledAnswerDTO;
    }
```

We use the ResultGradeDecorator to decorate the StudentExamResult class to make it ready to calculate the score.

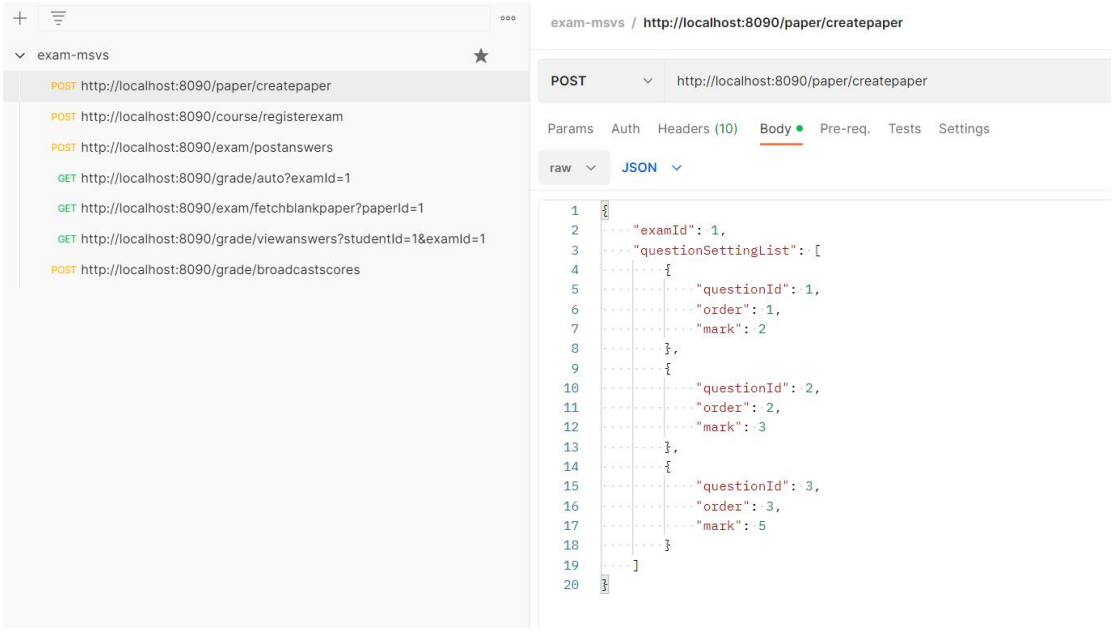
```java
1 usage
public class CalculateScoreCommand implements GradeCommand {
    @Override
    public void execute(long examId, List<ResultGradeDecorator> resultDecoratorList) throws Exception {
        resultDecoratorList.forEach(ResultGradeDecorator::calculateScore);
    }
}
```

When using this decorator, just traverse the decorator list and call calculateScore on each of them.

8.9 Automated testing

We use JUnit5 to test all the REST-apis. And we also use the tool application Postman to test the endpoints.

```java
@Slf4j
@SpringBootTest
@AutoConfigureMockMvc
public class ControllerTests {
    4 usages
    @Autowired
    private MockMvc mockMvc;

    @Test
    public void testRegisterExam() throws Exception {
        String url = "http://localhost:8090/course/registerexam";
        String content = "{\"examId\":1,\"studentIdList\":[1,2]}";
        mockMvc.perform(post(url).content(content)
                .header(HttpHeaders.CONTENT_TYPE, ...values: "application/json"))
                .andDo(print()).andExpect(status().isOk());
    }
    @Test
    public void testPostAnsweredPaper() throws Exception {
        String url = "http://localhost:8090/exam/postanswers";
        String content = "{\"studentId\":1,\"examId\":1,\"answerList\":[{\"questionId\":1,\"order\":1,\"mark\":5,\"answer\":\"B\"},{\"questionId\":2,\"ord
        mockMvc.perform(post(url).content(content)
                .header(HttpHeaders.CONTENT_TYPE, ...values: "application/json"))
                .andDo(print()).andExpect(status().isOk());
    }
    1 usage
    @Test
    public void testCreatePaper() throws Exception {
        String url = "http://localhost:8090/paper/createpaper";
        String content = "{\"examId\":1,\"questionSettingList\":[{\"questionId\":1,\"order\":1,\"mark\":2},{\"questionId\":2,\"order\":2,\"mark\":3},{\"qu
        mockMvc.perform(post(url).content(content)
                .header(HttpHeaders.CONTENT_TYPE, ...values: "application/json"))
                .andDo(print()).andExpect(status().isOk());
    }
    @Test
    public void testFetchBlankpaper() throws Exception {
        testCreatePaper();
        String expectedResult = "{\"blank_question_list\":[{\"decorator\":{\"id\":1,\"refAnswer\":\"B\",\"tags\":\"golang\",\"questionType\":\"SINGLE\",\"
        mockMvc.perform(MockMvcRequestBuilders
                        .get( urlTemplate: "http://localhost:8090/exam/fetchblankpaper?paperId={paperId}", ...uriVariables: 1)
                        .accept(MediaType.APPLICATION_JSON))
                .andDo(print())
                .andExpect(status().isOk());
    }
}
```



9. Added Value

9.1 RabbitMQ and Spring AMQP

We use RabbitMQ and Spring AMQP to implement an event-driven architecture for microservices, decoupling them and increasing their scalability.
RabbitMQ is a messaging broker - an intermediary for messaging. It gives applications a common platform to send and receive messages, and provide the messages a safe place to live until received.
Advanced Message Queuing Protocol (AMQP) is a wire-level protocol that defines the data format of messages as a stream of bytes. RabbitMQ supports the AMQP and other protocols.
The Spring AMQP can be considered as an adapter of RabbitMQ. The Spring AMQP project applies core Spring concepts to the development of AMQP-based messaging solutions. It provides a "template" as a high-level abstraction for sending and receiving messages. It also provides support for Message-driven POJOs with a "listener container".

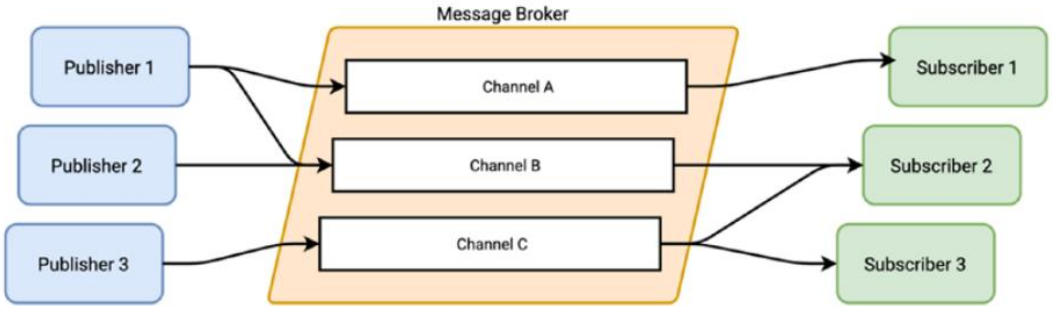


Figure. High level view of the message broker

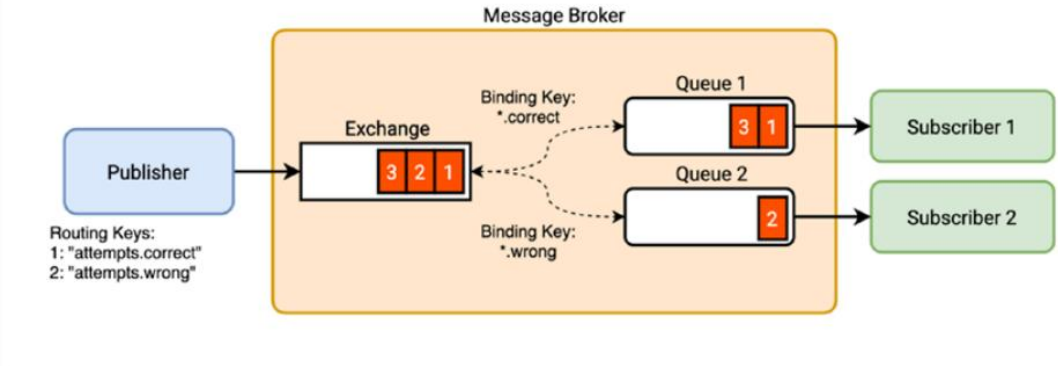
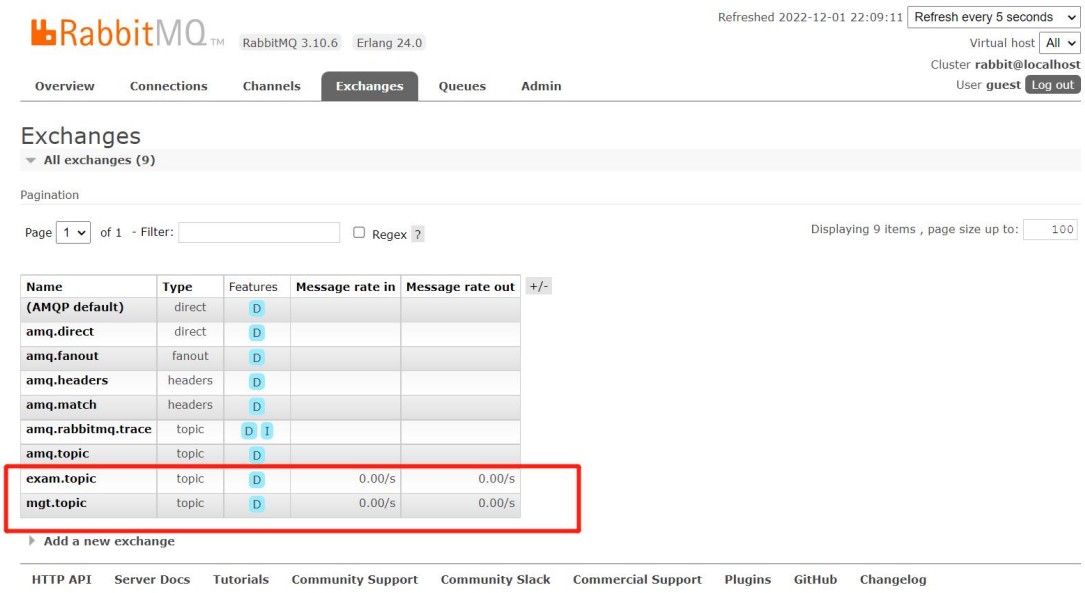
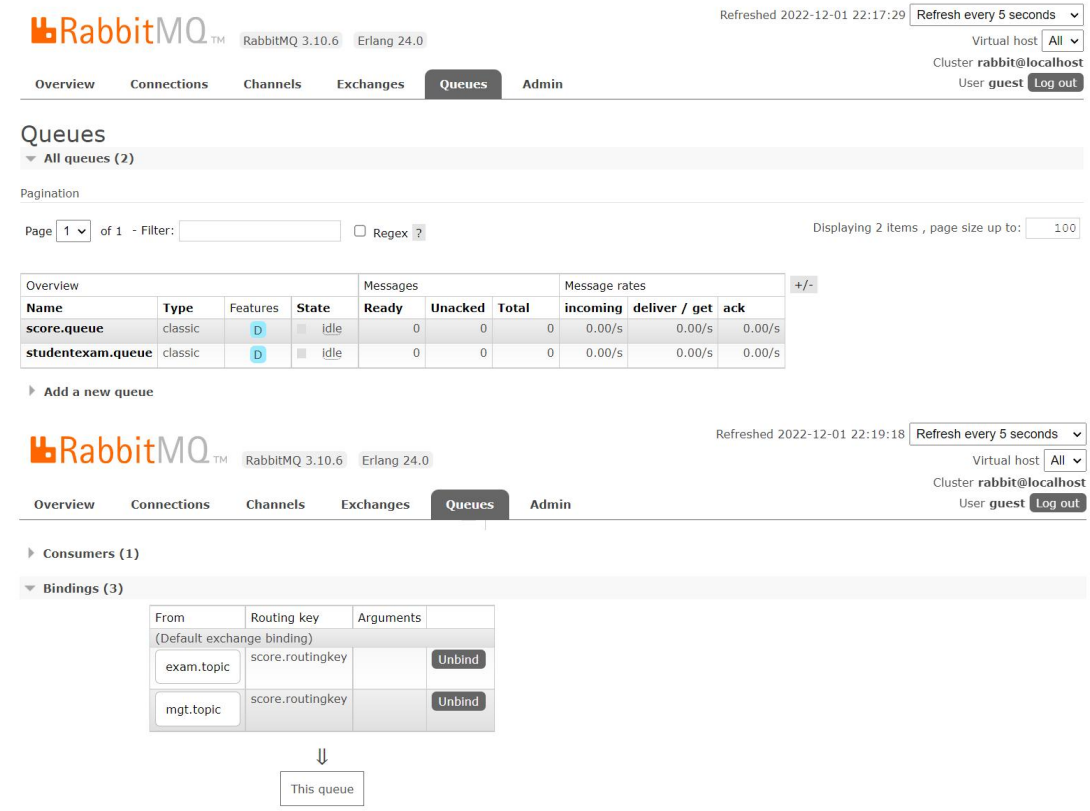


Figure. Basic concepts of the RabbitMQ Message broker. Notice the main concepts of Publisher, Subscriber, Queue, Exchange, Binding Key. They play as the main blocks of building a even-driven system. A Publisher has an Exchange for sending event. A Subscriber(Listener) has a queue for receiving event. The Binding Key is for routing to make sure the specific queue receives the specific event.



This is the RabbitMQ management plugin UI. After the system is started, the two exchanges exam.topic and mgt.topic are built.

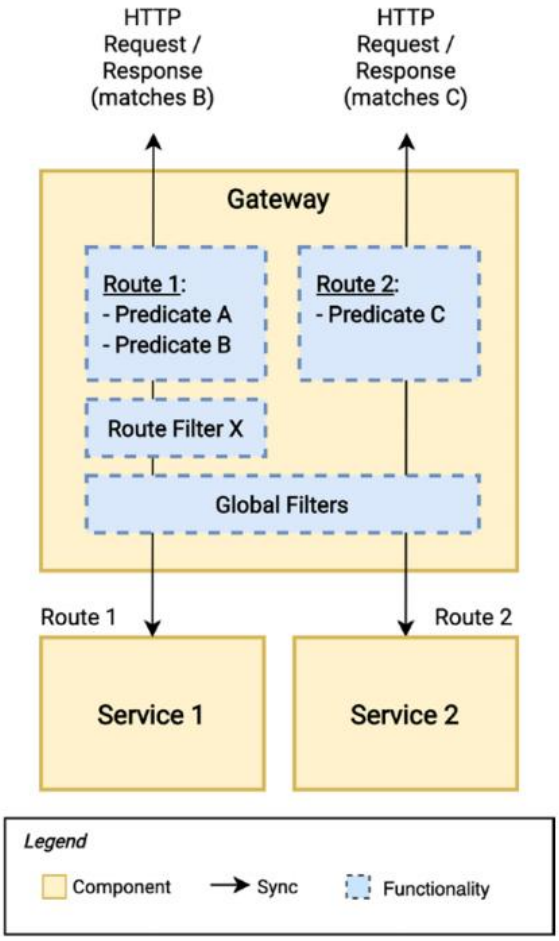


In the Queues label view, we can see the queues are build and binding to the exchange with the routing key.

9.2 Spring Cloud Gateway and Micorservices

We implement the application by microservices(refer to section 10.1 for detailed description of the architecture). But the the front-end needs to point to multiple back-end microservices

to interact with their APIs. This is wrong because the front-end should treat the back end as a single server with multiple APIs. Then, we don't expose our architecture, thus making it more flexible in case we want to make changes in the future.
To solve these problems, we add a Gateway microservice in our system. The gateway pattern centralizes the HTTP access and takes care of proxying requests to other underlying services. Usually, the gateway makes the decision for where to route a request based on some configured rules (a.k.a. predicates).



This is the main concepts of Spring Cloud Gateway: routes, predicates, filters

```yaml
server:
  port: 8090

spring:
  cloud:
    gateway:
      routes:
        - id: Mgtmsv
          uri: http://localhost:8092/
          predicates:
            - Path=/course,/course/**,/role,/role/**,/score,/score/**
        - id: Exammsv
          uri: http://localhost:8091/
          predicates:
            - Path=/exam,/exam/**,/grade,/grade/**,/paper,/paper/**
      globalcors:
        cors-configurations:
          '[/**]':
            allowedOrigins: "http://localhost:3000"
            allowedHeaders:
              - "*"
            allowedMethods:
              - "GET"
              - "POST"
              - "OPTIONS"
# Uncomment the configuration below if you want to enable route matching logs
logging:
  level:
    org.springframework.cloud.gateway.handler.predicate: trace
```

This is the configuration file applicaton.yml where the routes are configured. The system only exposes localhost:8090 to the front-end. And Spring Cloud Gateway will routes the url with a prefix exma/grade/paper to the url localhost:/8091 which is Exam-microservice deployed in. And the other request with a url prefix of course/role/score will be routed to localhost:8092, which is the other microservice deployed in.

```
: Loaded RoutePredicateFactory [CloudFoundryRouteService]
: Netty started on port 8090
: Started GatewayApplication in 1.515 seconds (JVM running for 1.943)
: Pattern "[/course, /course/**, /role, /role/**, /score, /score/**]" does not match against value "/exam/postanswers"
: Pattern "/exam/**" matches against value "/exam/postanswers"
: Pattern "[/course, /course/**, /role, /role/**, /score, /score/**]" does not match against value "/exam/postanswers"
: Pattern "/exam/**" matches against value "/exam/postanswers"
: Pattern "[/course, /course/**, /role, /role/**, /score, /score/**]" does not match against value "/grade/auto"
: Pattern "/grade/**" matches against value "/grade/auto"
: Pattern "[/course, /course/**, /role, /role/**, /score, /score/**]" does not match against value "/grade/viewanswers"
: Pattern "/grade/**" matches against value "/grade/viewanswers"
: Pattern "[/course, /course/**, /role, /role/**, /score, /score/**]" does not match against value "/grade/broadcastscores"
: Pattern "/grade/**" matches against value "/grade/broadcastscores"
```

This is the runtime log showing how Spring Cloud Gateway matches the request with the configured patterns and routes it to different url.

9.3 Refactor

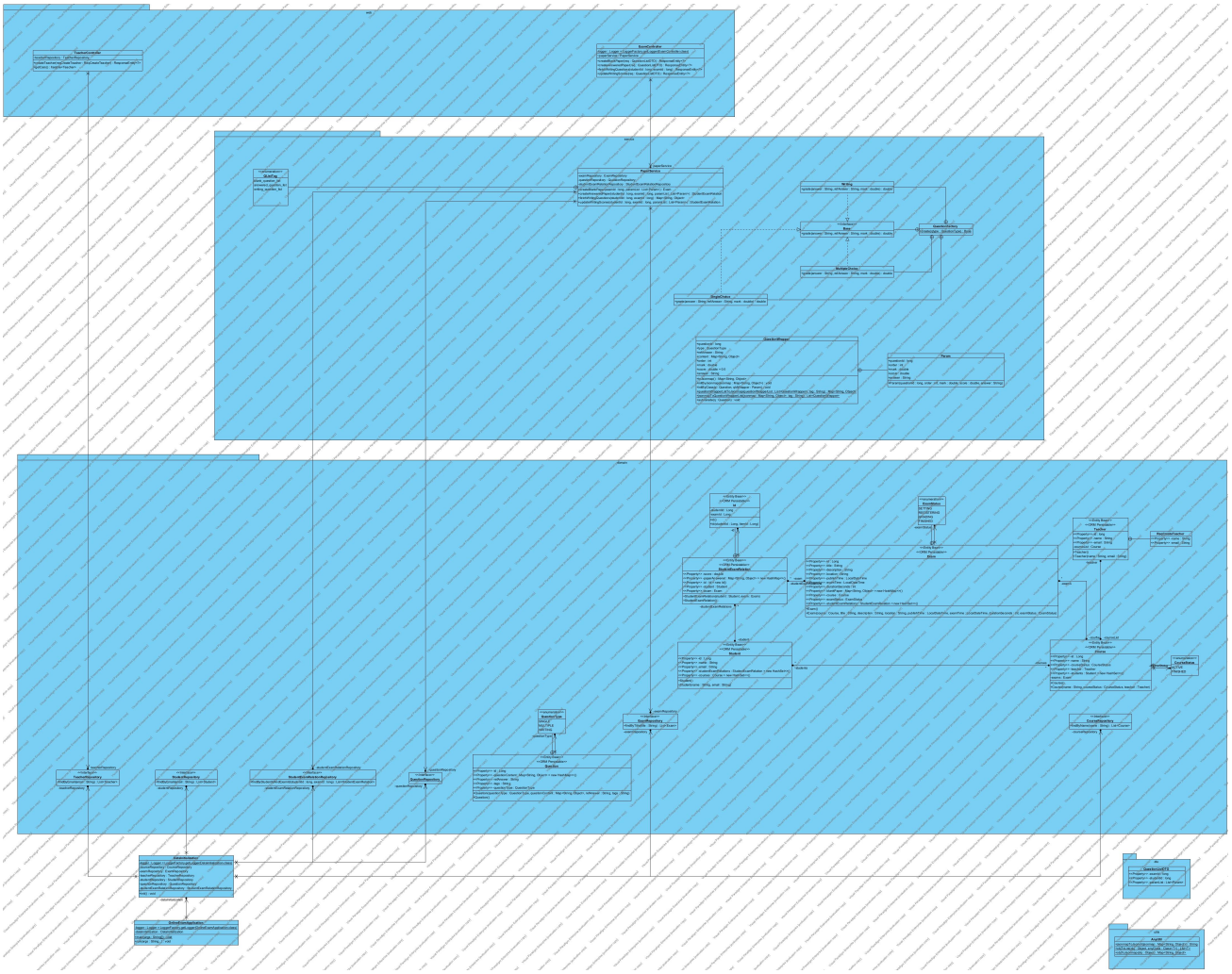
9.3.1 Refactor the monolith to microservices

When we first implemented this project, we used a monolithic architecture and focused on implementing the three most important use cases: creating an exam, taking an exam, and scoring an exam. Later we noticed that the creation and management of teachers, students, courses, and exams could be split into a separate microservice. And creating test papers, taking exams, and scoring exams can be done with another microservice.
With microservices, we can choose more flexible strategies for scalability. In our practical

project, we considered the exam microservice a critical part of our system, having to cope with a big number of concurrent requests. Therefore, we could decide to deploy two or more instances of the exam microservice but only one instance of the mgt(aka management) microservice. If we would keep all the logic in one place, we would replicate also the mgt logic, even though we probably don't need those resources.
So we decided to split the monolithic architecture into a microservice architecture, where they communicate with each other only through messages and do not share REST API nor database. Thanks to the microservice architecture, we can even use NoSQL such as MongoDB in the exam microservice, because the data persisted by this microservice is almost all in json format. However, because PostgreSQL can support json format well and it can be used as a NoSQL database in industrial practice, we keep it.

The steps of split the system:
1. Setting up two microservices, one is exam-microservice, the other is mgt-microservice. then the domain layer in the monolithic application, that is, the database, is split into two parts and put into the two microservices respectively.
2. Split the REST-API and service layer. Because the microservice split is properly divided according to the domain, so it does not destroy a lot of business logic, that we can smoothly put the management logic into the mgt-microservice and the exam logic into the exam-microservice.
3. Refactor the code in each microservice to make the basic unit tests can pass.
4. Introduce message broker RabbitMQ to realize the communication of the two systems.

9.3.1.1 Figure. Comparison of class diagrams before and after refactor

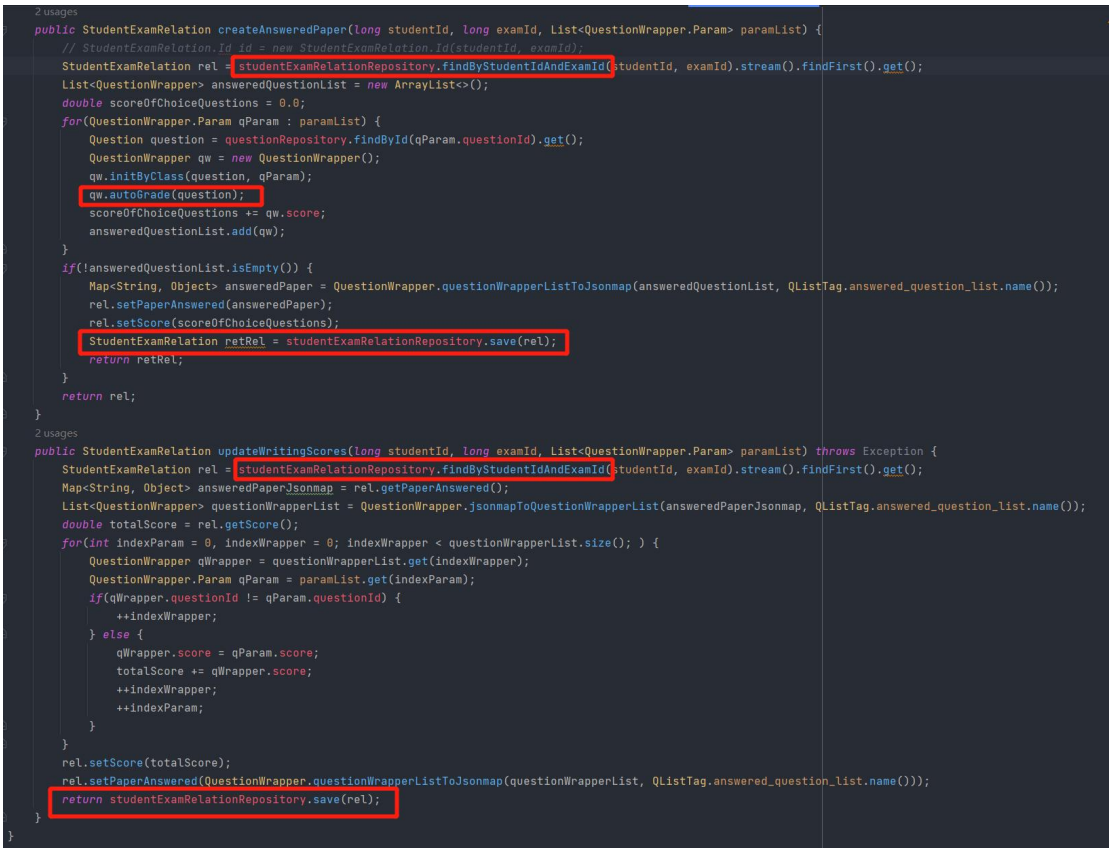


This is the class diagram of the monolith before the refactoring. As you can see, the modules are simply divided according to controller, service and repository. No patterns are used. The code style is more process-oriented, for example, the PaperService class has more than 100 lines methods. It is difficult to maintain and extend.
The class diagram after refactoring, you can check it in section 10.2.1 and section 10.2.2. They are well designed and totally different with the previous design.

9.3.3 Refactor the main business logic to a Command Pattern

The implementation of the use case for scoring exam papers is very complex. This is because of the complex json format to handle and the multiple steps to handle requests from the user, such as manually scoring and adjusting the average score. In the first version of the implementation, we used a process-oriented coding approach, which resulted in a lot of duplicate code, as shown in Figure 9.3.3.1. In the refactoring process, we used the command design pattern. Each fine-grained scoring step is put into a command, and then these commands are put into different arrays and executed in order. To add new scoring logic, just add a new array, as in Figure 9.3.3.2.

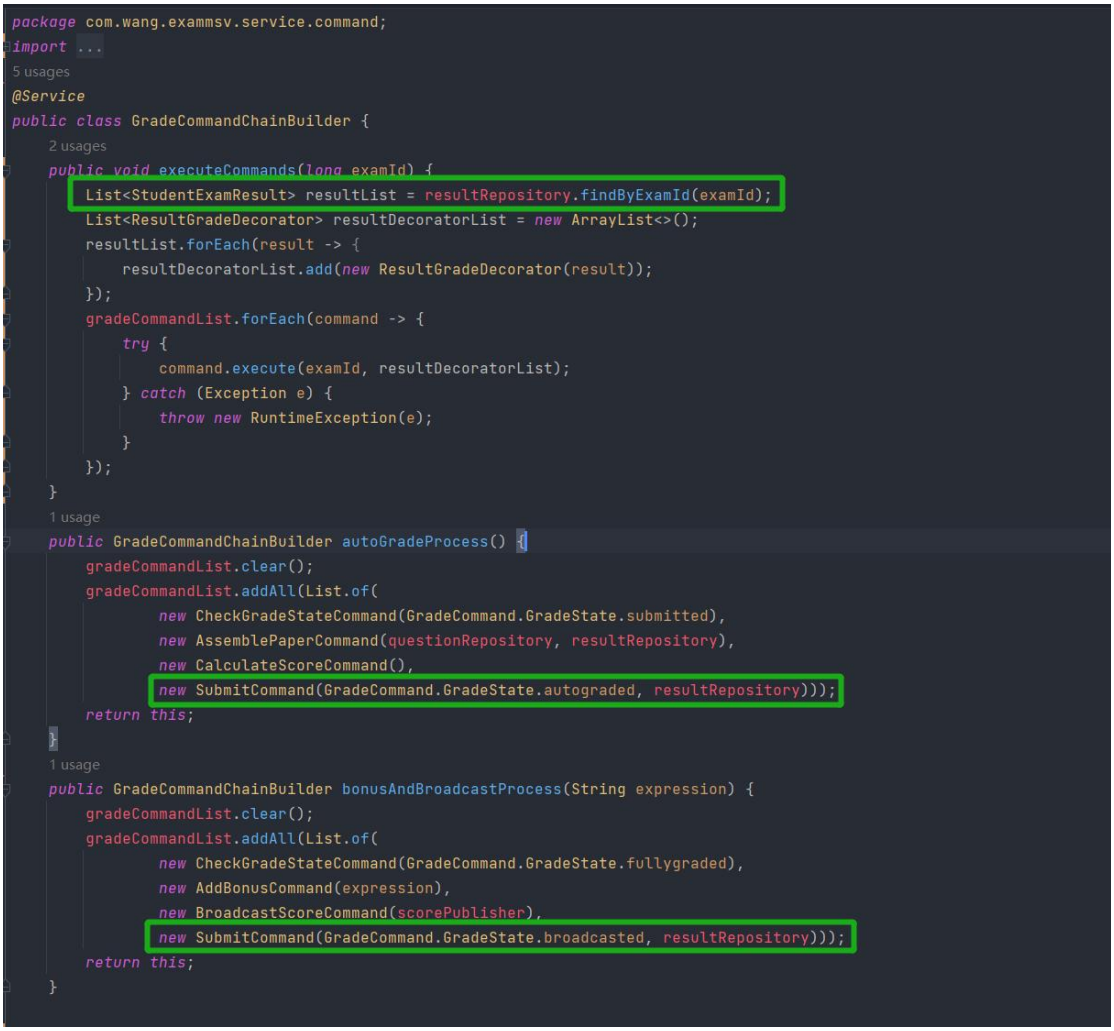
9.3.3.1 Figure. In the first version, bad smell of grading logic code



```
2 usages
public StudentExamRelation createAnsweredPaper(long studentId, long examId, List<QuestionWrapper.Param> paramList) {
    // StudentExamRelation.Id id = new StudentExamRelation.Id(studentId, examId);
    StudentExamRelation rel = studentExamRelationRepository.findByStudentIdAndExamId(studentId, examId).stream().findFirst().get();
    List<QuestionWrapper> answeredQuestionList = new ArrayList<>();
    double scoreOfChoiceQuestions = 0.0;
    for(QuestionWrapper.Param qParam : paramList) {
        Question question = questionRepository.findById(qParam.questionId).get();
        QuestionWrapper qw = new QuestionWrapper();
        qw.initByClass(question, qParam);
        qw.autoGrade(question);
        scoreOfChoiceQuestions += qw.score;
        answeredQuestionList.add(qw);
    }
    if(!answeredQuestionList.isEmpty()) {
        Map<String, Object> answeredPaper = QuestionWrapper.questionWrapperListToJsonmap(answeredQuestionList, QListTag.answered_question_list.name());
        rel.setPaperAnswered(answeredPaper);
        rel.setScore(scoreOfChoiceQuestions);
        StudentExamRelation retRel = studentExamRelationRepository.save(rel);
        return retRel;
    }
    return rel;
}
2 usages
public StudentExamRelation updateWritingScores(long studentId, long examId, List<QuestionWrapper.Param> paramList) throws Exception {
    StudentExamRelation rel = studentExamRelationRepository.findByStudentIdAndExamId(studentId, examId).stream().findFirst().get();
    Map<String, Object> answeredPaperJsonmap = rel.getPaperAnswered();
    List<QuestionWrapper> questionWrapperList = QuestionWrapper.jsonmapToQuestionWrapperList(answeredPaperJsonmap, QListTag.answered_question_list.name());
    double totalScore = rel.getScore();
    for(int indexParam = 0, indexWrapper = 0; indexWrapper < questionWrapperList.size(); ) {
        QuestionWrapper qWrapper = questionWrapperList.get(indexWrapper);
        QuestionWrapper.Param qParam = paramList.get(indexParam);
        if(qWrapper.questionId != qParam.questionId) {
            ++indexWrapper;
        } else {
            qWrapper.score = qParam.score;
            totalScore += qWrapper.score;
            ++indexWrapper;
            ++indexParam;
        }
    }
    rel.setScore(totalScore);
    rel.setPaperAnswered(QuestionWrapper.questionWrapperListToJsonmap(questionWrapperList, QListTag.answered_question_list.name()));
    return studentExamRelationRepository.save(rel);
}
}
```

Red marked the duplicated DB query and saving operations in different but similar handlers of requests. And the method autoGrade should not be included in the business logic of postAnsweredPaper. Also, if a AddBonus function needs to be added into the system, it will be difficult to implement with these bad smell code.

9.3.3.2 Figure. after refactor, the grading logic is well organized into Command design pattern



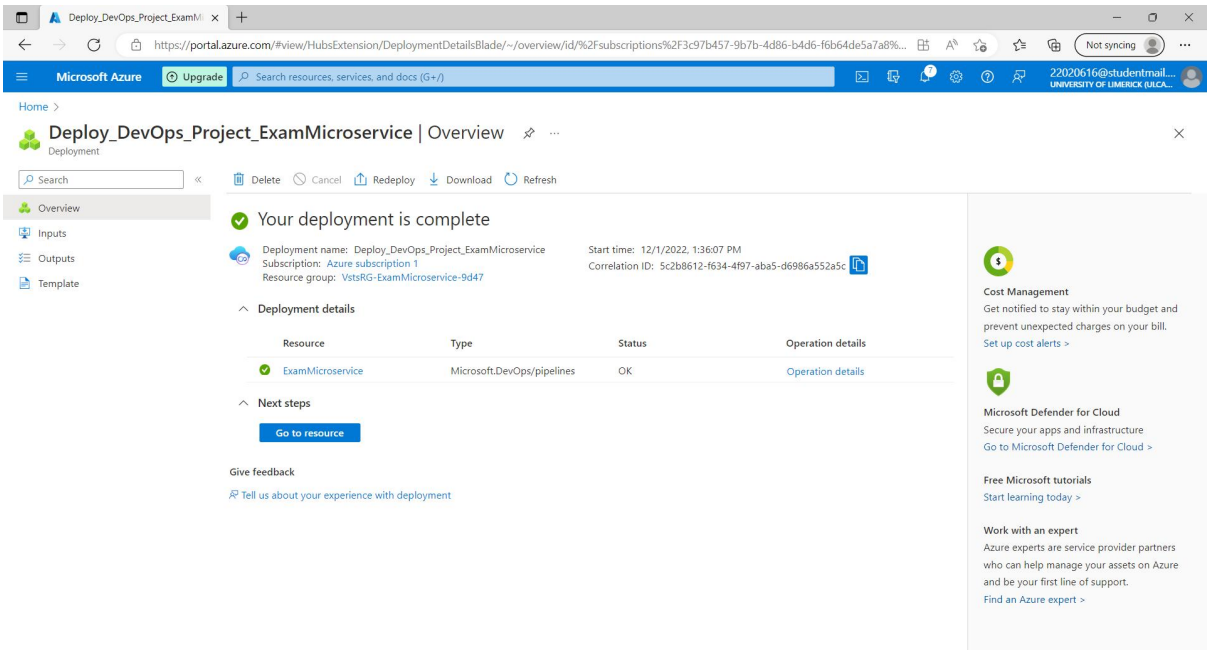
```
package com.wang.exammsv.service.command;
import ...
5 usages
@Service
public class GradeCommandChainBuilder {
    2 usages
    public void executeCommands(long examId) {
        List<StudentExamResult> resultList = resultRepository.findByExamId(examId);
        List<ResultGradeDecorator> resultDecoratorList = new ArrayList<>();
        resultList.forEach(result -> {
            resultDecoratorList.add(new ResultGradeDecorator(result));
        });
        gradeCommandList.forEach(command -> {
            try {
                command.execute(examId, resultDecoratorList);
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        });
    }
    1 usage
    public GradeCommandChainBuilder autoGradeProcess() {
        gradeCommandList.clear();
        gradeCommandList.addAll(List.of(
                new CheckGradeStateCommand(GradeCommand.GradeState.submitted),
                new AssemblePaperCommand(questionRepository, resultRepository),
                new CalculateScoreCommand(),
                new SubmitCommand(GradeCommand.GradeState.autograded, resultRepository)));
        return this;
    }
    1 usage
    public GradeCommandChainBuilder bonusAndBroadcastProcess(String expression) {
        gradeCommandList.clear();
        gradeCommandList.addAll(List.of(
                new CheckGradeStateCommand(GradeCommand.GradeState.fullygraded),
                new AddBonusCommand(expression),
                new BroadcastScoreCommand(scorePublisher),
                new SubmitCommand(GradeCommand.GradeState.broadcasted, resultRepository)));
        return this;
    }
```

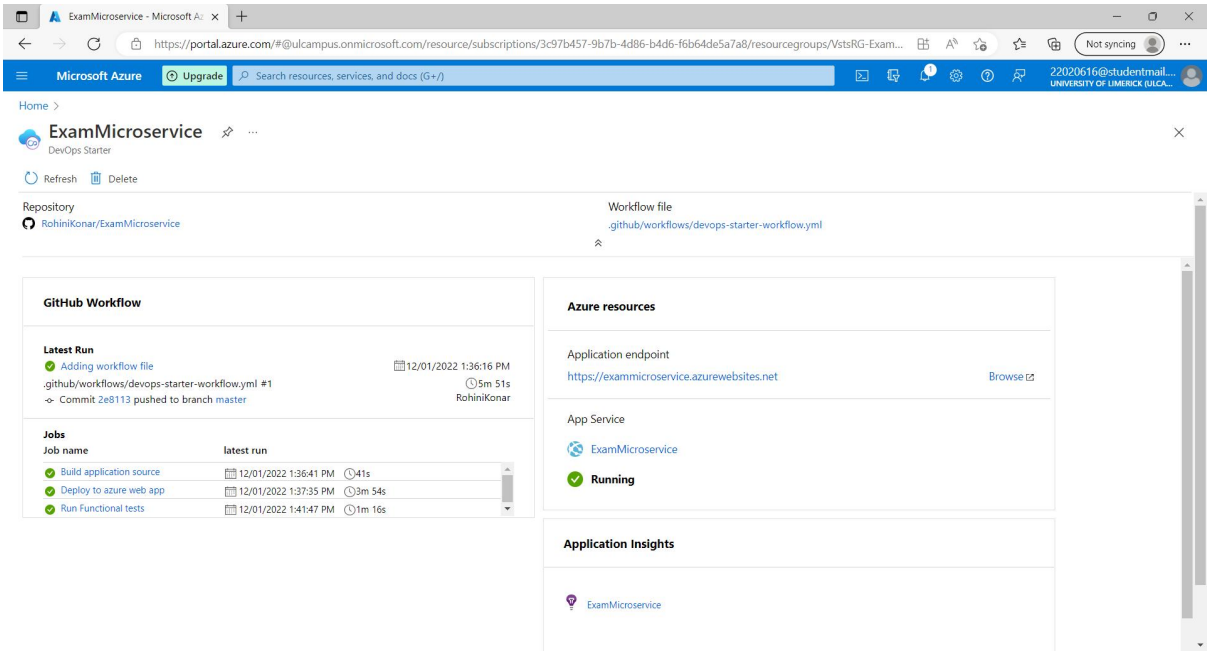
Green marked code is the red one after refactor. The DB query and save operations are only appeared once in the different processes. When a new process like addBonus and broadcast needs to be added, only add a CommandList and that is all. The code becomes much easier to read and modify and less bugs.

9.4 Devops

9.4.1 Azure DevOps Server

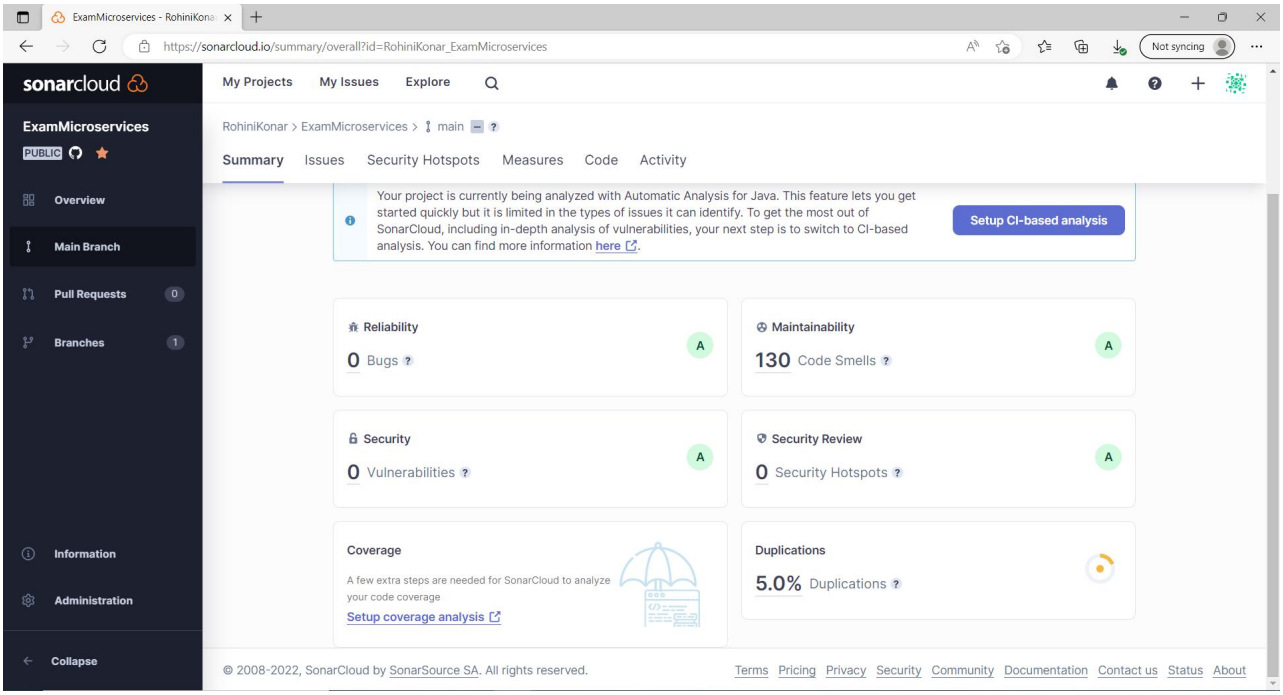
Azure DevOps Server is a Microsoft product that provides version control, reporting, requirements management, project management, automated builds, testing and release management capabilities. In our project we have used it for CI/CD implementation by using DevOps starter service. We have deployed the code using this service.

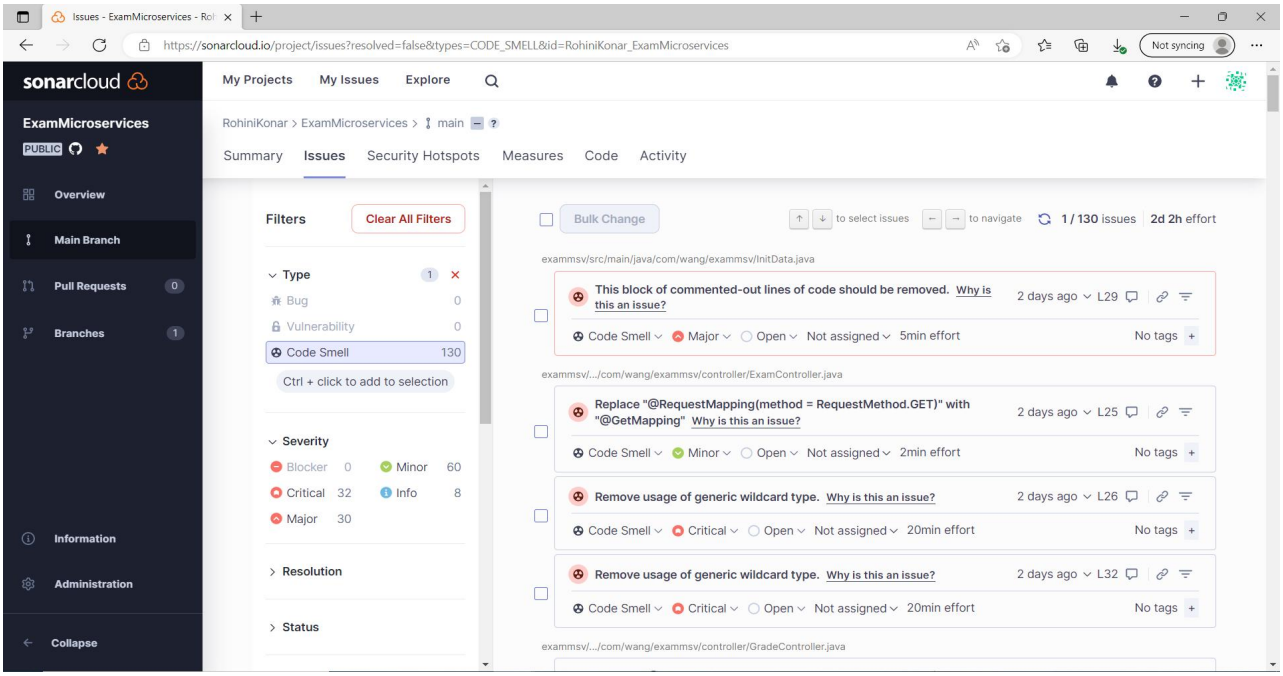




9.4.2 Sonar Cloud

Sonar Cloud is a cloud-based code analysis service designed to detect code quality issues in 25 different programming languages, continuously ensuring the maintainability, reliability, and security of your code. Sonar Cloud analyses the code you and your team write using cutting-edge static code analysis techniques to find issues and potential issues. We used Sonar Cloud to find out the code smells in our code. Below snippet shows us that we had 130 code smells, and it categorizes as well depending on how major, minor, critical or blocker code smell can be.

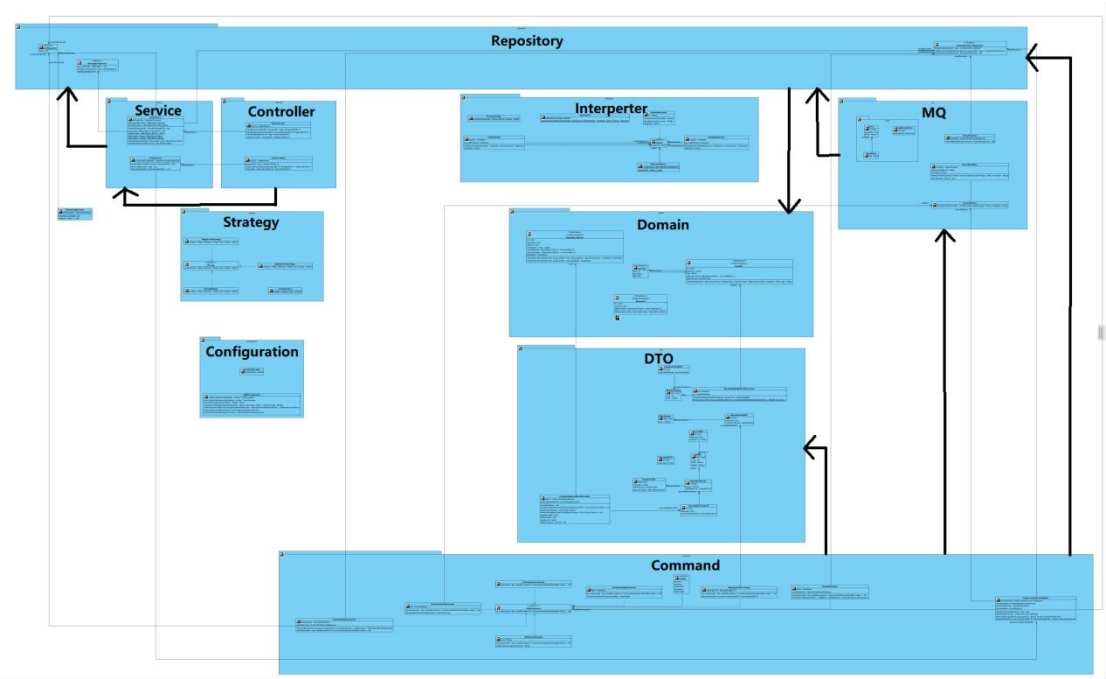

ExamMicroservices - RohiniKonar
https://sonarcloud.io/summary/overall?id=RohiniKonar_ExamMicroservices
Not syncing
sonarcloud
My Projects
My Issues
Explore
ExamMicroservices
PUBLIC
RohiniKonar > ExamMicroservices > main
Summary
Issues
Security Hotspots
Measures
Code
Activity
Overview
Main Branch
Pull Requests 0
Branches 1
Information
Administration
Collapse
Your project is currently being analyzed with Automatic Analysis for Java. This feature lets you get started quickly but it is limited in the types of issues it can identify. To get the most out of SonarCloud, including in-depth analysis of vulnerabilities, your next step is to switch to CI-based analysis. You can find more information here.
Setup CI-based analysis
Reliability
0 Bugs
A
Maintainability
130 Code Smells
A
Security
0 Vulnerabilities
A
Security Review
0 Security Hotspots
A
Coverage
A few extra steps are needed for SonarCloud to analyze your code coverage
Setup coverage analysis
Duplications
5.0% Duplications
© 2008-2022, SonarCloud by SonarSource SA. All rights reserved.
Terms
Pricing
Privacy
Security
Community
Documentation
Contact us
Status
About


Issues - ExamMicroservices - Rohi
https://sonarcloud.io/project/issues?resolved=false&types=CODE_SMELL&id=RohiniKonar_ExamMicroservices
Not syncing
sonarcloud
My Projects
My Issues
Explore
ExamMicroservices
PUBLIC
RohiniKonar > ExamMicroservices > main
Summary
Issues
Security Hotspots
Measures
Code
Activity
Overview
Main Branch
Pull Requests 0
Branches 1
Information
Administration
Collapse
Filters
Clear All Filters
Type 1
Bug 0
Vulnerability 0
Code Smell 130
Ctrl + click to add to selection
Severity
Blocker 0
Minor 60
Critical 32
Info 8
Major 30
Resolution
Status
Bulk Change
to select issues
to navigate
1 / 130 issues
2d 2h effort
exammsv/src/main/java/com/wang/exammsv/InitData.java
This block of commented-out lines of code should be removed. Why is this an issue?
2 days ago L29
Code Smell Major Open Not assigned 5min effort
No tags
exammsv/.../com/wang/exammsv/controller/ExamController.java
Replace "@RequestMapping(method = RequestMethod.GET)" with "@GetMapping" Why is this an issue?
2 days ago L25
Code Smell Minor Open Not assigned 2min effort
No tags
Remove usage of generic wildcard type. Why is this an issue?
2 days ago L26
Code Smell Critical Open Not assigned 20min effort
No tags
Remove usage of generic wildcard type. Why is this an issue?
2 days ago L32
Code Smell Critical Open Not assigned 20min effort
No tags
exammsv/.../com/wang/exammsv/controller/GradeController.java

10. Recovered architecture and design blueprints

All the diagram files can be checked at:
https://github.com/wang-chonghuan/ExamMicroservices/tree/main/workflow

10.1 Design-time package diagram

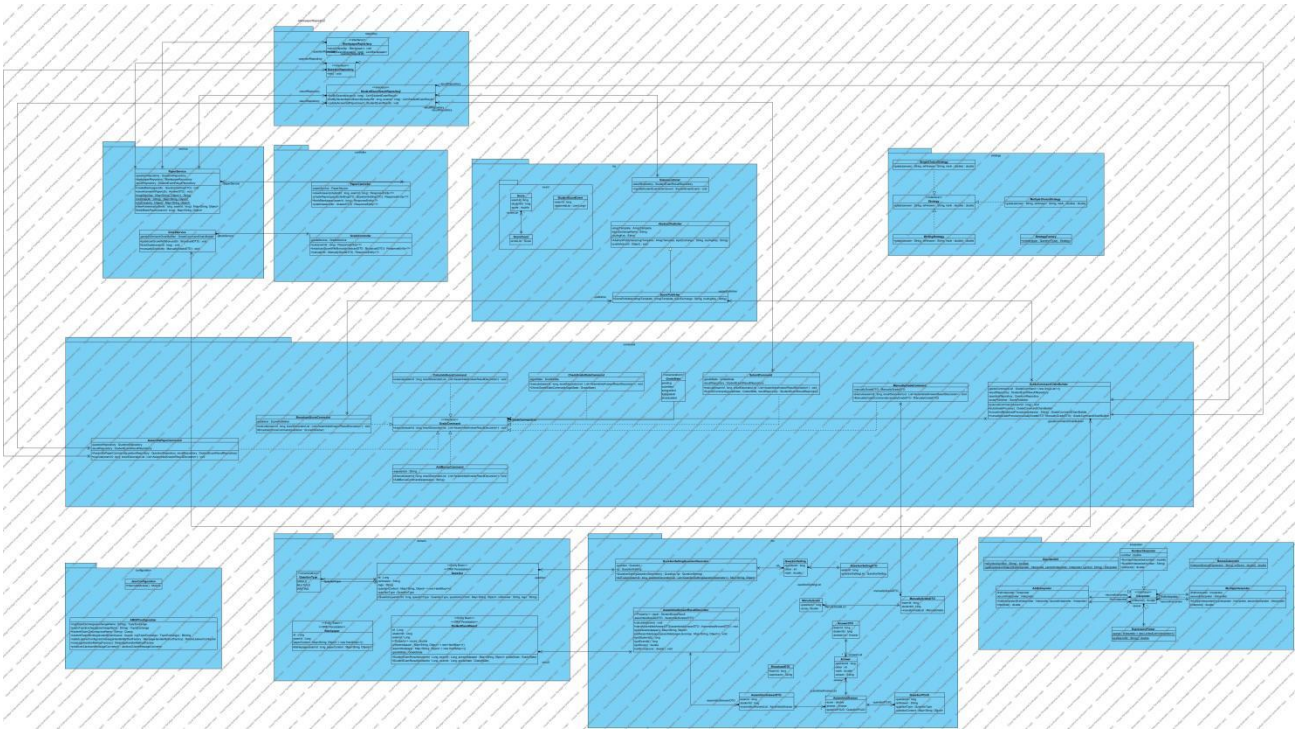


This is the package diagram of Exam-microservice. Arrow indicates dependency, which means a composite relationship. For example, Command depends on MQ(Message Queue), MQ depends on Repository. In our project, the dependencies are very simple and clear, with no confusion. Mgt-microservice's package diagram is not showed here because it is almost the same with this one but much simpler.

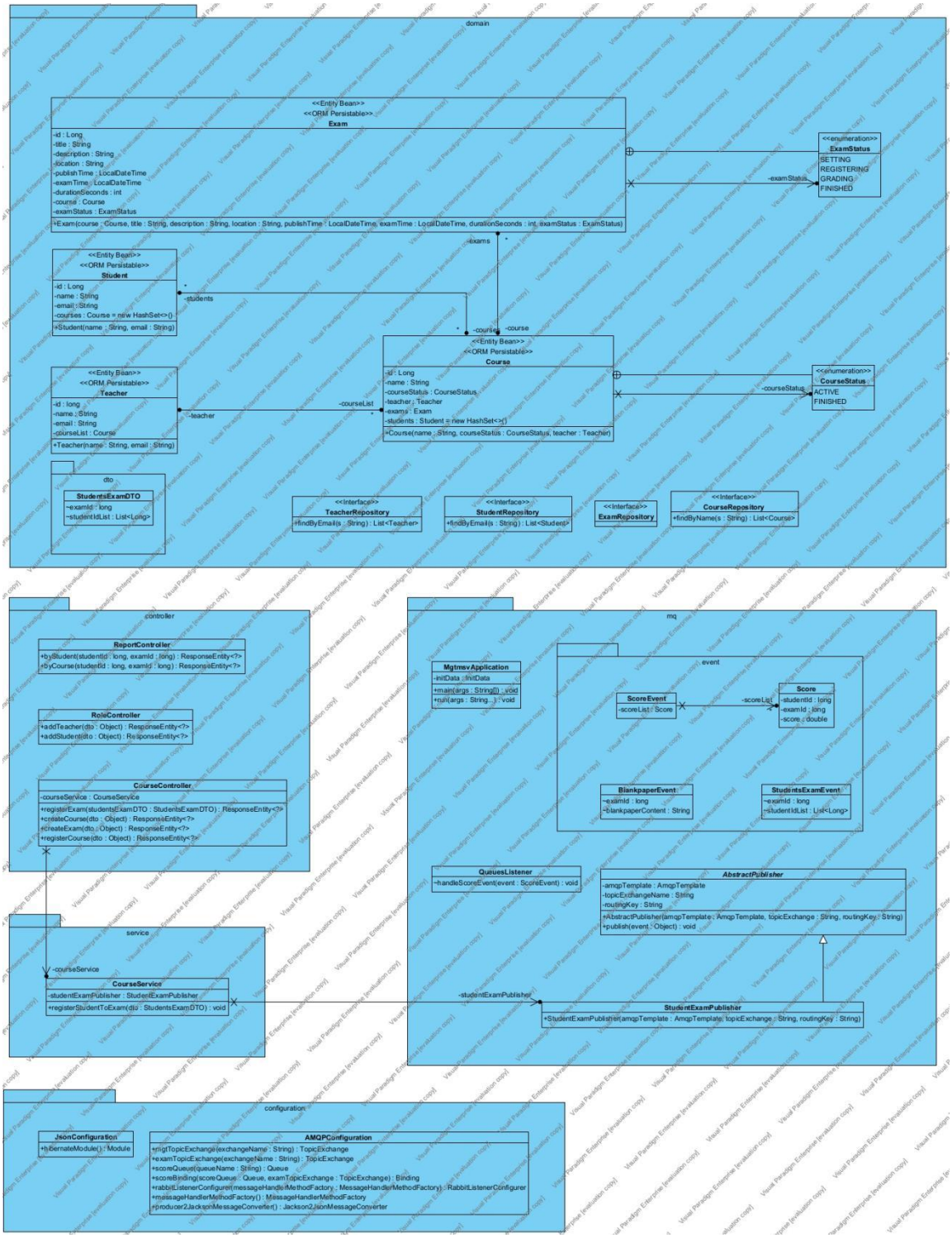
10.2 Design time class diagram

Since we have more than 50 classes in the Exam-microservice, so the class diagram can only be shown and described by packages. And the related classes will be dragged around the target package and marked by white frame to show the relations between them.

10.2.1 Overview of class diagram of Exam-microservice

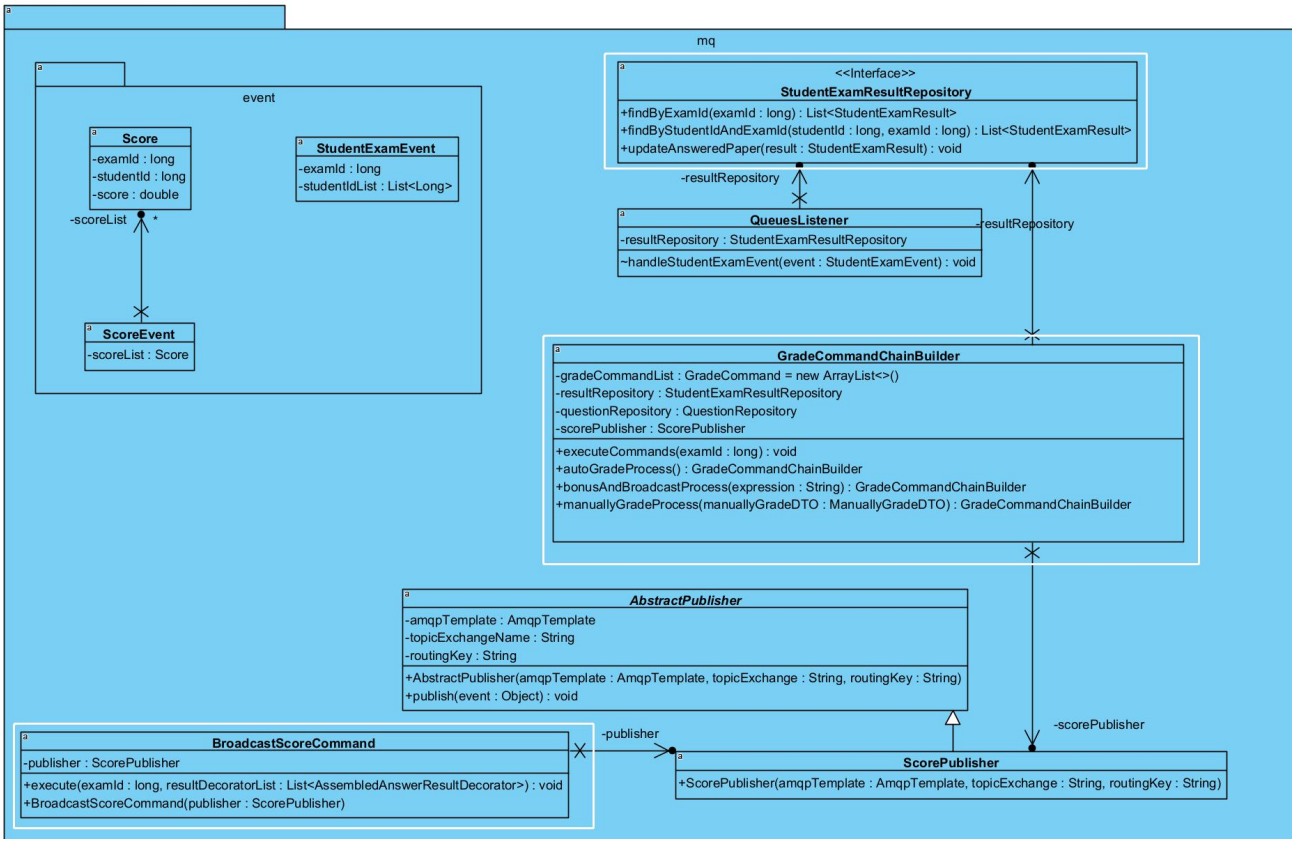


10.2.2 Class diagram of Mgt-microservice



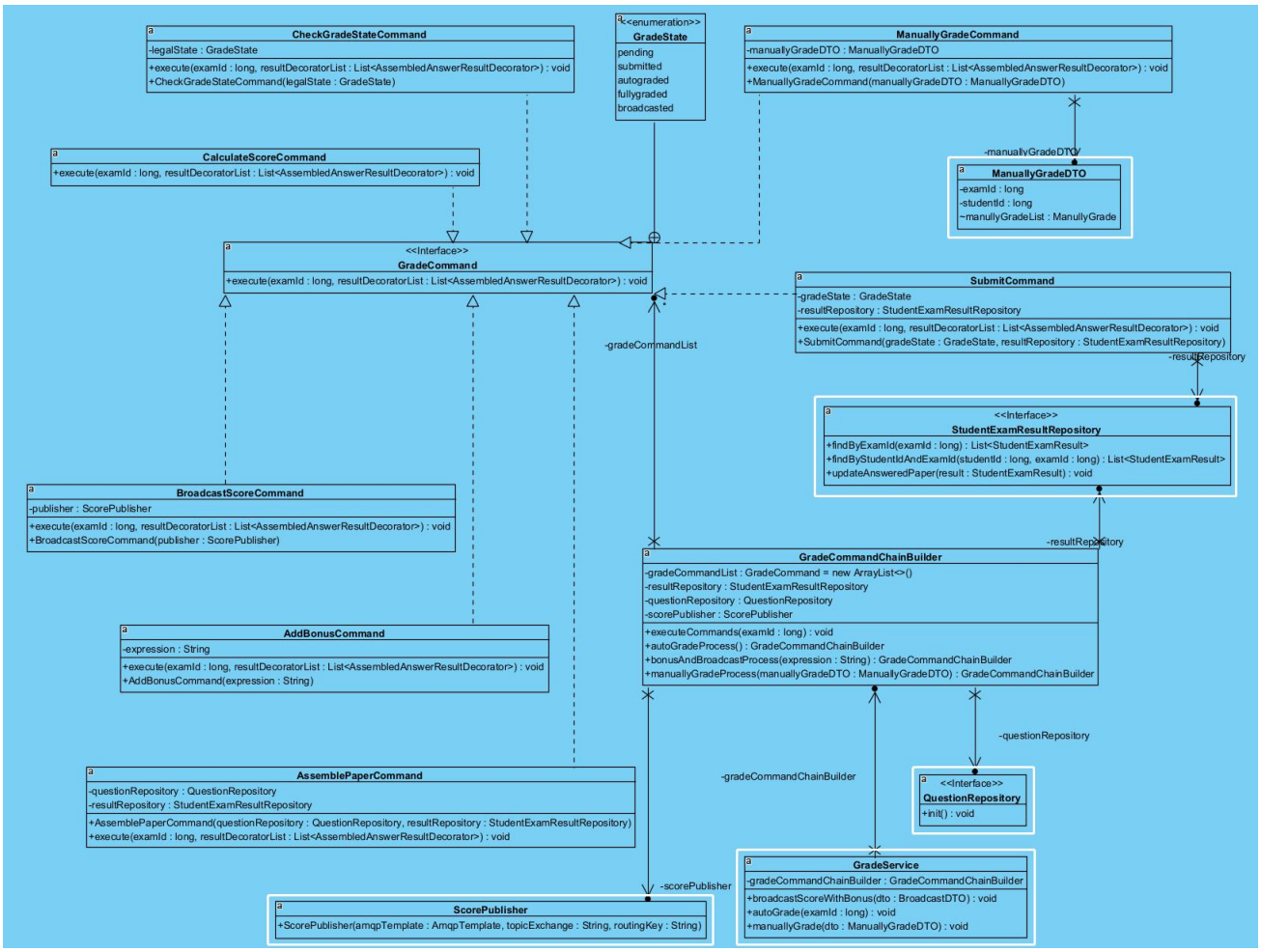
This diagram is similar to the one in Exam-microservice, but much simpler.

10.2.3 Publisher and Listener



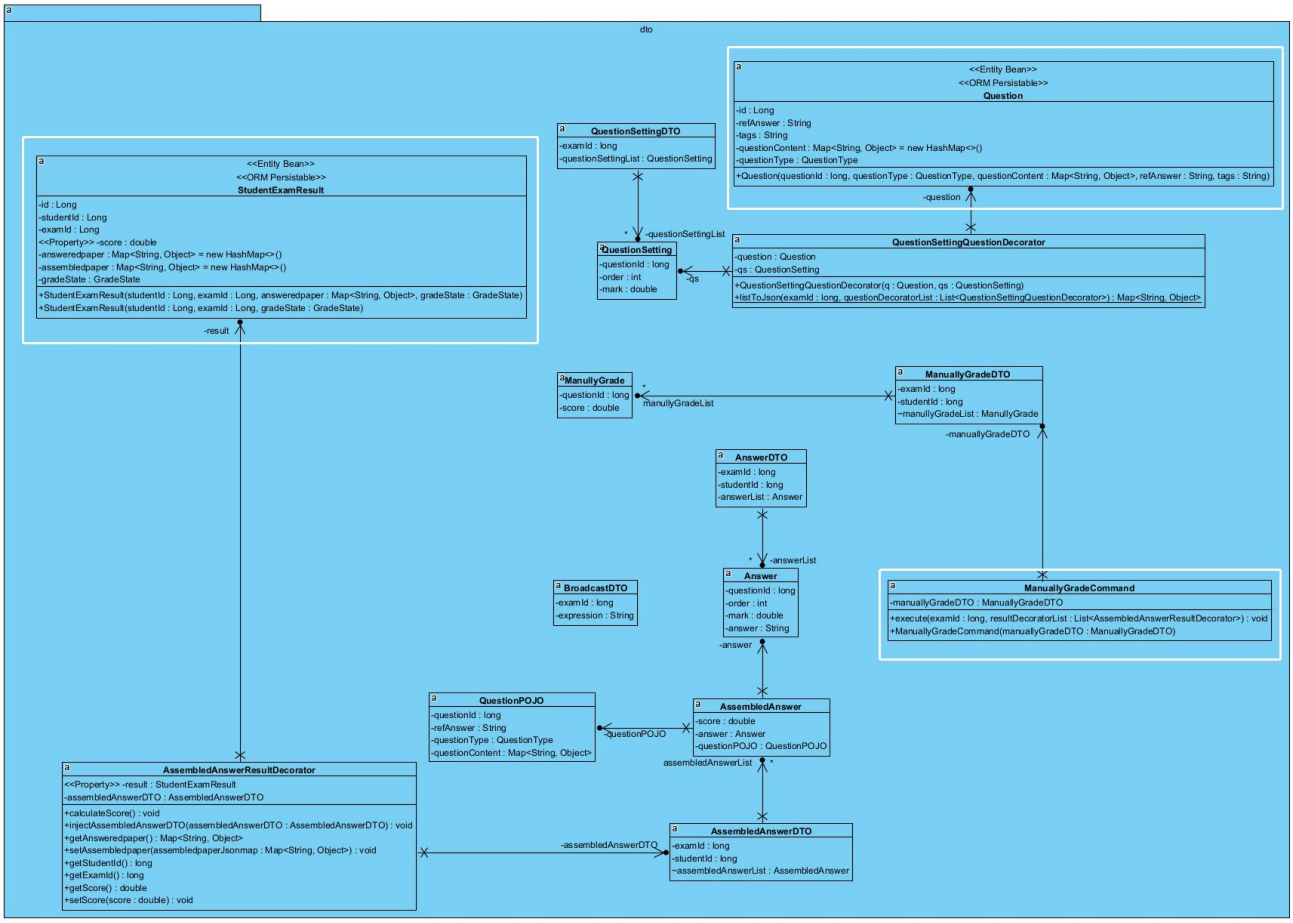
The BroadcastScoreCommand contains a ScorePublisher to publish the scores. QueueListener contains the StudentExamResultRepository to save the students that attended a specific exam to the database. The events ScoreEvent and StudentExamEvent are data structure passed in the message broker.

10.2.4 Command



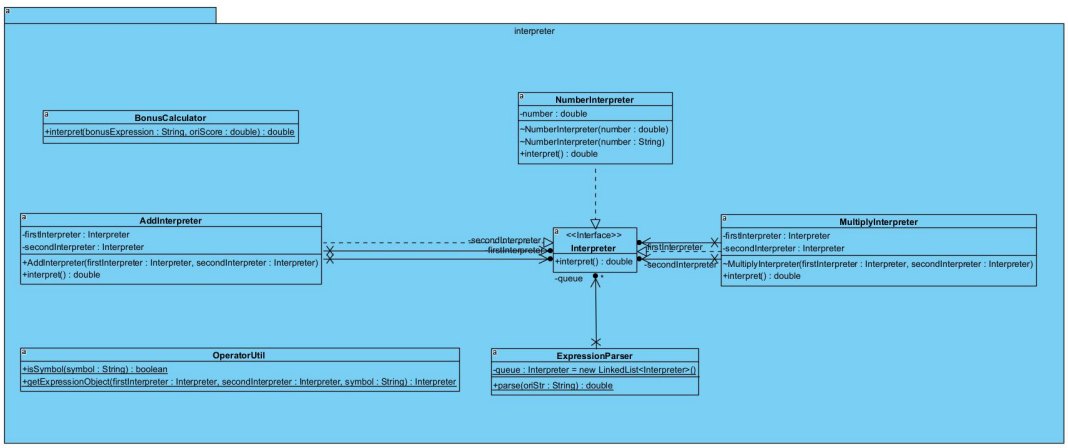
The command package is an implementation of the command pattern. The specific code snippet can be found in Section 8.4. It is the most important package and is responsible for handling the complex scoring process and the grade posting process. Although it has a large responsibility, it is very well implemented with high cohesion and low coupling. It has only GradeService as its user, and it only depends on a few data sources, namely Repository, Publisher and DTO.

10.2.5 DTO

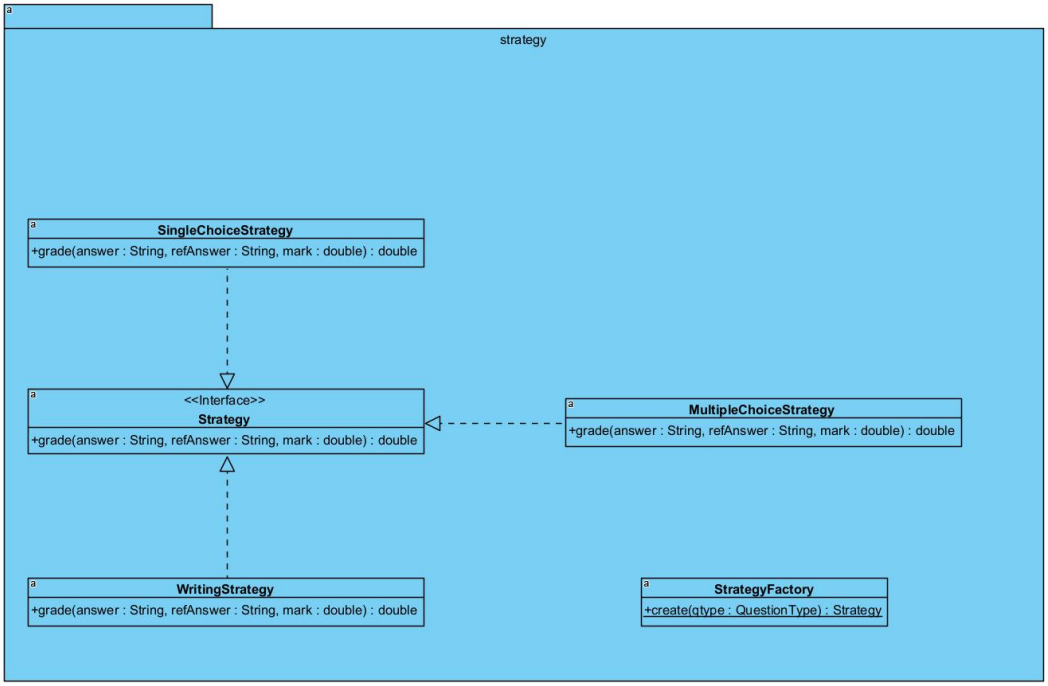


DTO is Data Transition Object, they are some POJO classes responsible for serialization and deserialization of REST-API interface parameters, and data in JSON format from the database. In this package, the decorator pattern is used to add functionality to DTO objects. The classes in the white box are not part of this package, they are referenced to illustrate the relationship between the classes.

10.2.6 Interperter and Strategy

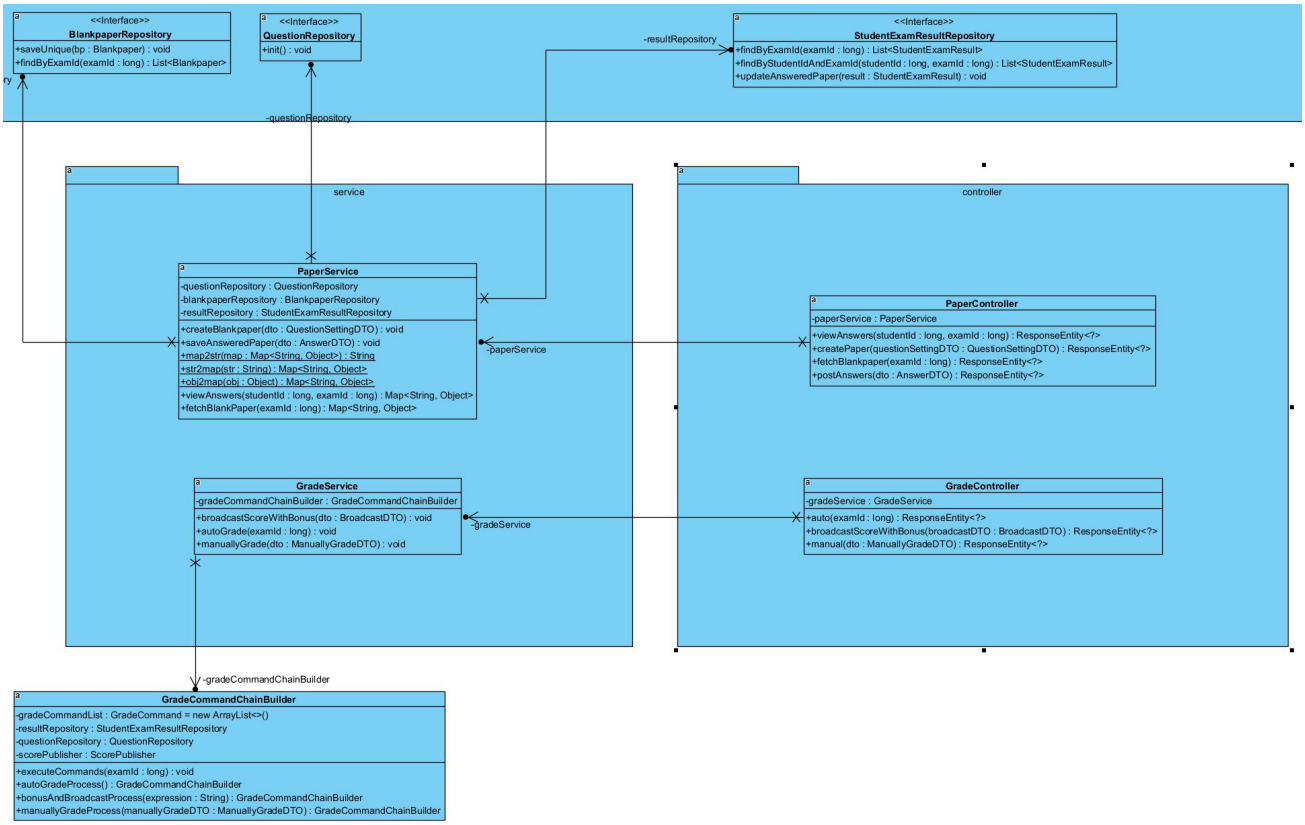


This is the implementation of the interpreter design pattern for interpreter the expression that calculates bonus score. This package has no dependencies because other classes only create it in the method, and none of them hold an instance of any classes in Interpreter package. Refer to section 8.5 to check the detailed description of this diagram.



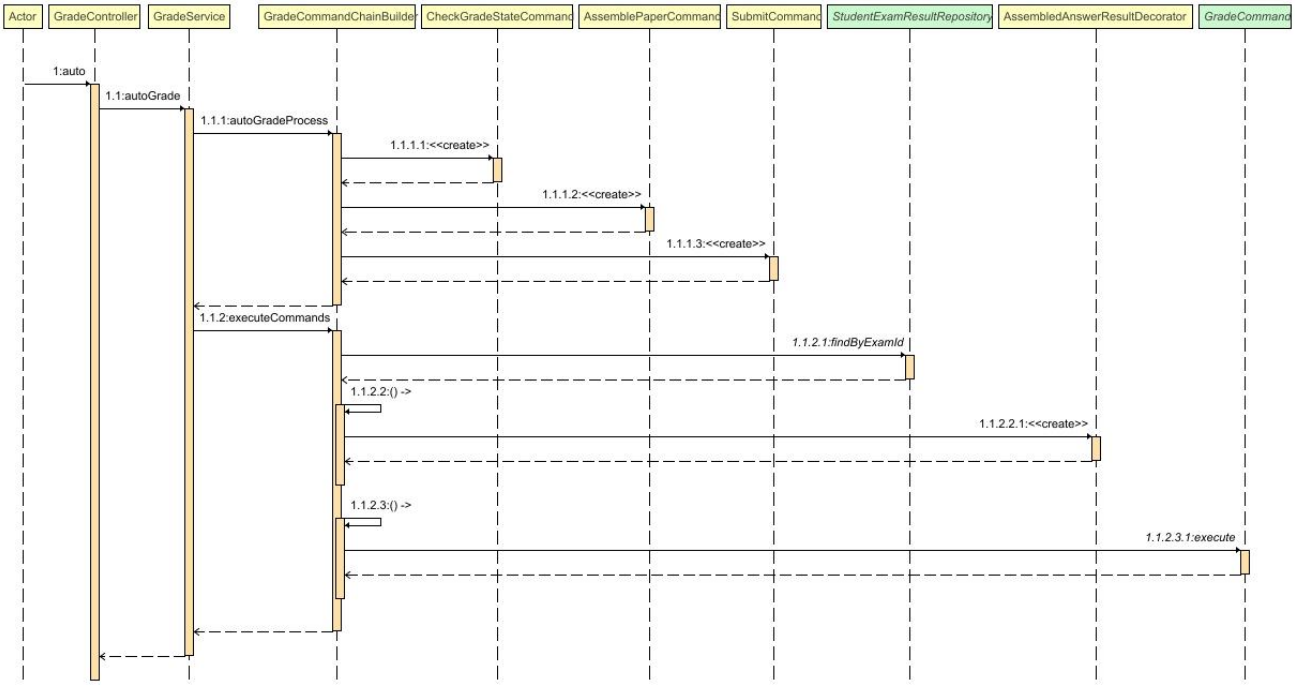
This is the implementation of Strategy pattern for calculating the score of different question types. Refer to section 8.6.

10.2.7 Controller and Service



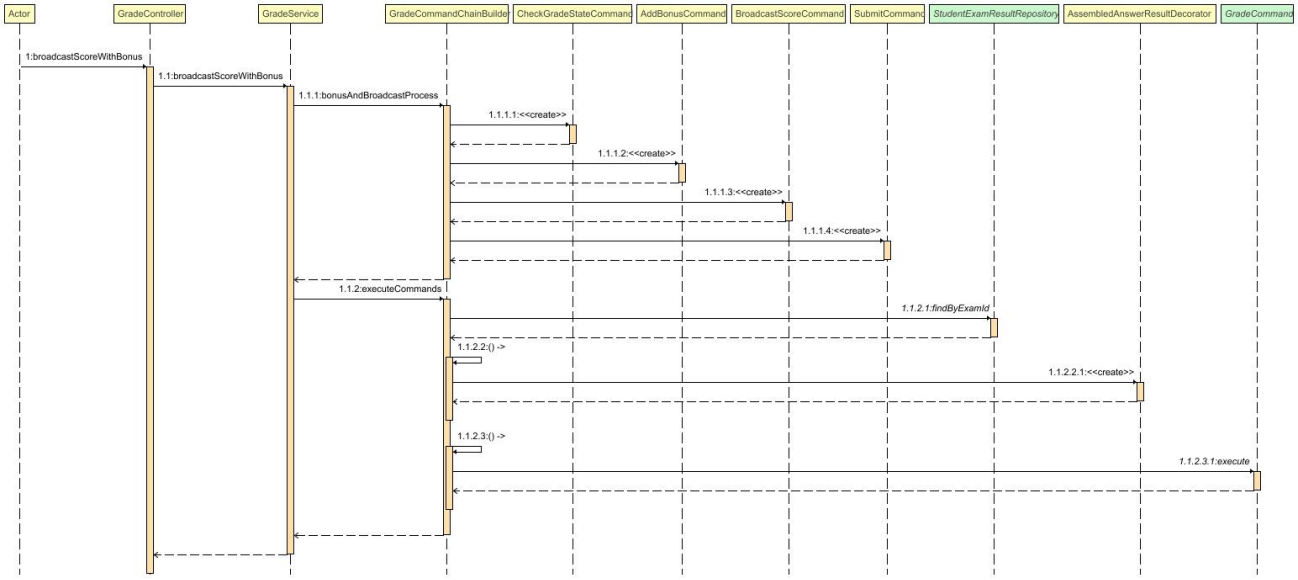
Services contains Repositories and GradeCommandChainBuilder. Controllers only contain Services.

10.3 Design-time sequence diagram



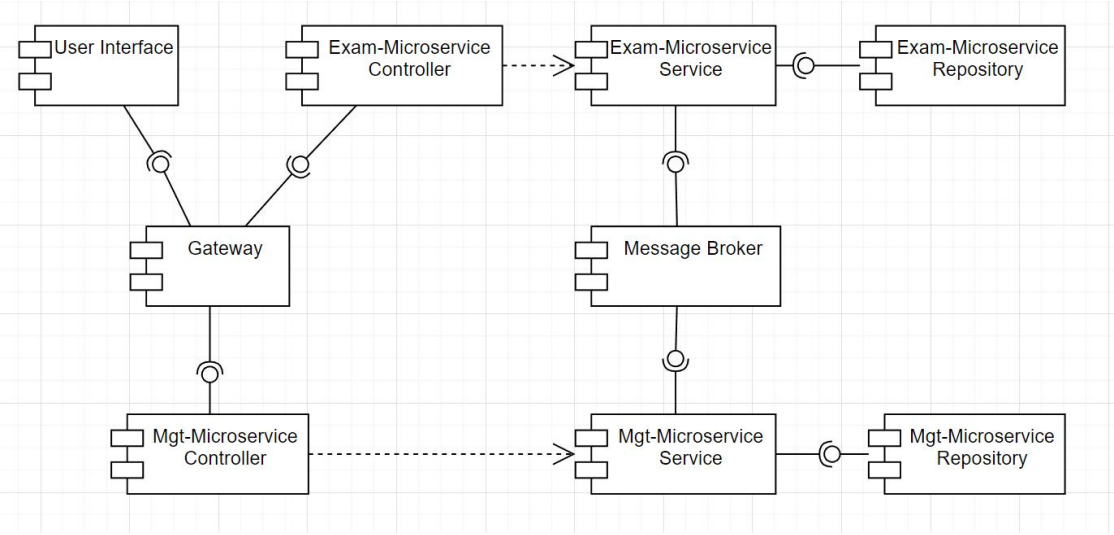
This is the first step of grading exam papers, which is the process of automatically grade all

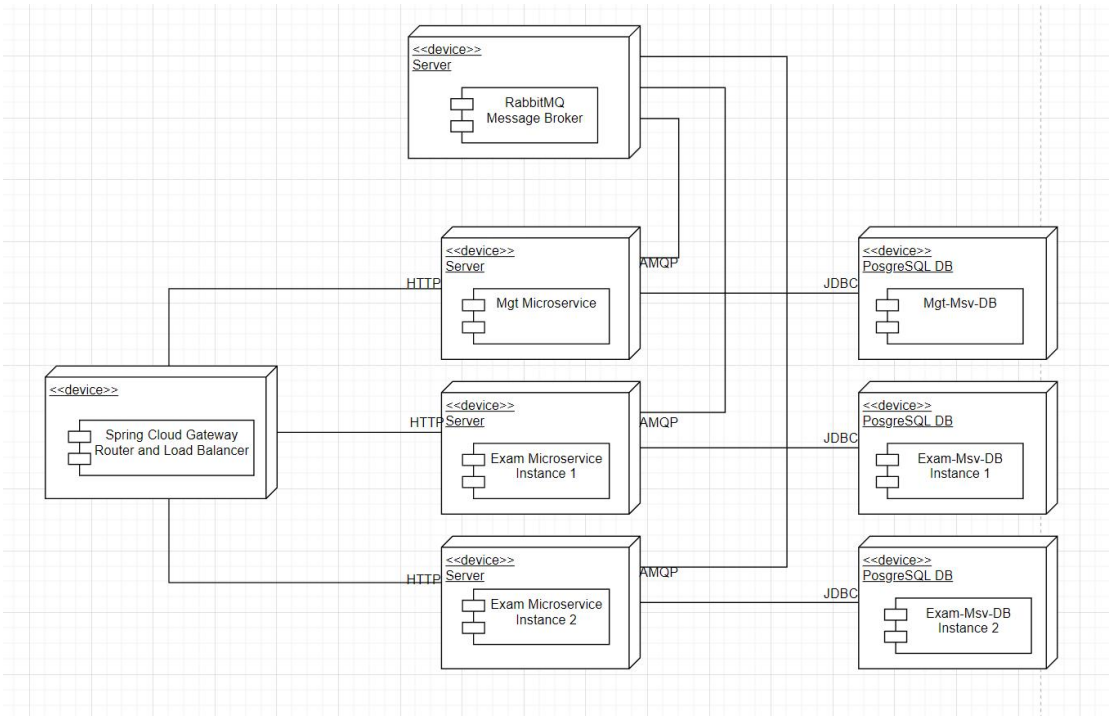
the students' papers of a specific exam. It shows how the command pattern works. 1.1.1 is creating the command chain. 1.1.2 is executing the command chain.



This is the second step of grading exam papers, which is to add bonus to each student's score and broadcast the scores to Mgt-microservice. It's similar to the last one. Compared to the analysis-time sequence diagram, you will find the sequence is more simple and readable, but implement even more complex functionalities.

10.4 Component and deployment diagrams





In our project, we consider the exam-microservice to be a critical part of our system that must handle a large number of concurrent requests. Therefore, we can decide to deploy two instances of the exam-microservice, but only one instance of the mgt-microservice, because mgt-microservice only handles some management requirements by teachers and administrators.

The message broker decouples these microservices from each other. These microservices do not call REST-APIs on each other, have no synchronous requests, and do not have to handle asynchronous requests, and they even don’t share databases.

Spring Cloud Gateway enables the entire microservices to provide only a unified interface and address to the users, hiding the internal implementation and allowing load balancing.

11. Quality of design and implementation

We estimate that 60% of our designs are finally applied to practice. In particular, requirements analysis, business analysis, E-R design, and conceptual class design. They did not show major changes in the subsequent development.
Although the class diagrams did not guide the development precisely, they in turn helped the requirements analysis. In the process of designing class diagrams, our understanding of requirements and use cases, and business scenarios became clearer and clearer, and what we didn't think about during requirements analysis, we figured out during class diagram design.
E-R diagrams can guide practice well. But class diagrams and development are so far apart that we don't think of all the details when we draw them. The design of specific classes can only be clear in development.
I think the main reason is that one we use agile development, the development is incremental, the initial design evolves during the development process. The second is that the tool stack used for development, we are learning for the first time, so in the design is not very familiar with the characteristics of the framework, can not be targeted to design business classes.
But after the implementation of the first version of the code and refactoring, the definition between the classes is very clear. Finally, for the clarity of the class diagram, I refactored the code again to make it more consistent with the object-oriented design principles. During the refactoring process, I kept looking at the generated class diagram, and the connection between classes and packages gradually decreased, indicating that high cohesion and low coupling were achieved. I think this is one of the important things that class diagrams can help us. If I couldn't view the class diagram, I wouldn't have known that there were so many bad flavors in the code.
When reconstructing the monolithic architecture into a microservice architecture, the architecture is slightly more complex because we introduce a message broker. At this point we used package diagrams and architecture design diagrams to help us understand the new architecture. One of the difficulties was which domains to put inside the same microservice. I analyzed the E-R diagram and the class diagram to put classes that have less interaction between them inside a microservice. Without a class diagram, you can't split the business model based on domains, and you can't split the microservices very well. In fact, I didn't split it based on the class diagram at first, and later found that this led to very complicated communication between microservices. Only after referring to class diagrams and E-R diagrams did I achieve such a nice domain division as in our implementation.

12. Reflection

A section entitled “Reflection”. Each team member describes their learning experience through this project. Also reflect on the value of the project. What else did you learn? What worked and what did not work for you?

12.1 Wang

This course has helped me a lot. I learned object-oriented design principles, UML, design patterns. And applied these techniques to this project. I have previous experience in C++/Python/PHP, without any experience of Java web deveoplement. But through this project I force myself to learn Spring Boot, Spring JPA, Spring MVC, JUnit, and initially learned microservices architecture with Spring Cloud, learned message broker RabbitMQ and its application in microservices. For learning these knowledge, I have read over 500 pages of books about Spring framework, which is mentioned in the section 13, and googled hundreds of web pages to learn this knowledge.
With this newly learned knowledge I developed a full-featured web project. The first version of this project had more than 1200 lines and after the midterm I refactored and added it to more than 2500 lines of code. In the process of applying design patterns, I got rid of my previous process-oriented programming habits and refactored the business logic with design patterns, with good results.
These studies took up more than half of my free time in this semester. But I think this class is crucial for my future career. I wish I had more time to learn some devops, front-end and domain-driven design and apply it to projects.

12.2 Rohini

I have 7 years of experience in IT industry, and I have been working on Data Testing and Automation Testing. So, I had less exposure to development side of things i.e., Coding. But with this project I was able to visualize the lifecycle of the project and end to end software development. We used spring boot for our project and for this I had to go through a lot of books and videos to understand how this framework works. With the tools that we used it was easy to start but as we progressed with the implementations, we faced a lot of challenges. I can proudly say because of those challenges I was able to gain knowledge. Design pattern was the one that I felt was difficult and interesting to implement. With MVC I was able to understand how the different layers communicate with each other and how important is it to have the architecture correctly build for the application to work.

12.3 Deepika

I have 5.5 years of experience working in the area of UX and UI design. I had never worked with back-end technology before, but I was making an application with HTML and CSS...This project has provided me with a significant chance for growth in addition to being a significant

challenge to overcome. Because I worked on this project, I now know a lot more about a lot of different things, such as Spring Boot, MVC, Restful APIs, and Swagger. My good friends Rohini and Wang, both of whom had experience in the industry, were a tremendous help to me while I was getting my feet wet and learning the ropes. They taught me things. For example, Rohini taught me about Java Script, agile methodology, and a lot more. Wang taught me about how complicated it is to deal with design patterns. Working on this project has provided me with a wealth of knowledge so far.

13. References

[1]. Moises Macero Garcia: Learn Microservices with Spring Boot, Second Edition
[2]. Juha Hinkula: Full Stack Development with Spring Boot and React, Third Edition
[3]. Bennett, S., McRobb, S., and Farmer, R. Object Oriented Systems Analysis and Design, Fourth Edition.
McGraw-Hill. 2010.
[4]. Fowler, M. UML distilled: a Brief Guide to the Standard Object Modelling Language. Addison Wesley. 2004
[5] Beginning Hibernate 6 Java Persistence from Beginner to Pro (Joseph B. Ottinger, Jeff Linwood, Dave Minter)
[6] Java Persistence with Spring Data and Hibernate (MEAP V04) (Cătălin Tudose)
[7] Practical Microservices Architectural Patterns - Event-Based Java Microservices with Spring Boot and Spring Cloud (Binildas Christudas)
[8] Spring Microservices in Action (John Carnell, Illary Huaylupo Sánchez)