Eötvös Loránd University

Faculty of Informatics

Dept. of Programming Languages and Compilers

Software Evaluation and Analysis Platform with Script Execution Engine

Supervisor:
Viktória Zsók, Ph.D
Assistant Lecturer

Author:
Zayar Htet
Computer Science BSc

Budapest, 2024

EÖTVÖS LORÁND UNIVERSITY
FACULTY OF INFORMATICS

Thesis Registration Form

Student's Data:
Student's Name: Htet Zayar
Student's Neptun code: DRCVG2

Course Data:
Student's Major: Computer Science BSc

I have an internal supervisor

Internal Supervisor's Name: *Zsók Viktória*

Supervisor's Home Institution:	***Department of Programming Languages and Compilers***
Address of Supervisor's Home Institution:	***1117, Budapest, Pázmány Péter sétány 1/C.***
Supervisor's Position and Degree:	*Assistant Professor, Ph.D, Lecturer*

Thesis Title: Software Evaluation and Analysis Platform (SEAP) with Script Execution Engine (SEE)

Topic of the Thesis:
(Upon consulting with your supervisor, give a 150-300-word-long synopsis os your planned thesis.)

There are many third-party frameworks regarding surveys and quizzes such as Google Forms, Microsoft Forms, and SurveyMonkey where we can customise any type of question. However, it is very rare to see a framework where we can create the forms/quizzes for the user to submit the source code and automate the execution and analyse the statistics.

Script Execution Engine (SEE) is a program that can execute any kind of script and source codes (including C, C++, C#, Java) submitted by the client and can be performed both at high-level execution and at low-level. SEE is implemented inside the restful web service framework called SEAP (Software Evaluation and Analysis Platform) where users can also develop the plugins to be able to execute different programming languages and different software with the engine (at low-level execution).

Normally, a vanilla execution for the simple script (for example: an execution of a single .java or .cpp file) does not require a plugin to be executable (high-level execution), while various kinds of personalised execution with many different files (for example: the whole Java project) must be processed through plugins (low-level execution).

SEAP comes with both Rest API and the user-friendly web UI and both can be accessible with three types of user accounts such as: admin, owner and ordinary. SEAP supports many options for how the submitted scripts should be executed. Users can implement their own plugins to specify the detailed plan of execution. An owner can create the web page to accept the submission together with the execution plan, and an ordinary user can submit the files on that page.

SEAP Rest API and SEE are independent, standalone software which allows us to integrate them into the existing application. For example, SEAP Rest API can be integrated as a Microsoft Team extension. SEE can also be integrated into Canvas. SEAP Web Ui itself has a use-case for university assignments and mass execution.

Script Execution Engine will be developed with a highly efficient concurrent programming language called GoLang and SEAP Restful web services will be implemented with the Gin Framework of GoLang. GoLang has a higher efficiency regarding concurrency than any other programming language, and it can generate a single binary output regardless of any number of dependencies.

React with TypeScript or Angular framework will be used for the SEAP Webui and MySQL or MongoDB for storing the data. Docker and Kubernetes will also be used to increase the software portability and to generate standalone software with easy deployment.

Contents

Chapter 1

Introduction

Last 20 years ago, it was hard to develop simple software and to execute written programming languages. Nowadays, our civilization has evolved and our lives have become more comfortable. If we analyse how we are living, we will see that most of the daily processes are fully automated, such as automatic closing doors, traffic lights, automatic switch-on switch-off light bulbs, escalators and so on. We barely notice how those processes have evolved from manually doing to fully automated without human effort.

In software engineering, a principle called DRY (Do not repeat yourself) reflects the actual world where people keep doing the same thing again every day. People always try to make new creations or inventions that help to reduce man's effort and make their lives more comfortable.

Nowadays, people have noticed the term "automation" because of the evolution of AI. In previous days, people do not realise what the term "automation" means. It is not today that "automation" has become popular. It is been throughout the human evolution. As of today's automation world, most labour has been replaced with fully automated robots, such as assembly factories, delivery chains, restaurant servers and self-driving vehicles. Humans are the most intelligent animal species in the world, kept inventing and creating many automated processes and today, those processes are combined with AI which will become a powerful civilization. It cannot be imagined how we will live in the next 100 years.

In this thesis, I would like to introduce the Software Execution and Analysis Platform (SEAP), an assessment system that can publish programming homework and assignments and other users can submit them. With just one click, SEAP can fully automate the execution of the submitted files. In SEAP, the user who publishes the assignment can grade each person based on the result evaluated by the engine called Script Execution Engine (SEE).

There are many similar software developed and used in enterprises. Even in ELTE, people have already developed a learning management system called TMS and used it in today's classes and exams[1]. This SEAP has the same approach as existing software but with a new feature called plugin-extendable.

It is tiring to download each file, set up the environment, and write a command to execute meanwhile SEAP can fully automate those processes with just one click. The Script Execution Engine (SEE) supports the plugin development feature and therefore, can automate the execution of any programming language.

In this documentation, there are two big chapters, user documentation that will explain the usage of the SEAP and SEE, and the developer documentation that specifies the entire software development life cycle and the plugin development.

The user documentation will provide clear explanations of important SEAP keywords, define user roles within the system, and guide users through the installation process. It shows the workflow of Rest API server authentication and recommended solutions for the future. The documentation for the Rest API usage is indicated in the standard Rest API documentation format. The user manual for the web interface is also included with the figures and explanation. Troubleshooting tips will help users resolve any issues they encounter. The instructions for setting up the script execution engine, selecting plugins, and executing plugins will be provided with a step-by-step guide.

The developer documentation has four parts, the database, the backend Rest API with Go, the frontend WebUI with Angular and the SEE with Go. It started with the the overall architecture and design patterns of the project together with the software and hardware requirements for the user level. Each microservice's architecture and design pattern are reported in detail and the database's entity-relationship and object relationship mapping are also stated with the figures. The documentation also highlights the development of the SEAP Rest API using the Go and Gin framework with the authentication and authorization setup using middleware. For the frontend, the web interface development using Angular and the authorisation flow with Guard are specified. In the SEE section, the implementation of the Script Execution Engine with the concurrent go routines and channels are illustrated, and the plugin loading, and the process of plugin development using plugin templates based on the plugin standard library are also described with the sample implementation.

The testing includes the unit test for the whole Rest API project and each test package for each layer of the project. The test methods cover the edge cases, success and error scenarios as well.

Let us move forward with the user documentation to know how to use the SEAP and SEE with the user level experience.

Chapter 2

User Documentation

The user manual for the SEAP Rest API server and endpoints, the usage of the SEAP WebUI, the troubleshooting, and the usage of the SEE plugin will be specified in detail in this chapter.

2.1 About SEAP and SEE

In other judging software, like TMS[1] or Biro in ELTE, it allows the user to submit the programming assignments and get the execution result immediately. When we analyse that existing automation process, we can see that the software just executes the submitted files with the pre-defined commands and verifies the result with the pre-defined output. This SEAP also has the same approach but is more customizable for the user how to analyse the result instead of checking if the result of the execution and the pre-defined results are the same or not.

For example: a user creates the assignment, sets the command to execute the files submitted by other users and sets the expected output as "ABC". When we trigger the execution (whether automated execution or not), the software checks if the result is ABC or not. There is no way to customise the verification process. What if we would like to analyse the implementation (through the unit test or context-free grammar), what if the output is correct but is different from "ABC", are we going to develop the new software again with these various scenarios?

However, we need to understand that we cannot create something always perfect for every scenario. We cannot create software that can perform all scenarios mentioned above. What we can do is we can create a framework-like software that allows the user to customise how to assess the execution.

The SEAP software uses the approach called reusable and maintainable, "implement one time, use anytime". The user needs to implement a simple plugin based on the SEE API library to customise the execution and analysis. For example, a user, let us say userA, implemented a plugin, PluginA, that runs the unit test for the submitted files and evaluates the result. PluginA can also be used by other users if they want to perform the same thing instead of redeveloping another plugin. UserB developed a plugin called PluginB, that verifies the execution results with the expected output. PluginB can also be used by other users if they want to evaluate the submitted files in that way. TMS[1] and Biro are already implemented with such kind of PluginB, but no possibility to change the software to PluginA. Even if there is a possibility to change the software to PluginA, there could be a huge effort load without the customizability of the software.

Moreover, what if we want to execute the different programming languages with different compilers? We can just simply change the compilers and commands on how to execute instead of redeveloping the plugins. PluginA and PluginB are just an example, there can be many more plugins based on the possible scenario.

In this project, instead of simple execution done by PluginA and PluginB, we will discuss extraordinary execution used in Clean Programming Language[2] as an example, which is different from normal execution like PluginA and PluginB.

The whole project is built with microservice architecture so SEAP Rest API and SEE. This SEAP came with the SEAP Rest API, which can be used to integrate further into our existing software, for example, canvas or MS Teams extensions.

2.2 Software Evaluation and Analysis Platform and its Features

2.2.1 Terms and Definitions

Definition 1 (Tutor)**.** *Tutor* is a role authorized to schedule, publish, execute and grade the assignments, similar to a Teacher.

Definition 2 (Tutee)**.** *Tutee* is a role that does not have the authority of a *Tutor*, but can submit the assignment and see the grade, similar to a Student.

Definition 3 (Family)**.** A course is known as a *Family* which can be created only by the *Tutor*, similar to Team in Microsoft Teams.

Definition 4 (Member)**.** If a *tutee* or a *tutor* is part of a family, it is called *Member*.

Definition 5 (Duty)**.** An assignment or homework is called a *Duty*.

2.2.2 Roles for Individual

The SEAP is made up of three roles, admin, tutor and tutee. For a normal signup user, only a tutee account will be granted and A tutor account can be granted only by an admin. A tutor can create a family and a tutee cannot do anything except submit the duty. The Figure 2.1 shows the scenario of registering and the account role.

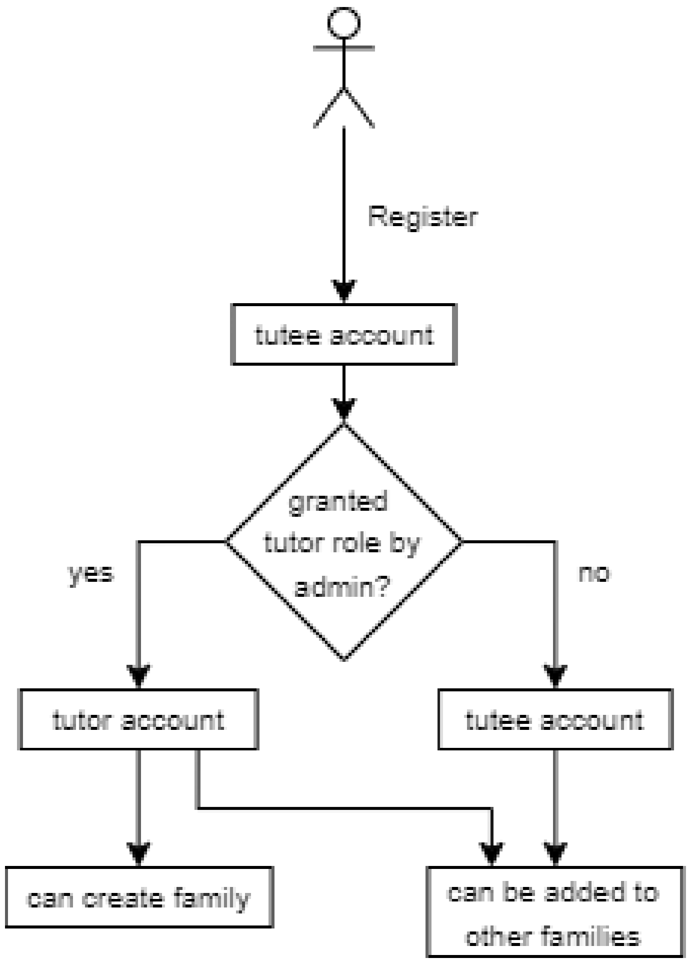


Figure 2.1: Role for individual account

2.2.3 Roles for Family

A family is made up of only two roles, tutor and tutee. When a tutor creates a family, he/she will automatically be granted a tutor role in that family, given control access to the family such as adding members, creating duty and grading duty.

A tutor of a family can add any user to his family to become a family member, upon adding the user, a tutor of a family will be asked to give the new user, a tutor or tutee role.

A tutor of a family has full access to that family and a tutee of a family is accessible only to the duty. All the family data and information are accessible only to the family member.

For example: A userA has a tutor account that allows creating and deleting the family. A userA can be a tutee in another family even though userA has a tutor account. Being a tutor in the family will allow to create, manage, grade and delete the duties in that family. The Figure 2.2 shows the family role explanation of the SEAP.

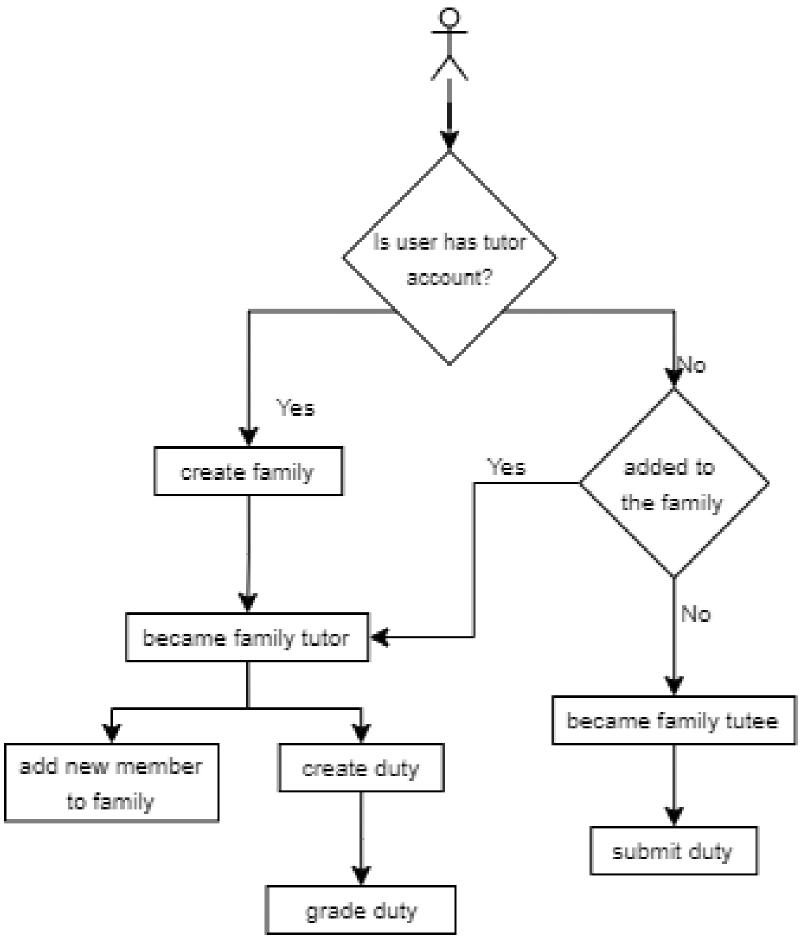


Figure 2.2: Role for family

2.2.4 Features

Assignment Management System

SEAP highlights the assignment management system combined with an auto execution mechanism using SEE. A user who is a tutor of a family can publish the duty, and execute and grade the submission.

Schedule Duty

When creating the duty, the tutor can choose to publish the assignment in the future. If the tutor selects the publishing date earlier than the current date, the duty will be published right away.

Manual Grading System

There are two possible grading systems: point system and pass-fail system. If the tutor chooses the point system, the maximum possible point should be filled. A tutor should grade all the submissions manually by revising the final report generated by SEE.

Late Submission

There are three dates to be filled in when creating the duty, publishing date, due date and close date. A tutee can still submit the duty even after the due date but before the close date. If the tutor wants the duty to be closed right after the due date, the same value should be filled in between the close date and the due date.

Execution and Report

A tutor can execute the plugin to check all the submitted files. A final report will be generated and a tutor should review it and grade the submission.

2.2.5 Installation

As the whole project is a web service, the client can simply use the latest stable version of modern browsers and load the URL where the web app is hosted.

- Google Chrome[3]
- Mozilla Firefox[4]

- Apple Safari[5]
- Microsoft Edge[6]
- Opera[7]

For the Rest API communication, it is always better to use the Postman App which is available as a desktop application for Windows, macOS, and Linux, as well as a web version. SEAP Rest API postman collection file is also included in the project[8].

For local hosting, please refer to the developer documentation for further development.

2.3 SEAP Rest API

SEAP comes with the SEAP Rest API which allows the user to integrate and extend into other popular platforms such as Microsoft Teams, Canvas, etc.

2.3.1 Authentication Model

SEAP Rest API use the simple JWT Token which has just an expired date without any refresh token. Obviously, it is not recommended to use such kind of method for the auth mechanism. As the auth mechanism is the frontline defence against cyber attacks, it should always be perfect without any leakage. The author wrote much more about the recommended auth mechanism called a rotational JWT token.

2.3.2 Payload Model for Rest API

Some POST methods are needed to include the payload to complete the request. The `Content-Type` of the payload can be `application/json` or can be the `multipart/form-data`. All the required payloads are specified and labelled below.

The Code 2.1 is the format of the payload. The `Login` payload is used for the login endpoint (see Table 2.1).

```json
{
    "username": "johndoe",
    "password": "HelloWorld!"
}
```

Code 2.1: Payload.Login

The Code 2.2 is the payload to put in the request body to the Rest API. All the fields are mandatory and can be used to register a new user.

```json
{
    "username": "John Doe",
    "email": "johndoe@john.com",
    "password": "HelloWorld!",
    "firstname": "John",
    "lastname": "Doe"
}
```

Code 2.2: Payload.Register

The Code 2.3 is not a simple JSON. It is a form data together with the file uploading. This form data is used to create a new family by the tutor account.

```
familyName=Thesis&
familyInfo=thesis submission&
family_icon=file
```

Code 2.3: Payload.Family

If the user wants to add a new member to the family, the Code 2.4 must be passed as a payload when sending the request to the new member endpoint.

```json
{
    "familyId": "ff716cbb-501f-471b-b84c-fdc1b6cd6f16",
    "username": "janedoe",
    "roleId": 2
}
```

Code 2.4: Payload.AddMember

The Code 2.5 is the sample JSON data for creating a new duty in a family. `isPointSystem` field has a `false` value by default. The datetime format should be `yyyy-MM-ddThh:mm`.

```json
{
    "closingDate": "2024-04-24T18:36",
    "dueDate": "2024-04-24T18:36",
    "familyId": "25929b0c-b2f3-4bf4-966e-7d5782d549c8",
    "instruction": "Download and solve it.",
    "isPointSystem": true,
    "multipleSubmission": false,
    "publishedAt": "2024-04-22T18:36",
    "title": "Assignment I.",
    "totalPoints": 100
}
```

Code 2.5: Payload.Duty

When uploading the file for the duty, the Code 2.6 multipart formdata must be used with the `files` key.

```
files=assignment.cpp,library.cpp
```

Code 2.6: Payload.FileUpload

If the family tutor would like to give a grade to the tutee, the Code 2.7 format can be used. The `gradingComment` field is optional.

```json
{
    "username": "janedoe",
    "gradingId": "u9rjj9jf-j9fj0v09099ek-9i92i9ei",
    "points": 100,
    "gradingComment": ""
}
```

Code 2.7: Payload.Grade

2.3.3 API Documentation

The table below describes all the endpoints exposed by SEAP Rest API. Except for the public endpoints, all the other routes need an authorization header with the

JWT Bearer token. The information about the authorised role for the endpoint is also included. Changes related to the family are protected only by the family tutor. Some post requests need the payload which is mapped with this. The term "user" in the Table 2.1 represents the client that sends the request. The terms "family tutor" and "family tutee" refer to the role of the user in the given family regardless of the role of the account. The familyId - `:famId` has to be passed as a route parameter where the endpoints need authentication for the family role.

The Table 2.1 and Table 2.2 are the list of available endpoints of Rest API. The first column has two values and the second column has only one value in each row. The first value of the first column indicates the HTTP request method and the relative endpoint. The second value of the first column shows the role that is authorised to access the data of the given endpoint. The second column is the description of the endpoint and the payload model needed to be passed during the request.

SEAP Rest API User Documentation	
HTTP routes	**Description and Payload**
Role	
`POST api/auth/login`	Log in to the app and retrieve the JWT token.
public	Payload.Login (Code: 2.1)
`POST api/auth/register`	Register new users to the app, with the tutee grant.
public	Payload.Register (Code: 2.2)
`POST api/my/valid`	Check if the JWT token is valid or not.
public	
`GET api/my/member`	Get my membership information.
tutor, tutee	
`GET api/my/role`	Get my account role.
tutor, tutee	
`GET api/my/families`	Accessible by any user account, return the information about all the families the user is a member of.
tutor, tutee	

HTTP routes / **Role**	**Description and Payload**
`GET api/my/duties` / tutor, tutee	The list of duties that are related to the user will be returned.
`GET api/my/family/:famId/myrole` / tutor, tutee	Get the role of the user in the given family. The family id is loaded into the route parameter.
`GET api/my/family/:famId/members` / family tutor	Get the list of family members. The request will be authorized only if the user is the tutor of the given family.
`GET api/my/family/:famId/duties` / family tutor, family tutee	Get the list of duties inside a given family, returning all the duties associated with that family. The family id is loaded into the route parameter.
`GET api/my/family/:famId/duty/:dutyId` / family tutor, family tutee	Get the duty by id.
`GET api/my/family/:famId/duty/:dutyId/grading` / family tutor	Get all the grades of the family members related to the given duty.
`GET api/my/family/:famId/duty/:dutyId/my-grading` / family tutee	Get the grade information about the given duty.
`POST api/my/family/:famId/duty/:dutyId/submit/:gradingId/done` / family tutee	Submit the duty. Once submitted, the user cannot modify the submission.

HTTP routes **Role**	**Description and Payload**
`POST` `api/my/create/family`	Create a new family only by a user that has a tutor account.
tutor	Payload.Family (Code: 2.3)
`POST` `api/my/family/:famId/addMember`	Add a new member to the family. The new member can be granted a tutor or tutee role.
family tutor	Payload.AddMember (Code: 2.4)
`POST` `api/my/family/:famId/create/duty`	Create a new duty inside a family. Granted only by a family tutor.
family tutor	Payload.Duty (Code: 2.5)
`POST` `api/my/family/:famId/create/grade`	Add a grade for the tutee who submitted the duty.
family tutor	Payload.Grade (Code: 2.7)
`POST` `api/my/cdn/upload/:dutyId/given-file`	Upload a file to the duty as a material by family tutor.
family tutor	Payload.GivenFile (Code: 2.6)
`POST` `api/my/cdn/upload/family/:famId/duty/:dutyId/submitted-file`	Upload a file to the duty as a submission by the family tutee.
family tutee	Payload.SubmittedFile (Code: 2.6)
`GET` `api/my/cdn/download/:famId/:dutyId/file/:fileId`	Download a given file in a duty. Accessible by family members.
family tutee, family tutor	

HTTP routes / **Role**	**Description and Payload**
`GET api/my/cdn/download/family/:famId/duty/:dutyId/submitted-file/:fileId`	Download a submitted file in a duty. Accessible only by a tutee who submitted this file and a family tutor.
family tutor, specific family tutee	
`GET api/my/cdn/download/:famId/family-icon`	Download a family picture. Accessible only by family members.
family tutor, family tutee	
`DELETE api/my/cdn/delete/family/:famId/duty/:dutyId/submitted-file/:fileId`	Delete a submitted file. It is not allowed to delete the file in which the duty has expired or submitted. This endpoint is only accessible by the member who uploaded this file.
specific family tutee	
`DELETE api/my/family/:famId/duty/:dutyId`	Delete a duty. All the data related to this duty will be deleted.
family tutor	
`DELETE api/my/family/:famId`	Delete a family. All the data related to this family will be deleted.
family tutor	

Table 2.1: Routes applicable by the user

The Table 2.1 contains all the endpoints information that a normal user can send and the Table 2.2 shows the endpoints that is accessible only by the admin.

SEAP Rest API Admin Documentation	
HTTP routes / **Role**	**Description and Payload**
`GET` `api/admin/members` / admin	List all the users.
`GET` `api/admin/member/:id` / admin	Get a user by ID.
`DELETE` `api/admin/member/:id` / admin	Delete a user account.
`POST` `api/admin/promote/member/:username` / admin	Grant a tutor role to the given username, which will be authorised to create a family.
`POST` `api/admin/demote/member/:username` / admin	Revoke a tutor role from the given username, and cannot create a family anymore.

Table 2.2: Routes applicable by the admin

2.4 SEAP WebUI

SEAP WebUI is specialised for its simplicity and minimalism. SEAP WebUI depends on the SEAP Rest API and SEAP WebUI cannot run by itself without establishing the connection with the SEAP Rest API. Http protocol is used for the communication betweeen the Rest API server and the WebUI. The pages for the whole web app is as follows.

- Login (2.4.1)
- Register (2.4.1)
- Welcome (2.4.1)
- Family (2.4.1)
- Duty (2.4.1)
- Profile (2.4.1)
- Create family form (2.4.1)
- Duties categorised by family (2.4.1)
- Create duty form (2.4.1)
- Family member (2.4.1)
- Grading (2.4.1)
- Submission (2.4.1)

2.4.1 WebUI Pages

Login

Route: `/login`

As soon as entering the web app, if the user has not been authenticated, the web app will be redirected to the login page where the user needs to fill username and password to log in. If the user does not have the username or password, a link will redirect to the register page.

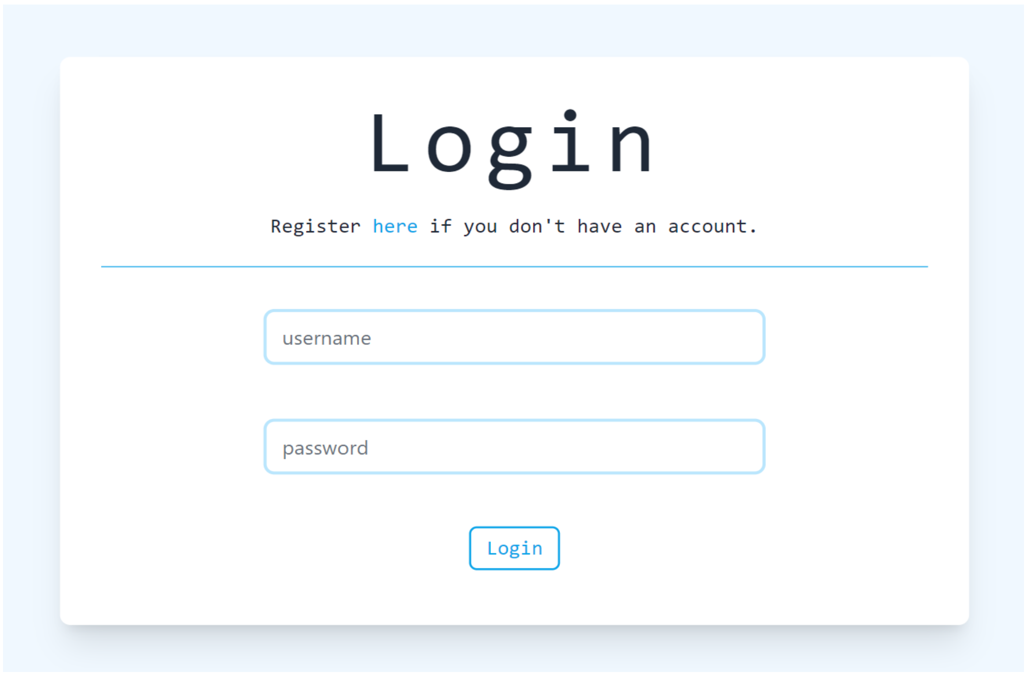


Figure 2.3: Login Page Snapshot

Register

Route: `/register`

If the user does not have a credential for the login, the user can register for the new account. Upon registering, the account will be granted a tutee role which is authorised to be added to other families by other family tutors.

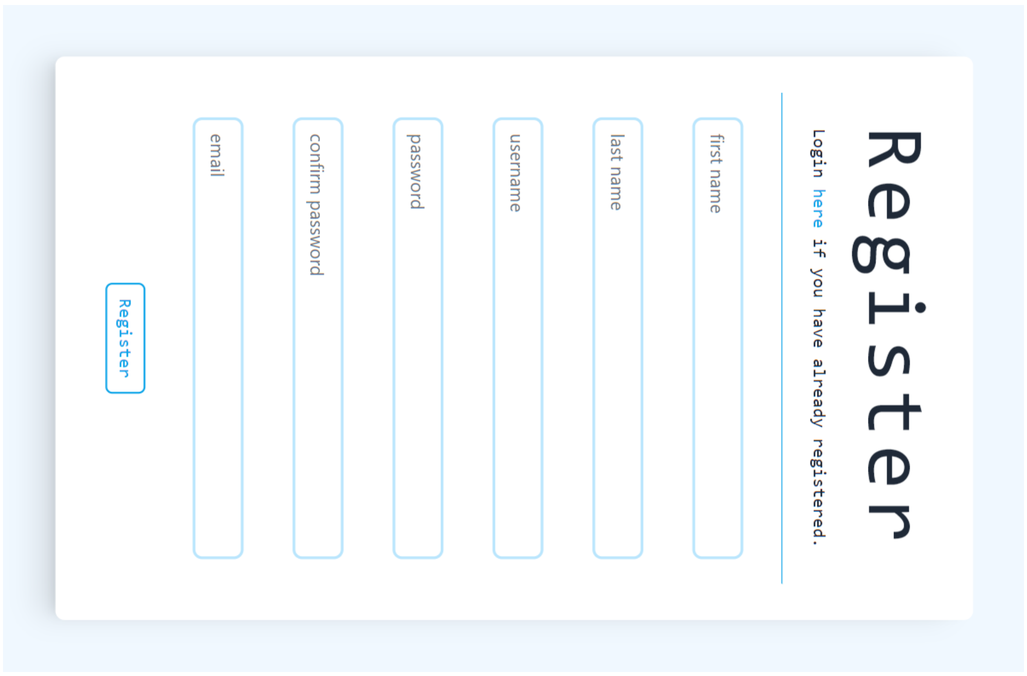
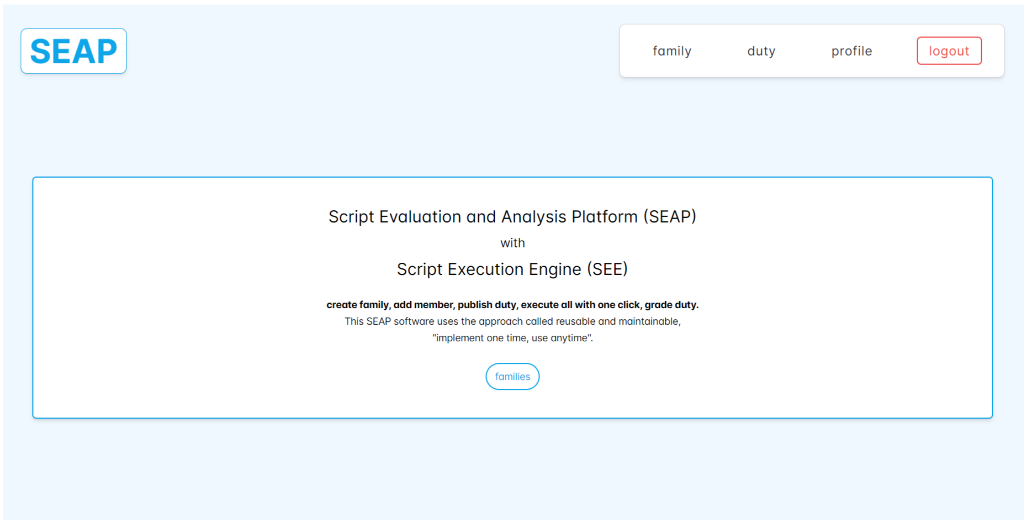


Figure 2.4: Register Page Snapshot

Welcome

Route: `/main`

After authenticating, this main landing page will appear. It is a static page which will be replaced with a statistic dashboard in the future.



Script Evaluation and Analysis Platform (SEAP)
with
Script Execution Engine (SEE)

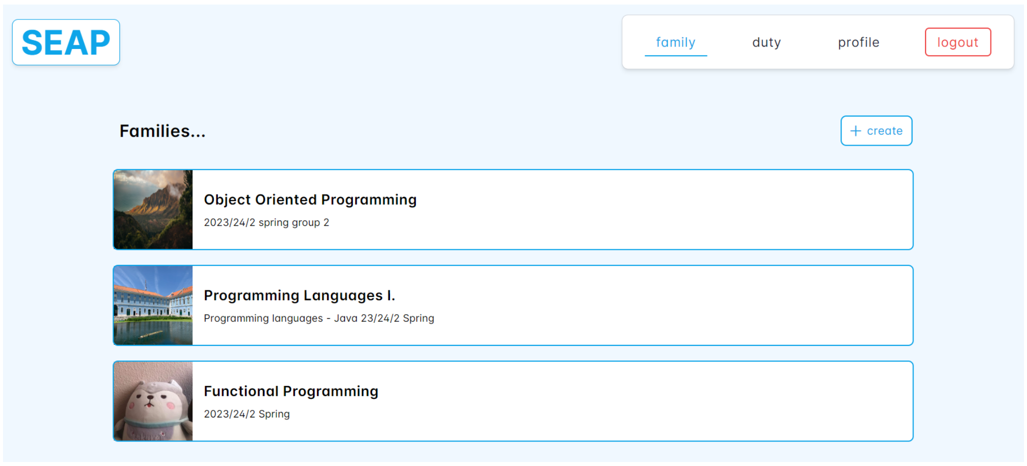
create family, add member, publish duty, execute all with one click, grade duty.
This SEAP software uses the approach called reusable and maintainable, "implement one time, use anytime".

Figure 2.5: Landing Page Snapshot

Family

Route: `/main/family`

All the families associated with the user are listed in this page. In this version, there is no sorting, searching, or filtering feature. Families in which the user is a tutor and families in which the user is a tutee are mixed. If the user has no family, then nothing will be shown but just a text message, `There is no family.`. Only the user who has a tutor account can create the new family.



Families...

Object Oriented Programming
2023/24/2 spring group 2

Programming Languages I.
Programming languages - Java 23/24/2 Spring

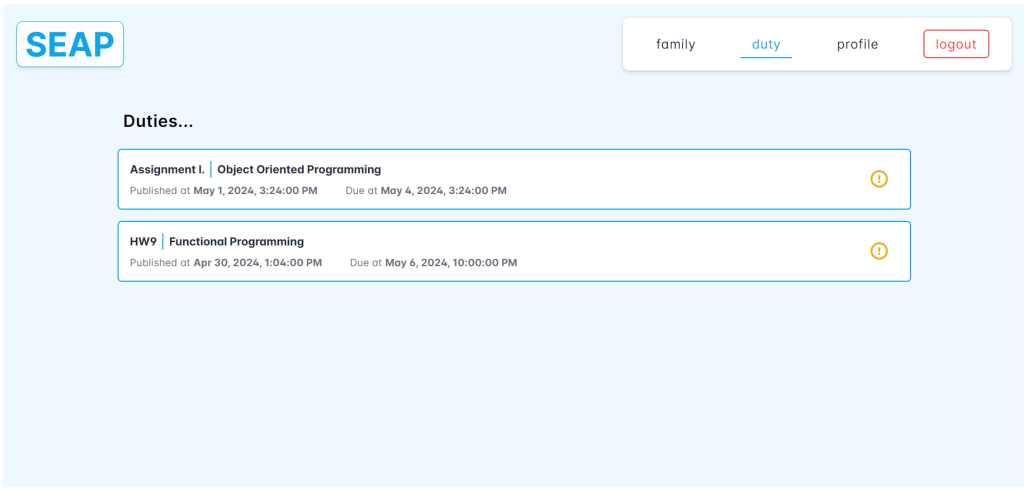
Functional Programming
2023/24/2 Spring

Figure 2.6: Family Page Snapshot

Duty

Route: `/main/duty`

All the duties where the user is assigned are listed in this page. A user can be a tutor in some families and can also be a tutee in other families. If a user is a tutee in a family, the user needs to submit the duty. This is the page where all the duties needed to submit by the user are collected. If the user has no duty, nothing will be shown except a text message. `There is no duty to submit`.



Duties...

Assignment I. | Object Oriented Programming
Published at May 1, 2024, 3:24:00 PM   Due at May 4, 2024, 3:24:00 PM

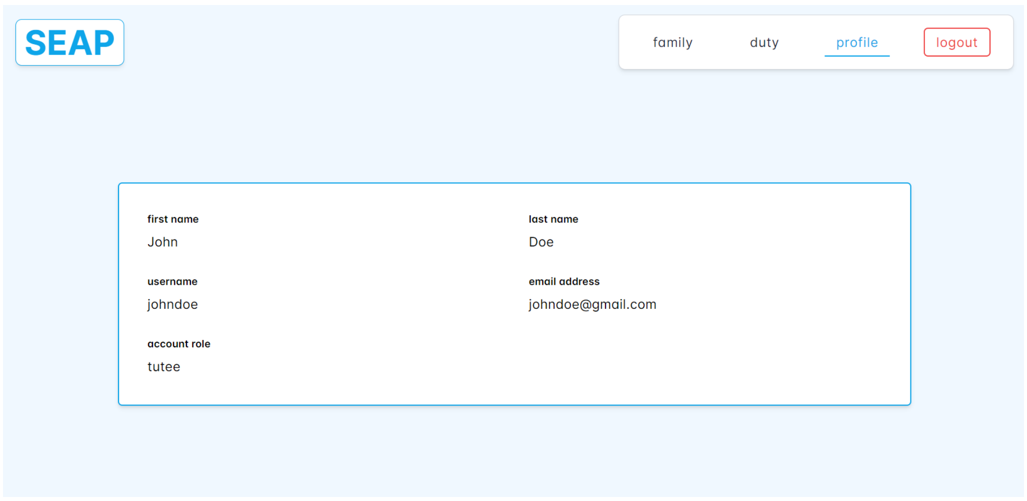
HW9 | Functional Programming
Published at Apr 30, 2024, 1:04:00 PM   Due at May 6, 2024, 10:00:00 PM

Figure 2.7: Duty Page Snapshot

Profile

Route: `/main/profile`

The logged user account information can be viewed here. Later, there will be a possibility to set up the profile picture and to modify the profile information.



first name
John

last name
Doe

username
johndoe

email address
johndoe@gmail.com

account role
tutee

Figure 2.8: Profile Page Snapshot

Create Family Form

Route: `/main/family/new-family`

For a tutor account user, it is possible to create a new family. While creating a family, the family name, description and family picture should be filled and submitted.

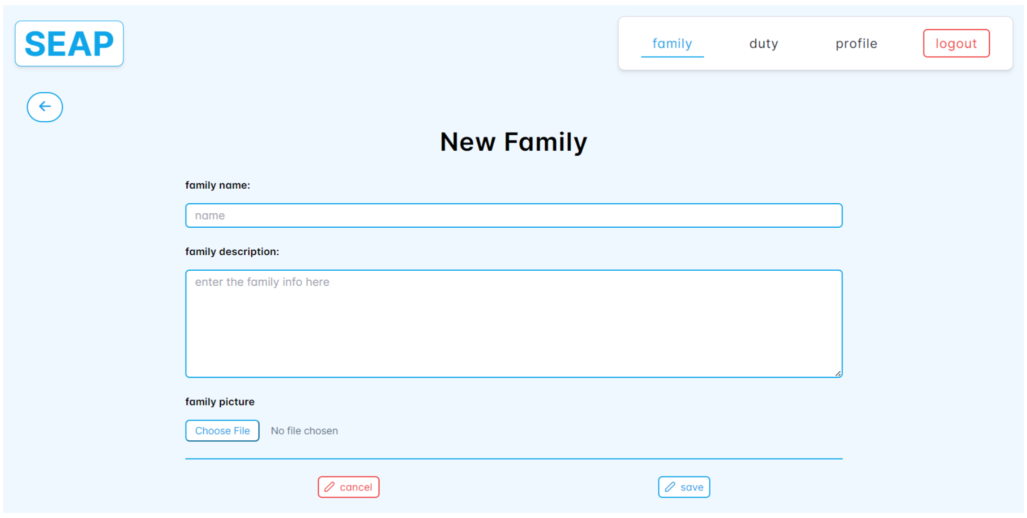
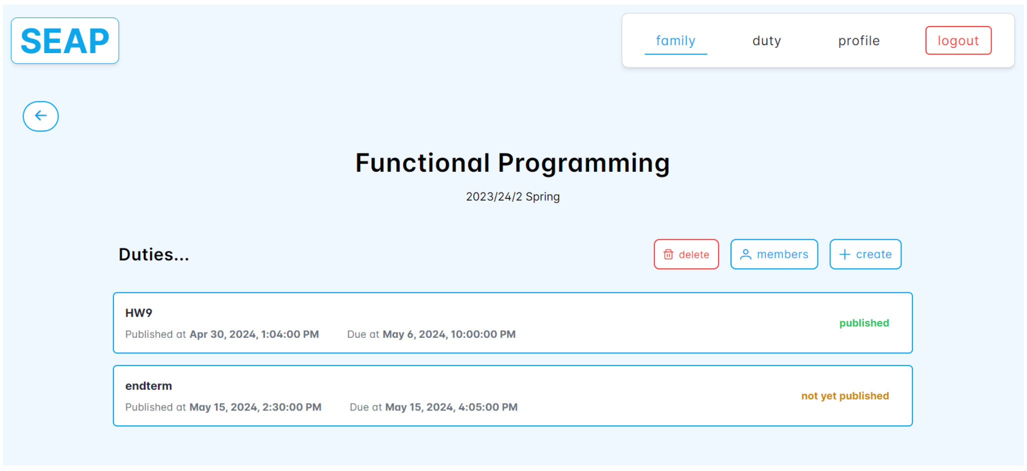


Figure 2.9: New Family Form Page Snapshot

Duties Categorised by Family

Route: `/main/family/:famId`

In the family page, all the available families are listed and when the user click one of the family, the web page will be redirected to the duty page associated to that family. For a tutor of the family, all the duties will be available to see and for a tutee of the family, only the duties that has been published can be seen.



Functional Programming

2023/24/2 Spring

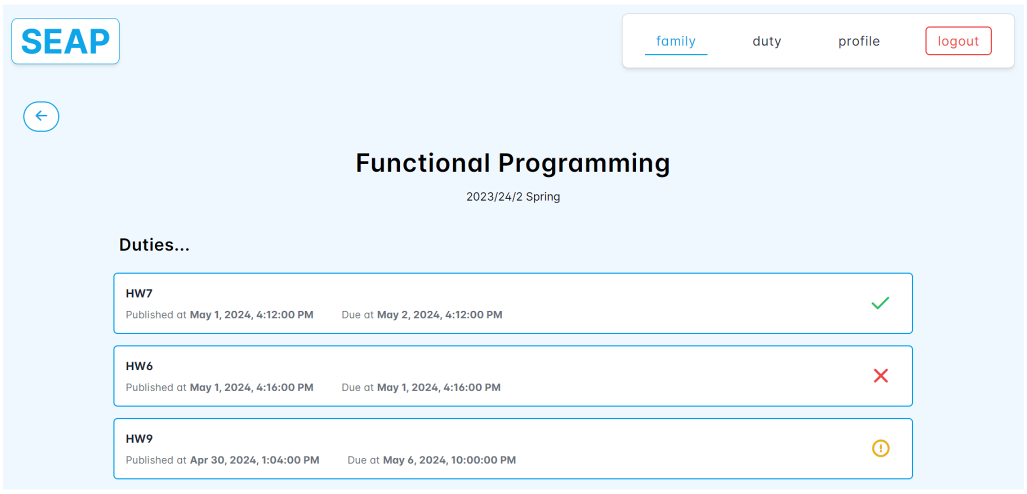
Duties...

HW9
Published at Apr 30, 2024, 1:04:00 PM Due at May 6, 2024, 10:00:00 PM
published

endterm
Published at May 15, 2024, 2:30:00 PM Due at May 15, 2024, 4:05:00 PM
not yet published

Figure 2.10: Single Family Page Snapshot with Family Tutor

For a tutee of the family, there is no authorization to modify or create the duty. The yellow `!` icon is used to highlight the duty that has not been submitted, the red cross `X` for the duty that missed the due date, and the green tick for the duty that has been submitted.



Functional Programming

2023/24/2 Spring

Duties...

HW7
Published at May 1, 2024, 4:12:00 PM Due at May 2, 2024, 4:12:00 PM ✓

HW6
Published at May 1, 2024, 4:16:00 PM Due at May 1, 2024, 4:16:00 PM ✗

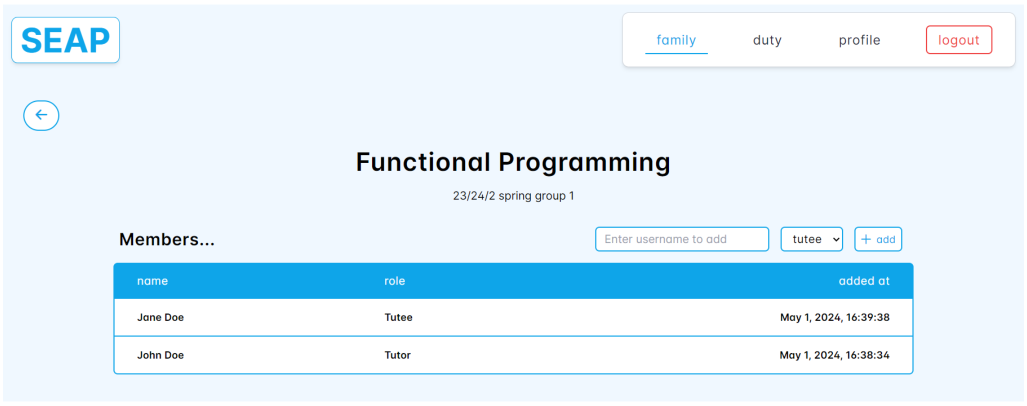
HW9
Published at Apr 30, 2024, 1:04:00 PM Due at May 6, 2024, 10:00:00 PM !

Figure 2.11: Single Family Page Snapshot with Family Tutee

Family Member

Route: `/main/family/:famId/members`

The member of the family can be viewed in this page. Only the tutor of the family is allowed to access this route and to add new member and to promote the tutee to become the tutor.



Functional Programming

23/24/2 spring group 1

Members...

name	role	added at
Jane Doe	Tutee	May 1, 2024, 16:39:38
John Doe	Tutor	May 1, 2024, 16:38:34

Figure 2.12: Family Member Page Snapshot

Create Duty Form

Route: `/main/family/:famId/new-duty`

A user who has a tutor role in the family can create the duty. When creating a duty, there is a possibility to schedule the duty for the future. The given material for the duty can also be attached. If the close date is not filled, the duty will be closed after the due date.

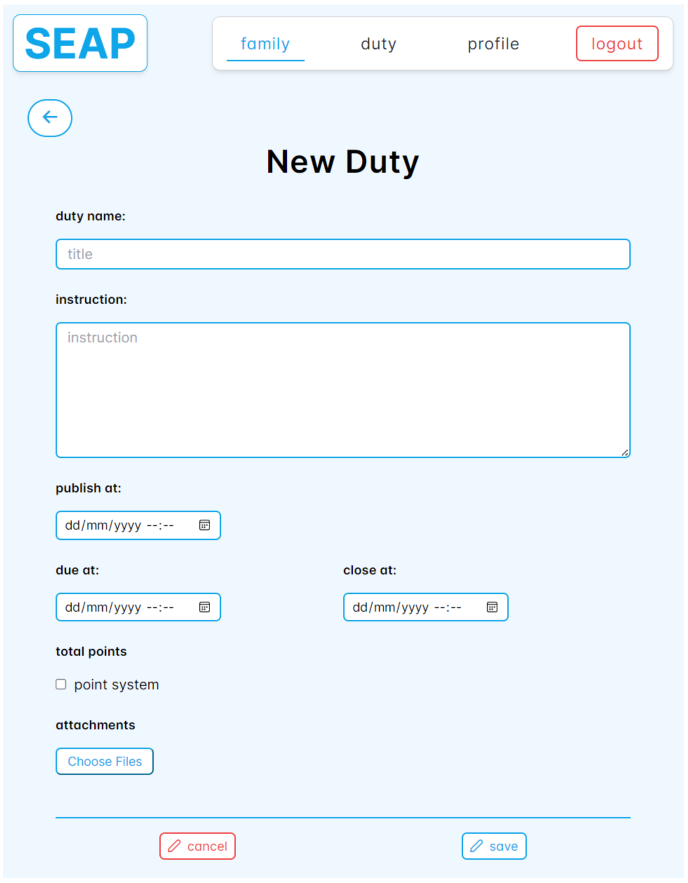
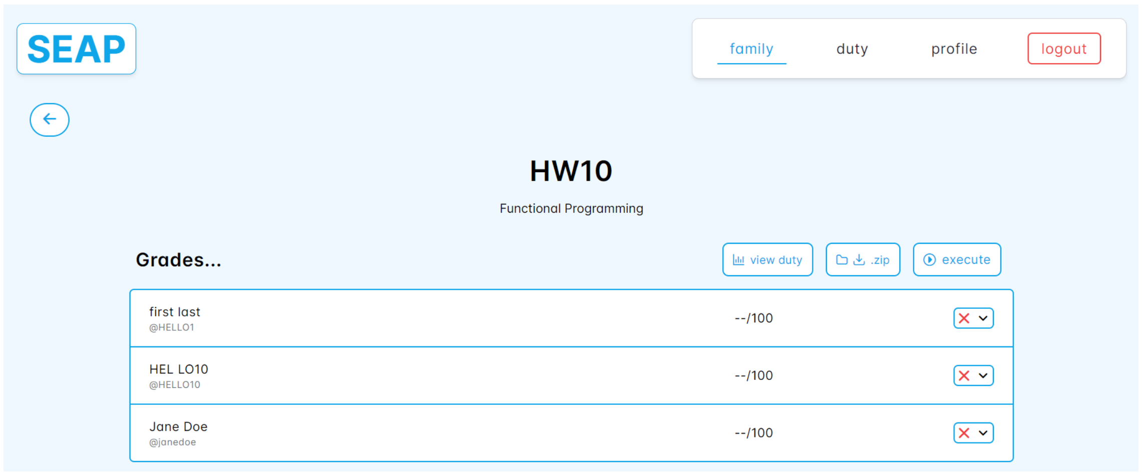


Figure 2.13: Create Duty Form Snapshot

Grading

Route: `/main/family/:famId/duty/:dutyId`

When a specific duty is opened, for a tutor, it will redirect to the grading page for that duty, and for a tutee, the submission page will be redirected. This grading page is restricted to be accessible only by the tutor of the family. A tutor can see all the student work in this page and can download submitted file, can grade the submission, can give a comment for the submission. If the student has not submitted the duty, the red cross is indicating that the student has not submitted the duty. If the row is clicked, the submission by the tutee will appear. (see Figure 2.15). The family tutor can download the submitted file and manually check it. The Figure 2.16 shows the form to grade the submission by the family tutor.



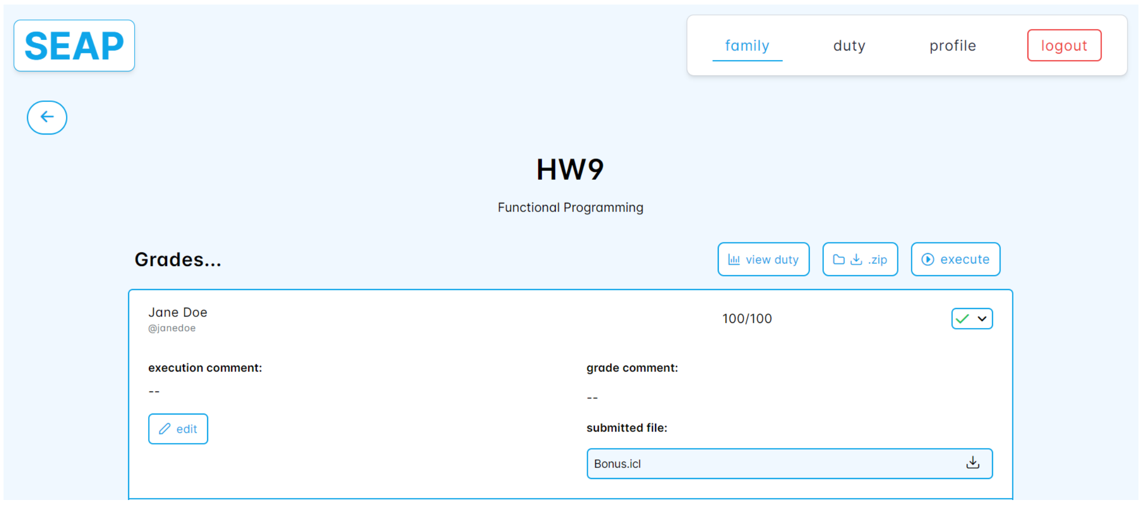
HW10

Functional Programming

Grades...

first last @HELLO1	--/100	✗
HEL LO10 @HELLO10	--/100	✗
Jane Doe @janedoe	--/100	✗

Figure 2.14: Grading Page Snapshot I.



HW9

Functional Programming

Grades...

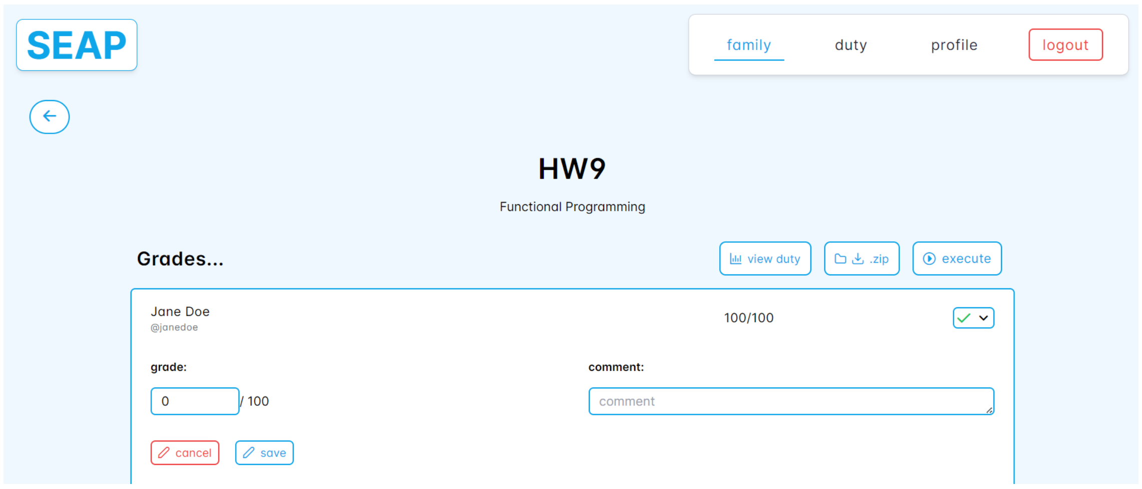
Jane Doe
@janedoe
100/100 ✓

execution comment:
--

grade comment:
--

submitted file:
Bonus.icl

Figure 2.15: Grading Page Snapshot II.



HW9

Functional Programming

Grades...

Jane Doe
@janedoe
100/100 ✓

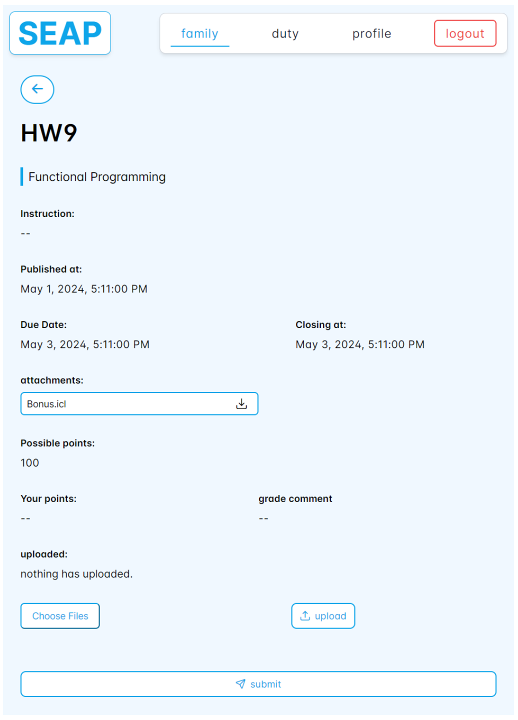
grade:
0 / 100

comment:
comment

Figure 2.16: Grading Page Snapshot III.

Submission

When a duty is clicked, if a user is a tutee of the family, the duty submission page will be redirected where the user can upload the necessary file and submit the duty. For now, the user can upload any number of files and any kind of files. Later, there will be a file input-checking feature. When the user attaches the file, the upload button should be clicked to send it to the server, otherwise, the attached files will not be uploaded. The user can attach or delete the uploaded file before submission. once the duty has been submitted, nothing can be changed. The Figure 2.17 shows the scenario when the tutee uploaded the attachments.


SEAP family duty profile logout

HW9

Functional Programming

Instruction:
--

Published at:
May 1, 2024, 5:11:00 PM

Due Date:
May 3, 2024, 5:11:00 PM

Closing at:
May 3, 2024, 5:11:00 PM

attachments:
Bonus.icl

Possible points:
100

Your points:
--

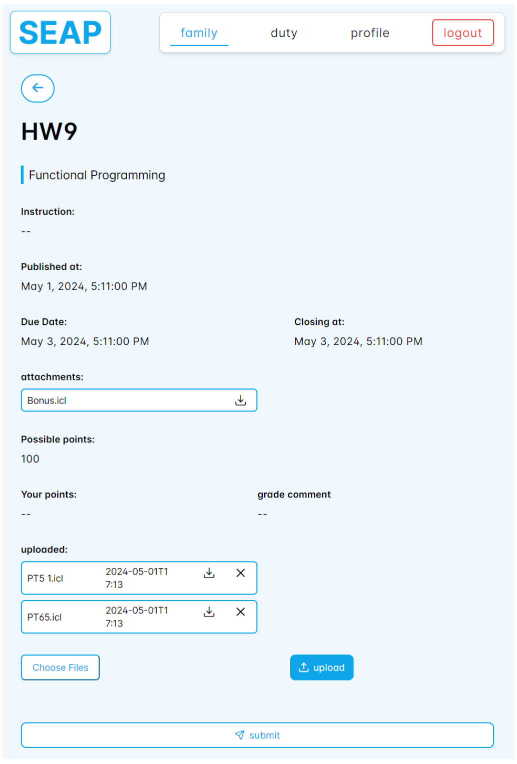
grade comment
--

uploaded:
nothing has uploaded.

Choose Files upload

submit

(a) Before Submission Snapshot I.


SEAP family duty profile logout

HW9

Functional Programming

Instruction:
--

Published at:
May 1, 2024, 5:11:00 PM

Due Date:
May 3, 2024, 5:11:00 PM

Closing at:
May 3, 2024, 5:11:00 PM

attachments:
Bonus.icl

Possible points:
100

Your points:
--

grade comment
--

uploaded:
PT5 1.icl 2024-05-01T1 7:13
PT65.icl 2024-05-01T1 7:13

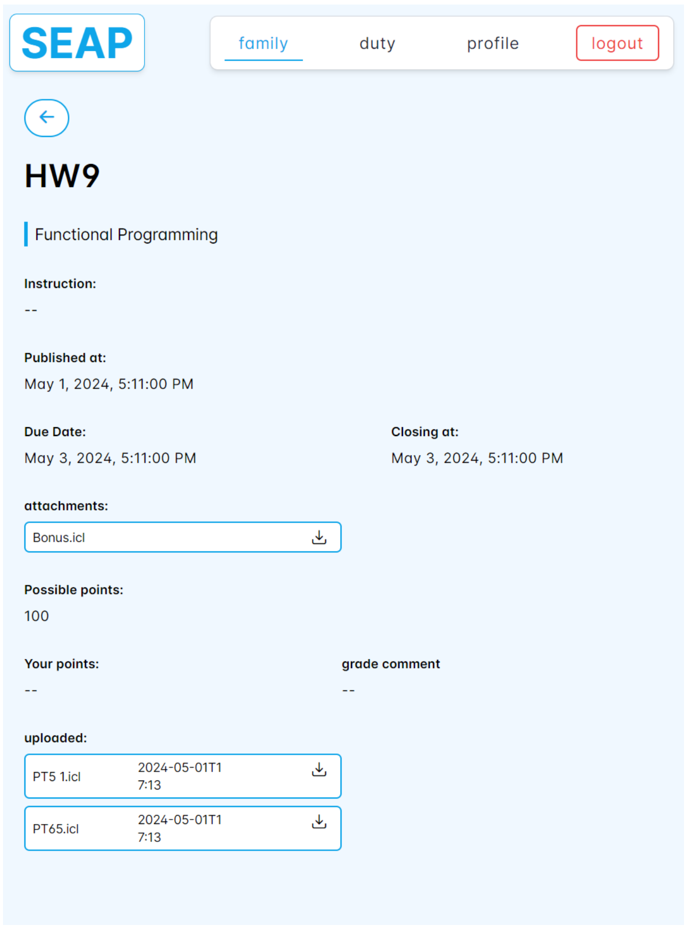
Choose Files upload

submit

(b) Before Submission Snapshot II.

Figure 2.17: Duty Snapshot Before Submitting

The Figure 2.18 indicates the submitted duty page.



HW9

Functional Programming

Instruction:
--

Published at:
May 1, 2024, 5:11:00 PM

Due Date:
May 3, 2024, 5:11:00 PM

Closing at:
May 3, 2024, 5:11:00 PM

attachments:
Bonus.icl

Possible points:
100

Your points:
--

grade comment
--

uploaded:
PT5 1.icl 2024-05-01T17:13
PT65.icl 2024-05-01T17:13

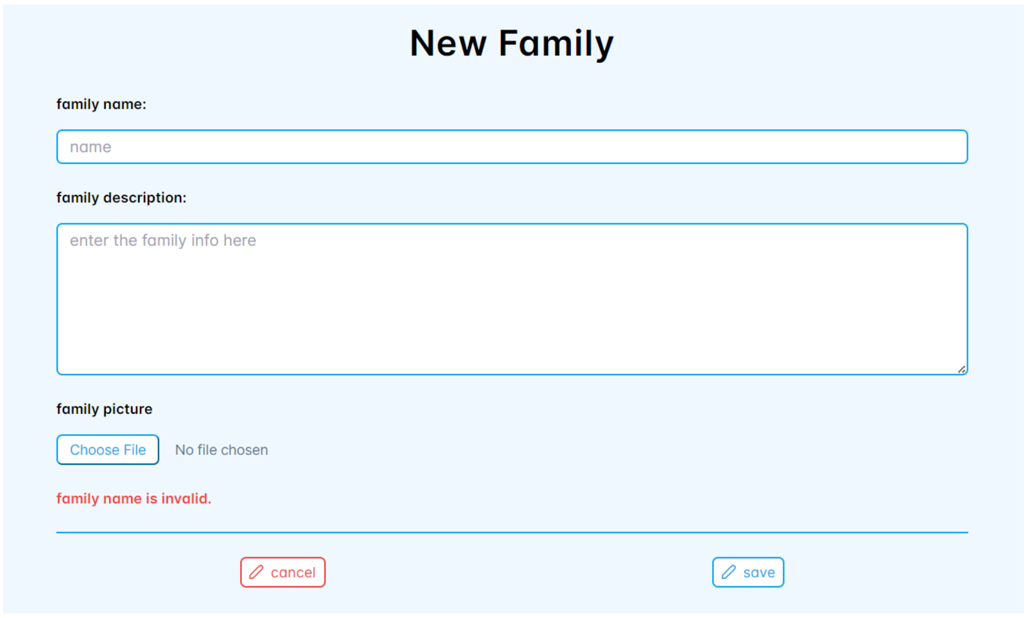
Figure 2.18: After Submission Snapshot

2.4.2 Error and Troubleshooting

There is a high possibility that there can be some errors when using the app. Mostly, the form page has many input validations and some mistakes can lead to showing an error message.

New Family

When creating a new family, the family name is the mandatory field to be filled. The family description and the family picture are not necessarily to be filled. If the form is submitted without the family name, it will raise the error.



New Family

family name:

name

family description:

enter the family info here

family picture

Choose File No file chosen

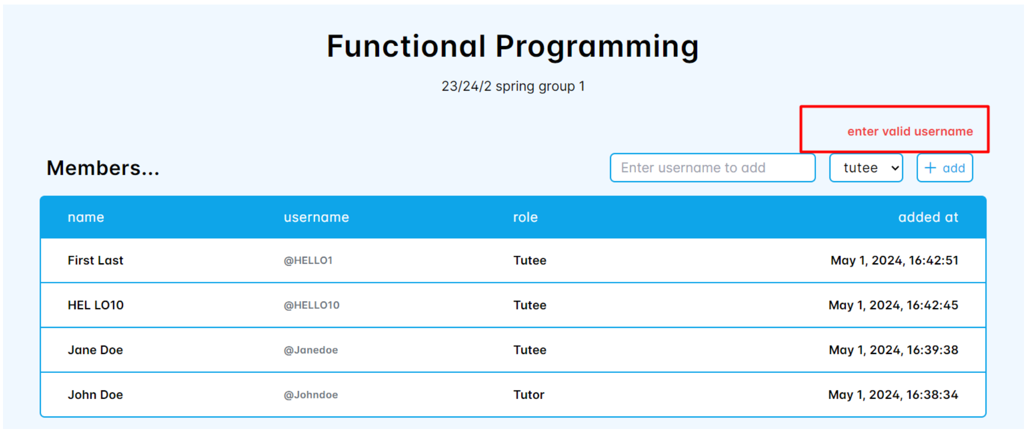
family name is invalid.

cancel save

Figure 2.19: New Family Error Snapshot

Family Member

When a new member is added by the family tutor, the form will be verified if the input is valid or not. If the form is submitted without any input, `enter valid username` will be appeared. If the family tutor is trying to add the existing member or the username that does not exist, it will show `username doesn't exist or already added to this family`.



Functional Programming

23/24/2 spring group 1

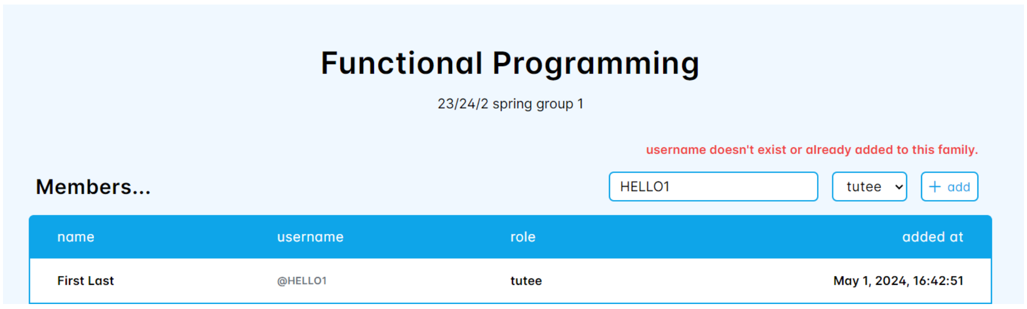
enter valid username

Members...

Enter username to add | tutee | + add

name	username	role	added at
First Last	@HELLO1	Tutee	May 1, 2024, 16:42:51
HEL LO10	@HELLO10	Tutee	May 1, 2024, 16:42:45
Jane Doe	@Janedoe	Tutee	May 1, 2024, 16:39:38
John Doe	@Johndoe	Tutor	May 1, 2024, 16:38:34

Figure 2.20: Member Form Error Snapshot



name	username	role	added at
First Last	@HELLO1	tutee	May 1, 2024, 16:42:51

Figure 2.21: Invalid Username Error Snapshot

New Duty

When creating a new duty, it is necessary to fill all the data, (Fig 2.13. The publishing date should not be earlier than the current date and the deadline should not be earlier than the publishing date. The closing date must be later than the deadline. The duty can be created without any given files.

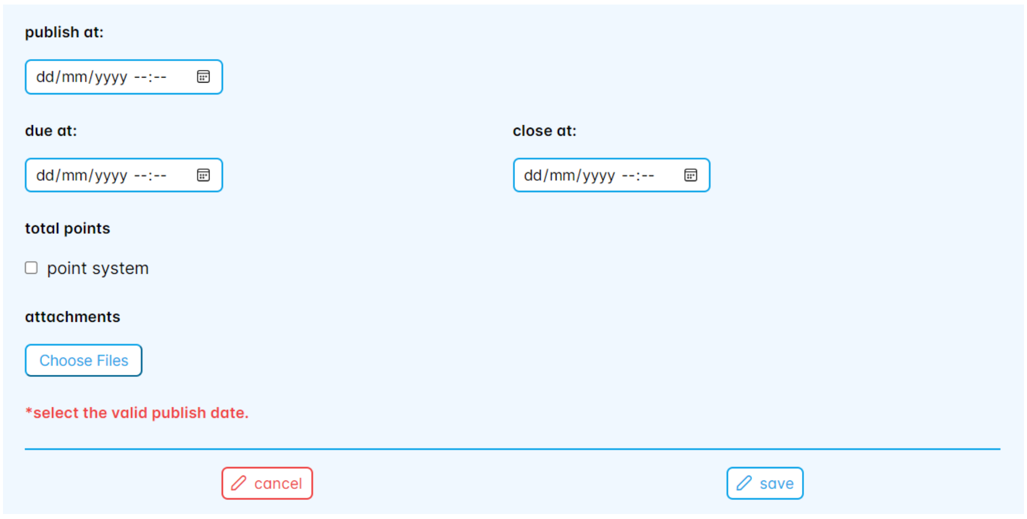
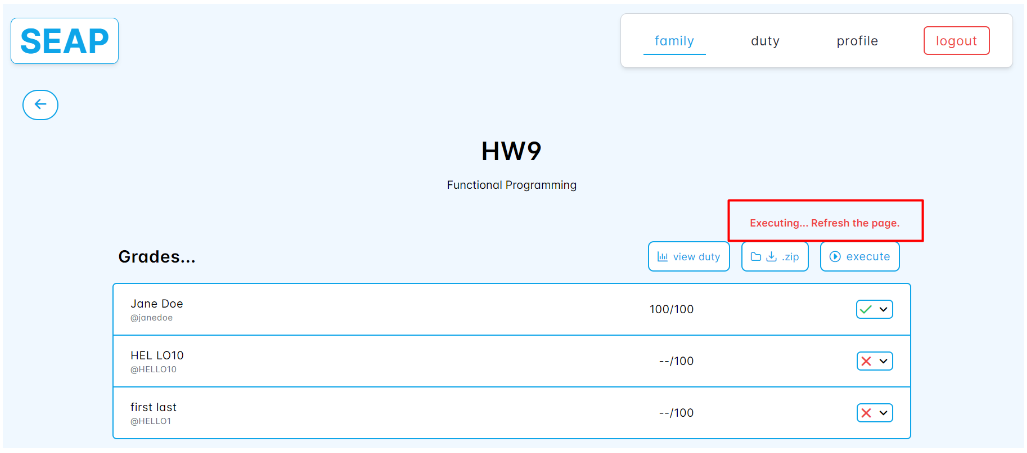


Figure 2.22: New Duty Error Snapshot

Execution

When the submission of the duties are executed, there can be error in the Rest API or any kind of warning will be appeared. When it comes to execution, it is always better to contact the maintenance, as the error cannot be solved from the client side.



Jane Doe @janedoe	100/100	✓
HEL LO10 @HELLO10	--/100	✗
first last @HELLO1	--/100	✗

Figure 2.23: Execution Error Snapshot

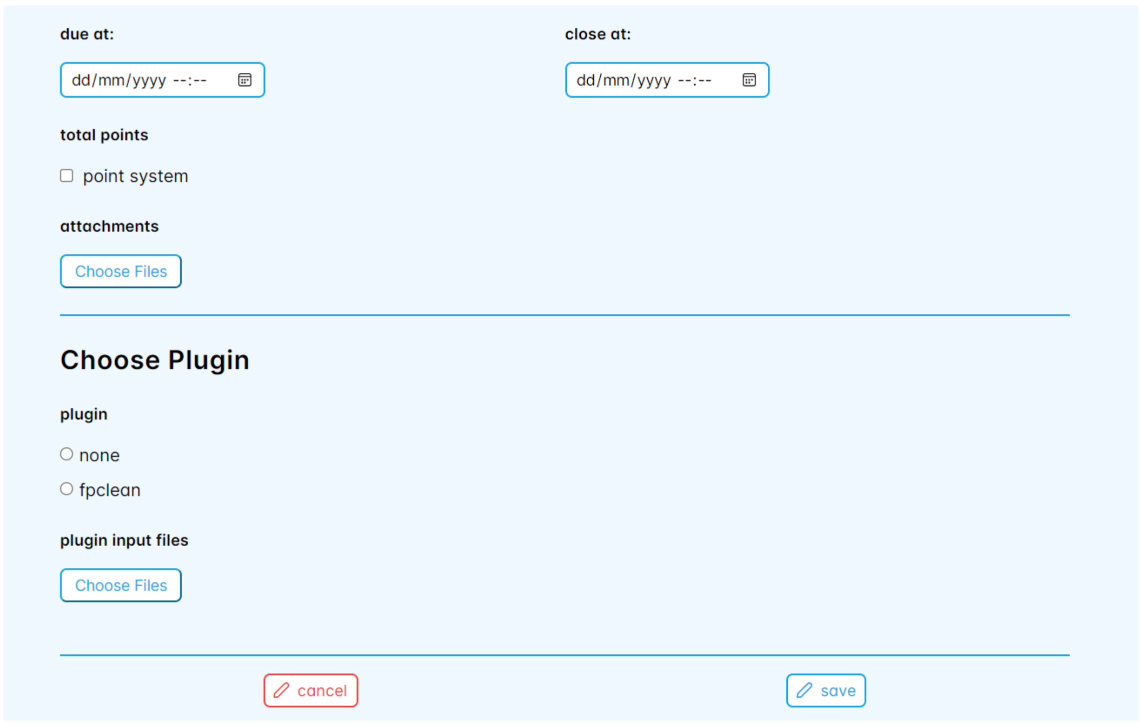
In this section, the usage of the web user interface is well explained with the figures. The troubleshooting and potential error by the user is also described in this section. However, this section does not contain the Script Execution Engine usage which is the next chapter of the user documentation.

2.5 Script Execution Engine (SEE)

The main feature of the SEAP is the automated execution of submitted files. When creating the duty, the user will be asked to select the plugin to execute the submissions. The execution can be triggered on the grading page. In the high-level user experience, there is not that much to do. The SEAP and SEE can only automate the execution and cannot automate the grading system. The user has to put the grading manually according to the execution output.

2.5.1 Plugin Selection

The available plugins are listed when creating the duty. It will not be a problem even if the user does not want to have automated execution and would like to use only an assignment management system. The available plugins are the only options to be chosen from, and if the plugin that the user wanted is not found, then the user should ask the maintenance to develop a new plugin with the specific requirement. The development of a plugin is specified in the developer documentation 3.6.2. In the version, only the plugin name can be viewed and there is no description of the plugin. In the future, the plugin description can be added and the user can read the description and get to know how the plugin is working and the user can choose it.


due at: dd/mm/yyyy --:--

close at: dd/mm/yyyy --:--

total points

☐ point system

attachments

Choose Files

Choose Plugin

plugin

○ none

○ fpclean

plugin input files

Choose Files

cancel

save

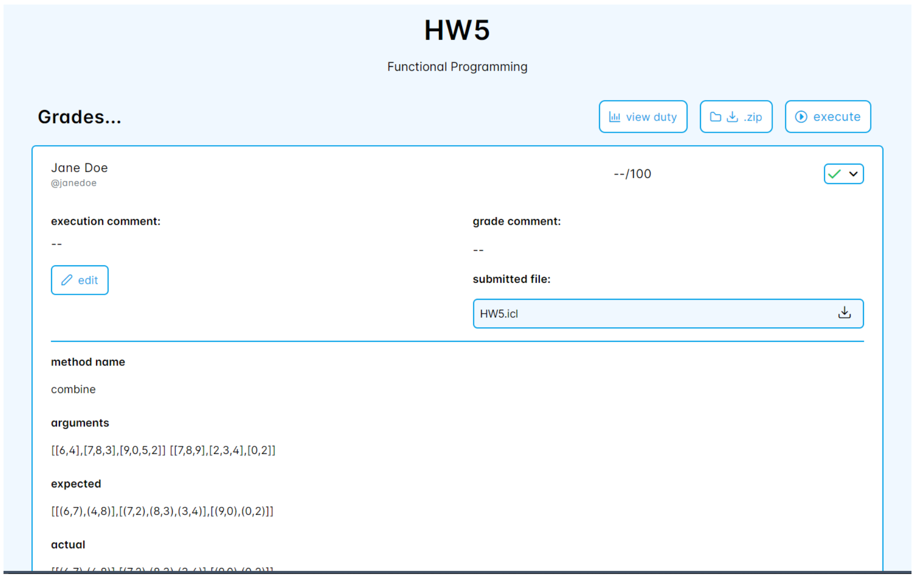
Figure 2.24: Plugin Option for Duty

The plugin can be built in many different ways. Sometimes, if the plugin is built to be generic, which means the plugin is working based on the user input, the plugin needs to specify the input values or parameters. It can be done by uploading the user input file in JSON format with the file name `plugin-param.see`. That user input parameter file can be uploaded together with the new duty.

Sometimes, the plugin needs some configuration file to execute the submission. For example, the plugin would like to receive the expected output file, so that the plugin can check the difference between the output from the execution of submitted files and the expected output. Any kind of input file can be uploaded and consumed from the plugin implementation using the SEE plugin standard library.

2.5.2 Execution Output

The `execute` button will trigger the execution and each submission will have its execution output. The plugin iterates and executes each submission and the output will be available on the grading page. In this version, the progress of the execution cannot be viewed and there is no queuing in the execution. If multiple users trigger the execution of different duties, all the execution must be queued and will be performed one by one. In this version, the execution cannot be triggered twice for each duty. In the future version, it will be more transparent and the user can view the order of the queue and estimated waiting time to finish the execution.



HW5

Functional Programming

Grades...

Jane Doe
@janedoe

--/100

execution comment:
--

grade comment:
--

submitted file:
HW5.icl

method name
combine

arguments
[[6,4],[7,8,3],[9,0,5,2]] [[7,8,9],[2,3,4],[0,2]]

expected
[[(6,7),(4,8)],[(7,2),(8,3),(3,4)],[(9,0),(0,2)]]

actual

Figure 2.25: Execution of Duty Output

There is no console output for the execution of the plugin. If the plugin raises the error, then there is no way to view it. Only the output of the execution of the submitted files will be shown. In the future, this SEAP will be improved and will have a better user experience.

In this chapter, we explored the foundational aspects of SEAP and SEE together with their respective features, roles, installation procedures, and API documentation. The user manual guide explained the specifics of the SEAP Rest API and SEAP WebUI combined with the authentication workflow, payload structures, and user interface pages. The plugin usage at the user level is also explained step by step with figures.

The next chapter will be about the development process of the SEAP and SEE. We will go deeper into the implementation details and software development lifecycle of SEAP Rest API, SEAP WebUI, and SEE. Architecture, database design, frontend and backend development, testing methodologies, deployment strategies, and maintenance considerations are described with developer-level terms. We aim to provide a thorough understanding of the development process to navigate the complexities of building robust software solutions.

Chapter 3

Developer Documentation

The whole project is developed using the microservice architecture, database, backend rest api server and frontend webui. Each microservice is based on a solid foundation of architecture using most software engineering approaches. The product is developed through a full software development life cycle such as planning, design, implementing, testing and deploying. The project had been constructed with the roadmap before starting and the author always tried to stick with the roadmap throughout the development.

This developer documentation covers all the technical aspects of the software development process. It starts with an overview of the system's architecture and how microservices connect. Then, it outlines the software and hardware requirements needed for the project. The database development, including diagrams and models, is explained next.

Additionally, it walks through the development of the Rest API server using the Go programming language, covering authentication, authorization, routing, middleware, and object mapping. Each layer of the architecture is detailed along with how they interact.

We will go further into WebUI development using Angular and TypeScript, learning topics such as architecture, guards, routing, and HTTP interceptors. The execution of the plugin and the implementation of the Script Execution Engine will be explained.

Furthermore, this documentation will provide a comprehensive overview of testing methodologies, including unit testing techniques like black and white-box testing.

3.1 Architecture and Design

When I started to develop this product, the database was the first step of the development phase. The database applied in this product is MySQL database and also design phase of the database system was quite tough. The draft database model is designed with a thorough evaluation of database and software engineering principles.

The second step was developing the backend rest api services and a combination of the previously developed draft version gave a bigger picture of modelling. The backend services are implemented with `Go` programming language using Gin library for http services and Gorm library for the database object relation mapping. The combination of the layered architecture, Model-Service-Controller, and the software engineering design patterns in each layer generates a beautiful implementation of fully maintainable, reusable clean source code. This SEAP is not just a product, but also the art of software engineering. It is not enough to take care only about the source code. It is also important to carefully design the api endpoints which will affect the user experience and the frontend architecture.

The third step was the implementation of the WebUI with the client-side framework, angular. The SEAP theme applies minimalism and simplicity, with analysis and thorough testing for the user experience. The modelling of the database and backend is creating a strong foundation for the frontend and because of the good design in the database and the backend, the effort needed is reduced by a significant amount.

Here in this section, every necessary detail and explanation will be specified for the maintenance and future development.

3.2 Installation

As this is a microservice web application, there are many components to install just to host in the local machine. Let us start with the database and backend rest api server. There are some software needed for the local development environment.

3.2.1 Hardware Requirements

- Processor: 2 GHz or faster CPU
- RAM: 4 GB or more recommended

- Storage: At least 1 GB of available disk space for application files and dependencies

3.2.2 Software Requirements:

- **MySQL server 8.0.31[9]**
  `https://dev.mysql.com/downloads/installer/`
  `mysql -version`
- **MySQL Workbench[10]**
  can be installed together in the MySQL installer if the command line interface is complicated to use.
- **Go Programming Language go1.22.0[11]**
  `https://go.dev/doc/install`
  `go version`
- **Nodejs v20.9.0[12]**
  `https://nodejs.org/en`
  `node -v`
  `npm -v`
- **Postman API Platform[13]**
  `https://www.postman.com/downloads/`
- **Linux**
  `https://ubuntu.com/`

After installing the necessary software, we can start configuring our microservices.

MySQL Database

Please enter the local MySQL server and open the `seap_db.sql` script and execute it in the MySQL server. Try to query some tables with the following commands:

```
use seap_db;
select * from member;
```

Go Rest API

If everything is working so far, we can continue to configure the backend rest API. The Go project can be built and run only in the Linux Operating System. The

Windows Subsystem Linux can also be used. Please download the project and enter the project directory. Create the `.env` file with the format given in the `sample.env`. Fill all the required data in the `.env` file.

Since the project uses Go modules, `go.mod`, dependencies are managed automatically. The project just needed to build, and Go will download and install the required dependencies.

```
go build
```

Once the dependencies are installed, the project can be run:

```
go run main.go
```

Open the new terminal and execute the curl command to send the request to the rest api server.

```
curl localhost:8000
```

If the welcome response is shown, then the configuration for the backend is done. Postman app can be used as a client application to send the http request to the server. Open the Postman app and import the api collections `SEAP_API.postman_collection.json` from the project directory,

Angular WebUI

Now we can start running the frontend service. Download the project and enter the project directory.

There is a file named `package.json` to install the required dependencies. The following command will download all the dependencies which will take a while.

```
npm install
```

After that, the backend rest api endpoint should be configured in this angular app. Enter this directory `src/environments/environment.development.ts` and change the `apiUrl` variable with the proper address and port. The app finally can be run with

```
npm start
```

Open the browser and we can start to use the app. There are three built-in credentials and those credentials must be changed.

```
username - password - role
admin - admin123 - admin
johndoe - johndoe123 - tutor
janedoe - janedoe123 - tutee
```

3.3 SEAP Database with MySQL

The development of the database system proved to be challenging. There were many steps to overcome just to get a good database. The database design phase is one of the most important steps and it will affect the performance of the application. If there is a good database in terms of the design, it is 50% winning stage in the development process. The design of the database will also affect the modelling of the backend implementation. The entity-relationship model between tables should be designed properly.

Instead of using the ID values as an incremental, the universal unique identifier (UUID) is used in every table. There are three built-in account credentials which are inserted into the respective table upon creation. Let us look at the database entity relation diagram (see Figure 3.1).

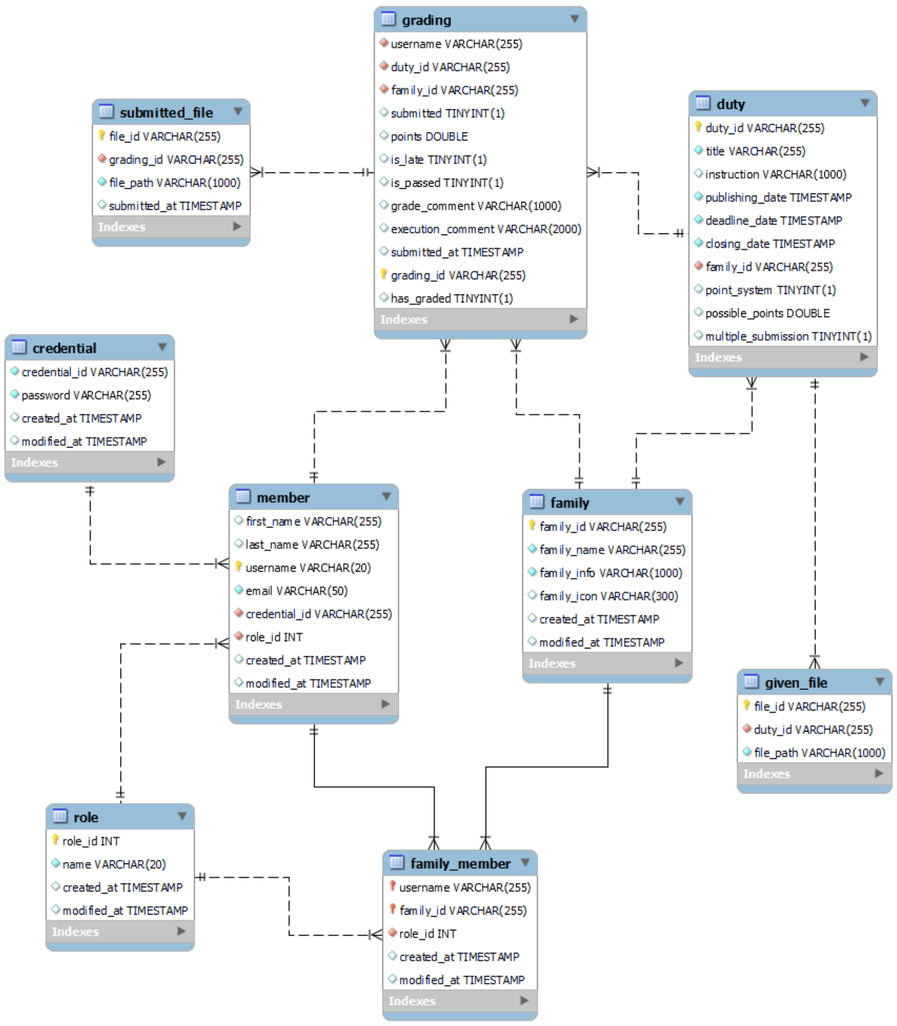


Figure 3.1: Database entity relationship diagram

3.3.1 Tables and their Functions

Some tables are connected with a foreign key constraint and if the data from the parent table is deleted, all the data in the child table associated with that parent will also be deleted. Some tables need auxiliary tables for many-to-many mapping and some tables are connected with many-to-one mapping. The list of tables are as follows:

- role
- credential
- member
- family
- family_member
- duty
- given_file
- grading
- submitted_file

seap_db.role

Primary key: `role_id int`

Overall, there are three roles in this SEAP, admin, tutor and tutee. This role table has a primary key called `role_id`, which is a hard-coded value. `admin` has a `role_id` 99, `tutor` as 1, and `tutee` as 2. This `role` table has a connection with `member` table and `family_member` table. It is necessary to create this `role` table instead of just simply using the role value string in the other table. By creating this table and connecting other tables with foreign keys, there will not be any duplicated entry in the role value and nobody can insert new role values, that are not inside this table, into other tables.

seap_db.credential

Primary key: `credential_id varchar(255)`

A `credential` table is storing the encrypted password together with `credential_id`. When the new member is created, the password should be stored in this table first, and the `credential_id` to map with the member table.

seap_db.member

Primary key: `username varchar(20)`

Foreign key:

`credential_id varchar(255) => seap_db.credential(credential_id)`

`role_id int => seap_db.role(role_id)`

This `member` table stores all the account credentials and user data. Two foreign key constraints have been set up in this table, `credential_id` which is a foreign key column referenced with `credential_id` of the `credential` table and `role_id` from `role` table.

The role value is mapped with the `role_id` from the `role` table. So, every user will be mapped with the role that exists in the `role` table. If someone attempts to insert the new role that is not aligned with the `role_id` of the `role` table, it will raise the constraint error.

If there is a new entry in the `member` table, the `credential_id` must be obtained first by inserting the encrypted password into the `credential` table. If the encrypted password is not saved in the `credential` table first, the new record cannot be added to the `member` table due to foreign key constraint.

seap_db.family

Primary key: `family_id varchar(255)`

The family data are stored in this table and this table is a standalone table which means, there is no foreign key inside this table. However, other tables depends on this table.

seap_db.family_member

Primary key and foreign key:

`username varchar(20) => seap_db.member(username)`

`family_id varchar(255) => seap_db.family(family_id)`

`role_id int => seap_db.role(role_id)`

This table is the auxiliary table for the connection between members and families. It is many-to-many mapping which means a member can have multiple families and a family can have multiple members. `role_id` column is a reference of the role table which represents the member role associated with the family.

seap_db.duty

Primary key: `duty_id varchar(255)`

Foreign key:

`family_id varchar(255)` => `seap_db.family(family_id)`

The duties are stored in this table. The `family_id` column is the reference of the family table and it is a many-to-one mapping, which means each duty should belong to only 1 family and a family can have multiple duties. All the data related to duty, such as title, instruction, publishing date, deadline, closing date and maximum possible points, are stored in this table.

seap_db.given_file

Primary key: `file_id varchar(255)`

Foreign key:

`duty_id varchar(255) => seap_db.duty_id`

The duty can have the pre-given file when publishing it so that the tutee can use the pre-given materials and solve the duty. This is the table to store the given files related to the duty together with the file path. A duty can have multiple files, however, each file must be associated only with one duty. That is why, many-to-one is used like a family-duty relationship.

seap_db.grading

Primary key: `grading_id varchar(255)`

Foreign key:

`username varchar(255) => seap_db.member(username)`

`duty_id varchar(255) => seap_db.duty(duty_id)`

`family_id varchar(255) => seap_db.family(family_id)`

This table is the main table of the whole SEAP. This table acts as the auxiliary table between member, `family` and `duty` with some additional columns. Regardless of whether the tutee submitting the assignment or not, the grade data of the `duty` for the tutee must be stored properly. Whenever a new duty is created in a family, all the members inside that family will be inserted into this `grading` table as a record for the grade. The submission data is also recorded here.

seap_id.submitted_file

Primary key: `file_id varchar(255)`

Foreign key:

`grading_id varchar(255) => seap_db.grading(grading_id)`

All the files submitted by the tutee are stored in this table together with the file path. The time when the file was uploaded is also recorded. It is many-to-one mapping such that each grading can have multiple files but a file must be associated only with 1 grading.

3.4 SEAP Rest API with Go, Gin and Gorm

The main part of the whole application is located in this backend service. SEAP Rest API is developed with the Go programming language[11] which is highly specialised in concurrency and parallel programming. The syntax is simple and the features are also minimalist. Go is compiled into the machine code, generating a single executable binary file, similar to C and C++. It is always better to use the static-typed language to develop the enterprise application, which gives a secure compilation and environment. Although Go is not a pure object-oriented language, it still supports the concept of object orientation.

3.4.1 Go, Gin and Gorm

Go itself can be used to create the web server without any dependencies unlike other programming languages because the http package is included in the standard library. When it comes to flexibility and simplicity, Go is always greater than Java. Even though the Rest API server can be created with a standard library, sometimes it is better to use the external library which is designed for these web frameworks. In this project, Gin, which is a library developed for creating the web server with Go, will be used[14].

The backend project should be configured to communicate with the database. Gorm[15], Golang Object Relation Mapping, is a library to build the entities and queries from the database. In the previous section, the database is already designed and structured. So, here, instead of allowing Gorm to create all the tables, Gorm can simulate the modelling of the existing database. There are no classes or objects in Go like Java, Python and C++, however Go has something similar called struct and interface. By using struct and interface, the object-oriented concepts can be implemented.

3.4.2 Architecture

This Go project is implemented with the layered architecture and in each layer, the software engineering design patterns are used. Each layer has its responsibility and purpose, implemented without any code smell. The layers include the Controller, Service and Repository. The model layer is a standalone layer that simulates the database entities and is used entire application. The controller layer should be communicating with the service layer only and the service layer is a bridge between the repository layer and the controller layer.

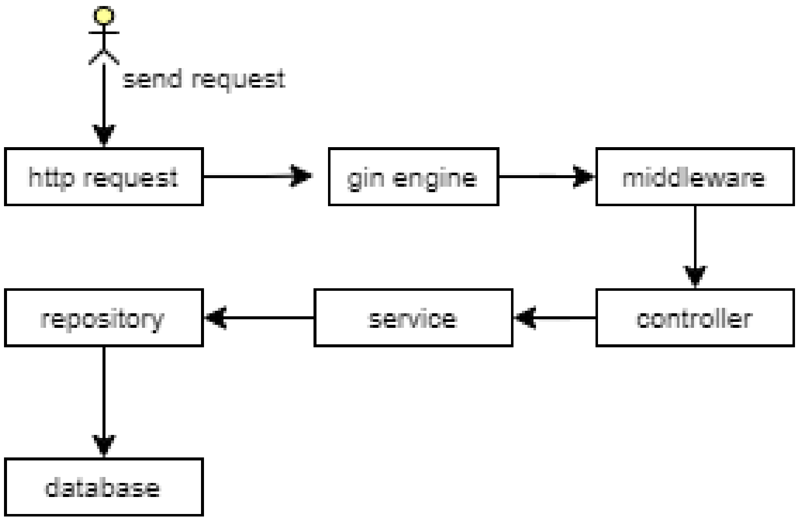


Figure 3.2: Layered Architecture of SEAP Rest API

This layered architecture helps to maintain a separation of concerns, making the codebase more modular, scalable, and maintainable. It also enables easier testing and reuse of components. The purposes and descriptions of each layer are as follows:

Model Layer

Defines the data structures (structs) representing the domain entities or data models used in the application. In this project, the model layer is aligned with the entity-relationship of the database.

Repository Layer

Handles the interactions and queries of the database using the entities defined in the model layer. It encapsulates all the database operations, not only CRUD (Create, Read, Update, Delete) operations, but also extraordinary querying, such as counting, summation, etc. This layer should only be called by the service layer and is not allowed to use the repository layer directly inside the controller layer. The Figure A.2 is the struct design for the repository layer.

Service Layer

Contains the business logic or application logic of the system. This layer serves as a middleware or a bridge between the controller and the repository. All the requests from the controller layer are evaluated and analysed in this layer and query the repository layer. In another way, it orchestrates the operations performed by the repository layer and enforces any business rules or constraints. The Figure A.1 shows the architecture of the service layer.

Controller Layer

Receives and handles incoming requests from the client, typically over HTTP in web applications. It delegates the processing of requests to the appropriate service methods and returns the appropriate responses. Authentication is also done in this layer. If the incoming request is unauthenticated or unauthorized, then the service methods will not be called and return the error. The controller layer does not interact with the repository layer directly as it will be dangerous without passing through the service layer. The Figure A.3 indicate the structure of the controller layer.

3.4.3 Authentication Workflow

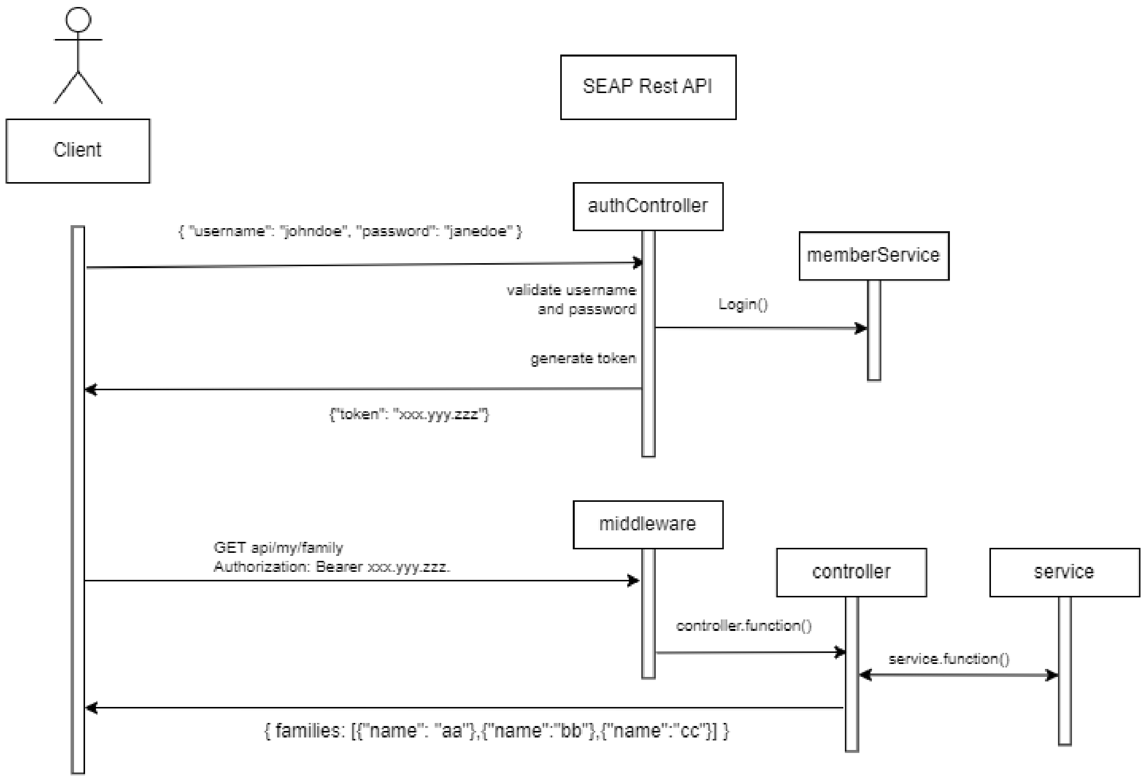


Figure 3.3: JWT Authentication Workflow

Overall, when the user logs in from the login endpoint, the server will generate a stateless JWT token with the expiration date for the user. Every request to the server has to include the JWT token for authorization. Whenever the user wants to access private resources, the user must first log in and obtain the JWT token. In this version, the authentication mechanism does not have the refresh token method. So, if the SEAP has reached the production level, it is recommended to use the rotational JWT token, which includes the refresh token in addition to the normal JWT token.

When generating a new JWT Token, some insensitive values, username and role, are added to the claims. The expiration date can be 15 minutes or 30 minutes. All the claims are signed with the unique private key stored in the server so that claims cannot be modified. The private key is very important and should be secret.

```go
func GenerateToken(username, role string) (string, error) {
    tokenTTL, _ := strconv.Atoi(os.Getenv("TOKEN_TTL"))
    token := jwt.NewWithClaims(jwt.SigningMethodHS256,
     ↪  jwt.MapClaims{
        "id":   username,
        "role": role,
        "iat":  time.Now().Unix(),
        "exp":  time.Now().Add(time.Second *
         ↪  time.Duration(tokenTTL)).Unix(),
    })
    return token.SignedString(privateKey)
}
```

Code 3.1: Payload.Duty

Recommended: Rotational JWT Token[16]

When the user logs in, besides generating a JWT token, the refresh token should also be generated and stored in the `refresh_token` table. When the JWT is about to expire or has already expired, the client side should request the new JWT with the refresh token at the refresh JWT endpoint. Upon getting the request for the refresh JWT, the server should verify the token and generate a new JWT token and a new refresh token. The refresh token should also have an expiration date which is at least 5-10 minutes longer than the expiration date of the JWT token and be allowed only for one-time use.

Whenever the users refresh the JWT with the refresh token, the server will check if the token has been used twice by verifying the token in the `refresh_token_history`. If the token is not used twice and is valid, then the token stored in the `refresh_token` table will be transferred to `refresh_token_history` table and the token in the `refresh_token` table will be overwritten with a new refresh token generated. If the token is used twice, the server should remove all the issued tokens from the `refresh_token` table. On the client side, the refresh token can be stored in the cookies or use the Backend-for-Frontend proxy server.

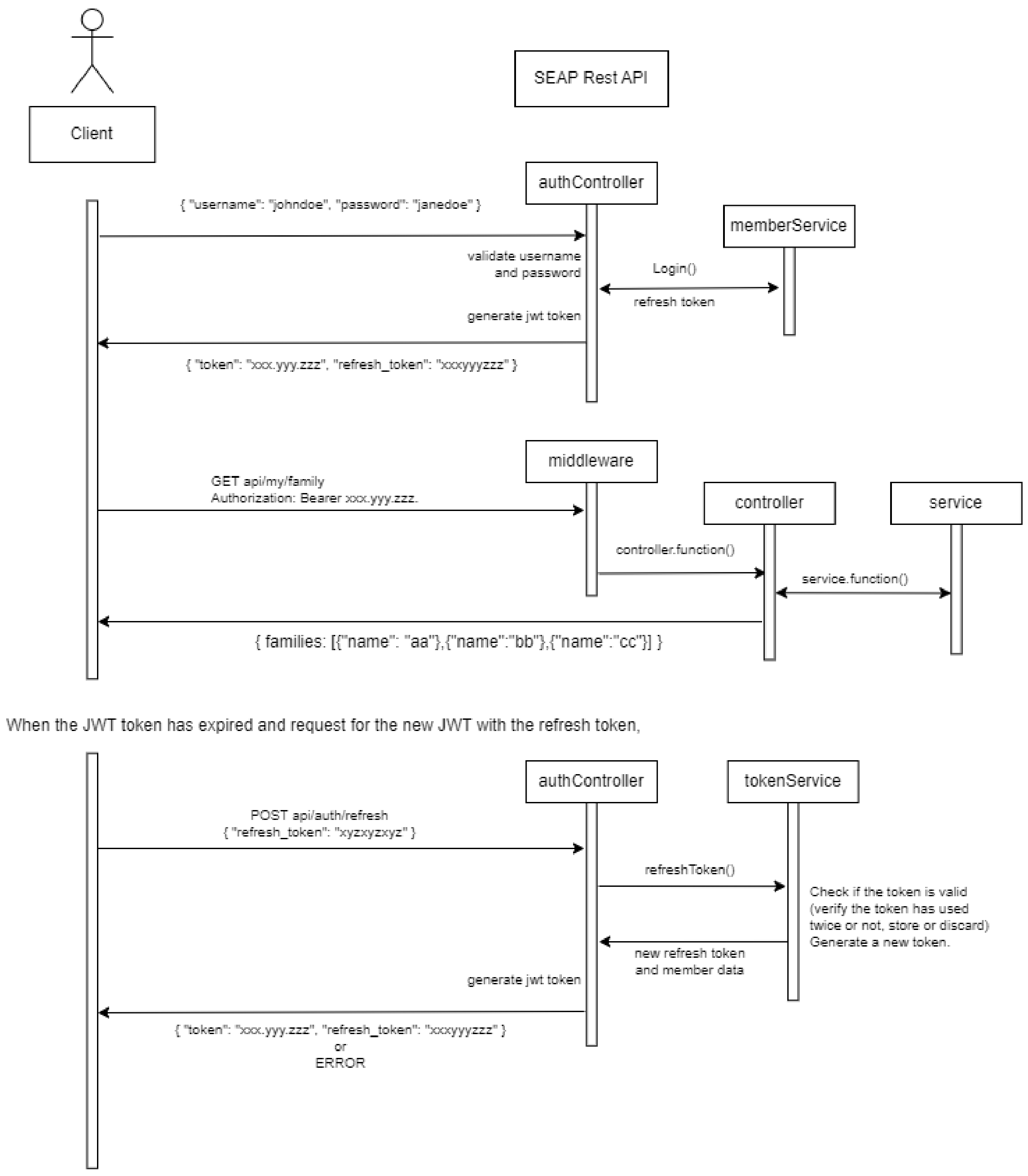

Client
SEAP Rest API
authController
memberService
{ "username": "johndoe", "password": "janedoe" }
validate username and password
Login()
refresh token
generate jwt token
{ "token": "xxx.yyy.zzz", "refresh_token": "xxxyyyzzz" }
middleware
controller
service
GET api/my/family
Authorization: Bearer xxx.yyy.zzz.
controller.function()
service.function()
{ families: [{"name": "aa"},{"name":"bb"},{"name":"cc"}] }
When the JWT token has expired and request for the new JWT with the refresh token,
authController
tokenService
POST api/auth/refresh
{ "refresh_token": "xyzxyzxyz" }
refreshToken()
Check if the token is valid (verify the token has used twice or not, store or discard) Generate a new token.
new refresh token and member data
generate jwt token
{ "token": "xxx.yyy.zzz", "refresh_token": "xxxyyyzzz" } or ERROR

Figure 3.4: Recommended: Rotational JWT Workflow

3.4.4 Routing and Authorization with Middleware

The routing and authorization are the front-line defence of the server. There are three roles, admin, tutor, and tutee. Therefore, some routes are defined only for specific roles. It is very important to restrict the specific user not to get unauthorized resources. For example: a family tutee should not be allowed to obtain duty, grading or family member information which are authorized only for the family tutor. Whenever the user requests private resources with JWT, the server should authenticate and authorize by role.

These authorizations can be implemented with the HTTP interceptor. HTTP interceptor means that any request to the server will be intercepted by the middleware and redirected to the respective controller. If the middleware rejects the request, then the request will not be redirected to the controller and the error will be returned as a response to the client.

Due to the simplicity of the Go and Gin library, it is easy to implement the HTTP interceptor or middleware. Gin supports the route grouping and each group can use different middleware. In this way, routes are grouped by their role and respective middleware is used[17]. In this SEAP, the routes can be analysed and categorised into six groups:

- admin:
  This group contains all the routes that are allowed only for the admin role.
- tutor:
  Some features, such as creating a new family, are allowed only for the user who has the tutor account. This group is authorised for the tutor account.
- individual:
  This group includes all the routes related to the individual regardless of their role. For example: fetching the family associated with the user, fetching all the duties assigned to the user, getting the profile information, and so on.
- family-member:
  Routes in this group usually have a route parameter `family/:famId` so that the middleware can authenticate if the user is part of the family or not. If the user is not part of the family, then the user should not be allowed to fetch the data related to the family.
- family-tutor:
  Family tutors can create a new duty, add a new member, grade the duty, and so on. These routes should not be allowed to enter by the other users. This group is similar to the `family-member` group, having route parameter `family/:famId` but just different middleware.
- family-tutee:
  Family tutees are permitted to submit the duty, and upload the materials to the duty. The tutor of the family cannot submit the duty or upload a file. This group is also similar to the above groups, having the route parameter

`family/:famId` and sometimes `duty/:dutyId`.

The implementation for the route grouping and middleware will look like the Code 3.2.

```go
var seapRouter *gin.Engine
seapRouter = gin.Default()

admin := seapRouter.Group("/api/admin/")
admin.Use(controller.AdminMiddleware())
admin.GET("/member/:id", controller.GetMemberById())
admin.DELETE("/member/:id", controller.DeleteMember())
admin.POST("demote/member/:username", controller.DemoteRole())
```

Code 3.2: Route Grouping Example

Each of the route groups are covered with the each middleware representing different purposes. The Code 3.3 expressed the sample implementation for the authorisation.

```go

func (a *authControllerImpl) adminMiddleware(context *gin.Context) {
    username, role, err := validateTokenAndClaims(context)
    if err != nil {
        context.JSON(parseErrorAndResponse(err))
        context.Abort()
        return
    }
    if role == "admin" {
        context.Set("username", username)
        context.Set("role", role)
        context.Next()
        return
    }
    context.JSON(http.StatusUnauthorized, "unauthorized")
    context.Abort()
}
```

Code 3.3: Middleware Implementation Example

Cross-origin resource sharing (CORS)[18]

All the requests to the server are also filtered by a common middleware specialised for Cross-origin resource sharing mechanisms. If the Rest API is integrated into other web apps hosted in different domains apart from the domain of Rest API, the browser will make the preflight OPTION request to ensure that the Rest API allows CORS. In the preflight OPTION request, the browser will request the Rest API server about the domain to be allowed to call the Rest API. If the server returns `www.aaa.com`, and if the client web is on the domain www.bbb.com, the browser will reject the API request. By enabling CORS, the server can choose which domains are allowed to integrate the Rest API. The CORS setup is illustrated in the Code 3.4.

```go
1  seapRouter.Use(controller.CorsMiddleware
2  
3  func (a *authControllerImpl) corsMiddleware(c *gin.Context) {
4    c.Writer.Header().Set("Access-Control-Allow-Origin", "http://localhost:4200")
5    c.Writer.Header().Set("Access-Control-Allow-Credentials", "true")
6    c.Writer.Header().Set("Access-Control-Allow-Headers", "Content-Type, Content-Length, Accept-Encoding, X-CSRF-Token, Authorization, accept, origin, Cache-Control, X-Requested-With")
7    c.Writer.Header().Set("Access-Control-Allow-Methods", "POST, OPTIONS, GET, PUT, DELETE")
8    c.Writer.Header().Set("Access-Control-Expose-Headers", "Content-Disposition")
9    if c.Request.Method == "OPTIONS" {
10     c.AbortWithStatus(204)
11     return
12   }
13   c.Next()
14 }
15 
```

Code 3.4: CORS Implementation Example

In the above code, line 4, the server can choose which domain will be allowed to fetch the data. The wild card, `*` can be used if the server allows any domain.

Not Found Route

Gin also supports the method to set `Not Found` page if the user enters the unknown endpoint, Code 3.5.

```go

seapRouter.NoRoute(func(context *gin.Context) {
    context.JSON(http.StatusNotFound,
            gin.H{"message": "Page not found"})
})
```

Code 3.5: Not Found Route

3.4.5 Model, Object Relation Mapping, and Repository

As Go is not an object-oriented programming language, there is no class or object. Instead, Go use the struct. All the entities in the database (see Figure 3.1) can be simulated with the struct or the map. In the backend code, there is a schema constructed with structs, similar to the actual entity-relationship model.

Model Layer and Object Relation Mapping

The data access objects (DAO) and the data transfer objects (DTO) are stored in the model layer. The DAO is applied to query the database and DTO is used for transferring the data in the controller and services. Gorm supports the struct schema to be transformed into new tables in the database schema. Gorm can also validate and connect the struct schema with the existing table in the database.

Struct for each table is created and also some smaller structs with the required data are created. As mentioned in the previous section, section 3.3, some entities are connected with the foreign key and many-to-many, many-to-one relationships. For many-to-many relationships, entities are connected with the auxiliary table.

When constructing the struct for the table in the database, the column name and the variable inside the struct should be the same. If the table name and column name are not the same, Go still supports the tag to specify which variable should be mapped with which column. The variable in the struct with the `PascalCase` will be parsed into the column with the `snake_case`. For example, a column name in the database is `first_name` and the variable name in the struct should be `FirstName`. For a member table and the family table, the structs are constructed as follows:

```go
type Member struct {
  FirstName    string      `json:"firstName"`
  LastName     string      `json:"lastName"`
  Username     string      `gorm:"primary_key" json:"username"`
  Email        string      `json:"email"`
  CredentialId string      `json:"-"`
  RoleId       uint        `json:"roleId"`
  Role         dto.RoleDto `gorm:"references:RoleId" json:"role"`
  CreatedAt    time.Time   `json:"createdAt"`
  ModifiedAt   time.Time   `json:"modifiedAt"`
}

func (Member) TableName() string {
  return "member"
}

type Family struct {
  FamilyId   string    `gorm:"primary_key" json:"familyId"`
  FamilyName string    `json:"name"`
  FamilyInfo string    `json:"info"`
  FamilyIcon string    `json:"icon"`
  CreatedAt  time.Time `json:"createdAt"`
  ModifiedAt time.Time `json:"modifiedAt"`
}

func (Family) TableName() string {
  return "family"
}
```

Code 3.6: Member and Family Structs

If the struct name and the table name are not the same, the `TableName()` function of the struct should be overridden with the table name from the database. Sometimes, the size of the data sent as a response to the client from the server should be reduced by ignoring unnecessary fields in the struct. In that case, `json:"-"` can be added to the tag. In Java, that can be performed with `@JsonIgnore` annotation.

So far, there is no connection between the member and family structs even though there is a many-to-many relationship in the database schema. The new smaller struct is created for the member which has a connection with the family using `family_member`. The `MemberWithFamilies` struct represents the `member` table meanwhile `FamilyForMember` struct represents the `family_member` table. The variable called `Families` in a `MemberWithFamilies` is a list of `FamilyForMember` is storing the family data. There is no direct interaction between the table `member` and `family`. `member` table interacts only with the `family_member` table and the same theory goes for the `family` table as well.

```go
type MemberWithFamilies struct {
  FirstName string             `json:"firstName"`
  LastName  string             `json:"lastName"`
  Username  string             `gorm:"primary_key" json:"username"`
  Email     string             `json:"email"`
  Families  []FamilyForMember `gorm:"foreignKey:Username"
  ↪  json:"families"`
}

func (MemberWithFamilies) TableName() string {
  return "member"
}

type FamilyForMember struct {
  Username   string                `gorm:"primary_key" json:"-"`
  FamilyId   string                `gorm:"primary_key" json:"-"`
  Family     dto.FamilyDtoMember
  ↪  `gorm:"foreignKey:FamilyId;references:FamilyId"
  ↪  json:"families"`
  RoleId     int                   `json:"-"`
  MemberRole dto.RoleDto
  ↪  `gorm:"foreignKey:RoleId;references:RoleId" json:"familyRole"`
  CreatedAt  time.Time             `json:"addedAt"`
}

func (FamilyForMember) TableName() string {
  return "family_member"
}
```

Code 3.7: Member and Family Connection Struct

If the user would like to get the family list where the user is a member, the server will query the database by using `MemberWithFamilies` struct.

Repository Layer

The repository is a layer that communicates with the database. Gorm is a library that supports object relationship mapping. The repository layer is split into smaller layers, such as memberRepository, familyRepository, dutyRepository, etc, and a main database object which is a type of `*gorm.DB`. The whole repository layer is initialised with the `Init()` function that connects the database with the proper credential from the `.env` variables. The detail for database initialization is mentioned in the Code 3.8

```go

type seapDataCenter struct {
  db *gorm.DB
}

var dc dataCenter

func Init() {
  if dc != nil {
    return
  }
  dc = &seapDataCenter{}
  dc.connectDatabase()
}

func (d *seapDataCenter) connectDatabase() {
  dbDriver := os.Getenv("DB_DRIVER")
  dbHost := os.Getenv("DB_HOST")
  dbUser := os.Getenv("DB_USER")
  dbPassword := os.Getenv("DB_PASSWORD")
  dbName := os.Getenv("DB_NAME")
  dbPort := os.Getenv("DB_PORT")

  DB_URL := fmt.Sprintf(
    "%s:%s@tcp(%s:%s)/%s?charset=utf8&parseTime=True&loc=Local",
    dbUser, dbPassword, dbHost, dbPort, dbName)

  var err error
  d.db, err = gorm.Open(mysql.Open(DB_URL), &gorm.Config{})

  if err != nil {
    fmt.Println("Cannot connect to database ", dbDriver)
  } else {
    fmt.Println("We are connected to the database", dbDriver)
  }
}
```

Code 3.8: Database Initialization

In this repository layer, the database object is set to a private level and cannot be accessible outside of the package. The data encapsulation concept is applied in this layer and instead of exposing the database object to other layers, this repository just reveals the wrapper methods. It is very important to keep the database object private, otherwise, other layers can easily call the database object directly, which will lead to potential code smell. So, for example, if the service layer would like to query all the families associated with the user, then just implement the function inside the repository layer that returns all the families. The sample implementation is as follows:

```go
func (d *seapDataCenter) getAll(dest any) *gorm.DB {
    return d.db.Find(dest)
}

func (fr FamilyRepositoryImpl) GetAllFamilies() *[]dao.Family {
  var families []dao.Family
  dc.getAll(&families)
  return &families
}
```

Code 3.9: Private Database with Wrapper Functions

In the above Code 3.9, `getAll` method which has direct access to the database object `*gorm.DB` is set to private and a wrapper public function, `GetAllFamilies()` is dedicating the private `getAll()` function. In the wrapper function, other conditions can be added as well. In this way, the database object cannot be called outside the package and it gives strong data security.

```go
func (d *seapDataCenter) insertOne(dest any) *gorm.DB {
  return d.db.Create(dest)
}

func (fr FamilyRepositoryImpl) SaveNewFamily(family *dao.Family)
 ↪  error {
    // any condition to filter or analyse before making changes to
     ↪  the database.
    return dc.insertOne(family).Error
}
```

Code 3.10: Insert New Record Wrapper Function

Code 3.10 shows the wrapper function of inserting a new record to the database.

3.4.6 Controller and Service Delegation

The controller layer is for handling the http request, payload, and route parameter and forwarding it to the service layer. The service layer is a collection of business logic and communicates with the database through the repository layer. The service layer is also a bridge between controller and repository, in another way, client and database.

Controller Layer

The controller layer is a collection of functions for the interaction with the http request. All the functions in the controller layer are assigned with the specific Gin route 3.2. This layer is also made up of many smaller controllers with a single responsibility. Each controller takes 1 parameter of type `*gin.Context`.

Requests coming to the server are intercepted by the middleware and only after the middleware has approved, the requests come into controller functions. The controller layer is specialised only for handling the request, parsing the route parameter, and query parameter, calling the service layer, returning the value from the service layer, and changing the response header and response status code. This layer does not contain any business logic and it is a stateless controller.

For example: `GetAllFamilies(*gin.Context)`, Code 3.11, is a controller function part of the family model. Inside `GetAllFamilies()`, the `GetAllFamiliesResponse()` from the service layer has been invoked and the response header and status code are adjusted if needed. This controller is a second layer after the middleware layer.

```go
func (fc *familyControllerImpl) getAllFamilies(context *gin.Context)
↪  {
  response, err := fc.fs.GetAllFamiliesResponse()
  if err != nil {
    context.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
    return
  }
  context.JSON(http.StatusOK, response)
}
```

Code 3.11: Stateless Controller Function

Some controllers need to parse the input request payload into the proper model so that the payload can be manipulated in the service layer. Mostly, the controller function responds to the request with JSON data. However, there are still some endpoints to upload or download the files.

```go
func (dc *dutyControllerImpl) saveNewDuty(context *gin.Context) {
    var input dao.Duty
    if err := context.ShouldBindJSON(&input); err != nil {
        context.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
        return
    }
    savedGrade, err := dc.ds.SaveNewDutyResponse(input)
    if err != nil {
        context.JSON(http.StatusBadRequest, savedGrade)
        return
    }
    context.JSON(http.StatusOK, savedGrade)
}
```

Code 3.12: Payload Parsing Controller Function

```go
func (cc *CDNControllerImpl) saveGivenFiles(context *gin.Context) {
  dutyId := context.Param("dutyId")
  err := context.Request.ParseMultipartForm(10 << 20)
  if err != nil {
    context.JSON(http.StatusBadRequest, gin.H{"error": "Failed to parse multipart form data"})
    return
  }

  files := context.Request.MultipartForm.File["files"]

  err = cc.ds.CreateGivenFiles(files, dutyId)
  if err != nil {
    context.JSON(http.StatusInternalServerError, gin.H{"error": "Failed to create assignment and upload files"})
    return
  }

  context.JSON(http.StatusOK, gin.H{"message": "uploaded."})
}
```

Code 3.13: Upload File Controller Function

For uploading files from the client to the server, the multipart form-data is parsed inside the controller function and forwards the files and their data to the service layer, `context.Request.MultipartForm.File["files"]`. The sample implementation for parsing the multipart form is written in Code 3.13.

```go
func (cc *CDNControllerImpl) downloadGivenFile(context *gin.Context)
 ↪  {
  dutyId := context.Param("dutyId")
  fileId := context.Param("fileId")

  filePath, err := cc.ds.GetGivenFilePath(dutyId, fileId)
  if err != nil || len(filePath) == 0 {
    context.JSON(http.StatusInternalServerError, gin.H{"error":
     ↪  "Failed to retrieve file path"})
    return
  }
  context.Header("Content-Disposition", "attachment;
   ↪  filename="+filepath.Base(filePath))

  context.File(filePath)
}
```

Code 3.14: Download File Controller Function

For downloading files from the server, the file path to be downloaded is retrieved from the service layer. `context.File(filepath)` is the statement for writing a file stream to the client. All the endpoints for file upload and download are protected with their respective middleware in the section 3.4.4.

Service Layer

This layer is a collection of business logic and algorithms. The filtering and preparing of the response data is done in this layer. Upon getting called by the controller, the raw data is retrieved from the database through the repository layer prepared according to the requirements and returned to the controller. Sometimes, two tables have a many-to-many relationship and the data to be returned for that relationship is prepared and parsed in this layer. For uploading files to the server, the file is saved in the machine and insert the file metadata into the database. For downloading files from the server, the file ID or file path is queried from the database and the file is read and returned to the controller. The example implementation for two different services, Code 3.15 shows the normal service implementation and Code 3.16 explains the business logic to prepare the response data.

```go
func (ds dutyServiceImpl) GetMyGradingResponse(dutyId string,
 ↪  username string) (dto.Response, error) {
  var grading *dao.Grading = &dao.Grading{
    DutyId:   dutyId,
    Username: username,
  }
  err := ds.dr.GetGradingByStructCondition(grading, grading)
  if err != nil {
    return  nil, err
  }
  return *grading, nil
}
```

Code 3.15: Normal Service Fuction

```go
func (ds dutyServiceImpl) GetSubmittedFilePath(dutyId string, fileId
 ↪  string, username string, familyRole string) (string, error) {
  var subFile *dao.SubmittedFile = &dao.SubmittedFile{
    FileId: fileId,
  }
  err := ds.dr.GetSubmittedFileById(subFile)
  if err != nil { return "", err }

  var grading *dao.Grading = &dao.Grading{
    DutyId:    dutyId,
    GradingId: subFile.GradingId,
  }
  err = ds.dr.GetGradingByStructCondition(grading, grading)
  if err != nil {
    return "", err
  }

  if familyRole == "tutee" && username != grading.Username {
    return "", errors.New("unauthorized file access")
  } else { username = grading.Username }

  return util.GetSubmittedFileAbsolutePath(dutyId, username,
   ↪  subFile.FilePath), nil
}
```

Code 3.16: File Download Service Function

3.5 SEAP WebUI with Angular and TypeScript

The SP WebUI is developed with the Angular framework using TypeScript. It is a perfect framework for people who want to use the software engineering approach in frontend development. Google develops it and has well-defined libraries and a good community online. Many UI libraries support good UI components with different themes. However, the goal was to keep the project clean and simple. That is why the Tailwind CSS library is the only third-party library used in this project.

3.5.1 Architecture

This project solely depends on the Rest API server and can be named as the integration of the Rest API server. This project is also built with layered architecture, unlike the backend project, only View and Service layers are included. The Rest API server provides the JWT token authentication which will be stored in the session storage on the client side. Once again, this is not the recommended solution and storing tokens in the frontend highly raises security concerns. The best solution is to use the proxy server to handle the token rotation if the Rest API server supports the rotational token authentication.

The service layer is for sending the request to the Rest API server and the view layer calls the methods from the service layer and renders the UI components.

3.5.2 Guards, Routing and HTTP Interceptor

Angular supports the routing configuration called Angular Router. Inside that Angular Router, the route information and configuration can be specified, for example, for a route `/main`, render the component `MainComponent` from the view layer. In each route information, the guard can also be set up. Guard means that if a user enters a route `/main`, the `MainComponent` is not rendered directly, instead, the route is redirected to a guard function where the authorisation and verification are done.

Guards for Authentication[19]

Overall, there are two scenarios in terms of authentication, when the user logged in and when the user has not logged in. If the user is logged in, some endpoints must not be rendered and should be redirected to the main page. For a logged-in user, the `login` endpoint and `signup` endpoint are not allowed to be rendered and For a user that has not logged in, only the `login` endpoint and `signup` endpoint should be rendered. `AfterLoggedinGuard` for the logged-in user and `AuthGuard` for the user without logging in, are implemented in the service layer.

The common method for validating the JWT token with the Rest API server is written in the service layer. Code 3.17 explains the authentication mechanism and Code 3.18 specifies the guard function for the routing.

```
async isLoggedIn() {
    let token = sessionStorage.getItem('token');
    if (token == null) { return false; }

    let response = await firstValueFrom(
        this.http.get(this._loggedinURL, { observe: 'response' })
    ).catch((r) => {});

    if (response == null) {
        this.deleteToken();
        return false;
    }
    return response.status == 200;
}
```

Code 3.17: Validating JWT Service

```
export const AuthGuard: CanActivateFn = async (route, state) => {
    const authService = inject(AuthService)
    const router = inject(Router)

    if (!await authService.isLoggedIn()) {
        return router.createUrlTree(['/login'])
    }
    return true;
};
```

Code 3.18: Authentication Guard

Routing with Components[20]

When it comes to the routing, Three scenarios are needed to consider, the normal routing to the respective component, the default routing and the invalid routing.

The normal routing is a routing to the component and the default routing is a routing when the user enters the default route without any children, and the invalid routing is a routing to the component if the user enters the unknown route. Route configuration in Angular is quite simple, just creating the list of JSON objects with the following structure.

```
export const routes: Routes = [
    {
        path: '',
        redirectTo: pathBeforeLogin.login,
        pathMatch: 'full',
    },
    {
        path: pathBeforeLogin.signup,
        component: SignupComponent,
        canActivate: [AfterLoggedinGuard],
    },
    .
    .
    .
    {
        path: '**',
        component: PageNotFoundComponent,
    },
];

```

Code 3.19: Route JSON Object

Some routes can have the children routes which means the components the children routes are redirecting will be rendered inside the parent component. In Code 3.20, if the user enters the route `main/`, the `MainComponent` will be rendered together with the children default route, `DashboardComponent` inside. If the user enters the route `main/family`, the `FamilyContainerComponent` is rendered and the routes can be defined recursively.

When redirecting from one route to another, the route parameter or query parameter can be passed, so the components can communicate and transfer the data to each other. If the components has a parent-children connection, not only the data can be transferred but also the event triggering from parent component to child component and vice versa can be done.

```
{
    path: 'main',
    component: MainComponent,
    children: [
        {
            path: '',
            component: DashboardComponent,
        },
        {
            path: 'family',
            component: FamilyContainerComponent,
            children: [
                {
                    path: '',
                    component: FamilyComponent
                },
                {
                    path: 'new-family',
                    component: CreateNewFamilyComponent,
                },
            ],
        },
    ]
}
```

Code 3.20: Parent Children Route

HTTP Interceptor[21]

The Rest API uses the JWT authentication and Angular supports something similar to the middleware implemented in the Rest API server. It is called `HttpInterceptor` which intercepts the HTTP protocol for `@angular/common/http` library. Whenever the http request is sent to the Rest API server, this interceptor will be intercepted and add necessary modifications. It is called HTTP Request Interceptor. In this frontend, the interceptor is written in the `TokenInterceptorService` to add the authorization header with the bearer tokens. There is also a possibility of implementing the HTTP Response Interceptor as well. In this way, whenever the http request is sent to the Rest API server, the authorization header does not need to be added and the `TokenInterceptorService` will do its job.

```
export class TokenInterceptorService implements HttpInterceptor {

    constructor(private _injector: Injector) { }

    intercept(req: HttpRequest<any>, next: HttpHandler):
     ↪  Observable<HttpEvent<any>> {
        let authService = this._injector.get(AuthService)
        let tokenizedReq = req.clone({
            setHeaders: {
                Authorization: 'Bearer ' + authService.getToken()
            }
        })
        return next.handle(tokenizedReq);
    }
}
```

Code 3.21: HTTP Request Interceptor

3.5.3 Service Layer

The service layer is responsible for sending the request to and receiving the response from the Rest API server. Instead of exposing the HTTP services to the view layer directly, it is always better to encapsulate and restrict services with functions. The service usually is a location for the business logic. Here in this project, all the business logic is already handled by the backend Rest API and this WebUI is just an integration based on the existing Rest API. If the response is not aligned with the component rendering, the service layer should handle the parsing of the response and prepare the data to be ready for use by the view layer.

```
private _myFamiliesURL = ENDPOINT + 'my/families';
getMyFamilies() {
    return this.http.get<any>(this._myFamiliesURL);
}
```

Code 3.22: Service Function

3.5.4 View Layer

In the view layer, the services are injected through the constructor upon initialization. Most components use the `Router` and the `ActivatedRoute` object for route manipulation. Sometimes, if the components need the data from the Rest API, the associated services are also injected.

Component and HTML Template

Each page of the WebUI (see subsection 2.4.1) is developed with the component. Every component implements the `OnInit` function for instantiation and getting the data from the Rest API through the service layer. In `ngOnInit()` function, the service functions are called and stored in the member variable. The component has some functions to trigger the event from the HTML and store the state of the HTML template.

The directives and `ngIf` are used in the HTML template to control the authorisation. The Rest API has already implemented the authorisation mechanism based on the user role and it would be better to control UI components as well.

Data Refreshing

Angular supports the rendering based on changes. If the HTML element is associated with the variable in the component, whenever there is a change in that variable, the HTML element is also rendered again. The `refreshData()` function calls the methods from the service layer and assigns them to the member variables. Whenever the data should be refreshed based on button actions or other events, the `refreshData()` can be called.

File Uploading

HTML supports the file uploading mechanism which is the input element with the `file` type. There are two possible methods for uploading the files. Both methods have advantages and disadvantages.

First, the user selects and attaches the files, the files are stored in the member variables and when the user clicks the `submit` or `upload` button, the files from the member variables together with other form data are sent to the Rest API with the post request. For the first method, the files are accumulated in the member variable and when the user clicks the `submit`, sending the post request to the Rest API server will take longer. If the user attaches the files and does not happen to submit the form, the files in the member variables will be discarded and the files will not reach the server.

The second scenario is that when the user attaches the file, the file is uploaded to the Rest API server. In the opposite of the first method, the files will reach the server as soon as the user attaches them. If the user cancels the form to submit, then the server has already accumulated the files and later, the server will become messed up and the storage consumption will get bigger.

In this project, the first concept is applied to the file uploading.

File Downloading

```
this._familyService.downloadGivenFile(famId, dutyId,
↪  fileId).subscribe({
    next: (res) => {
        if (res.body == null) return;
        const contentDispositionHeader = res.headers.get(
            'Content-Disposition'
        );
        const filename = contentDispositionHeader
            ? contentDispositionHeader.split(';')[1].trim()
                .split('=')[1]
            : 'file';

        const blob = new Blob([res.body], { type: res.body.type });
        const blobUrl = window.URL.createObjectURL(blob);

        const link = document.createElement('a');
        link.href = blobUrl;
        link.download = filename;

        link.click();
        window.URL.revokeObjectURL(blobUrl);
    },
    error: (err) => {
        console.log(err);
    },
});
```

Code 3.23: File Download Function

Rest API supports the file downloading. The files to be downloaded are matched with the file ID in the route parameter. The authorisation is already handled in the backend and every request to download the file needs to include the JWT token in the request header. The file can be received as an `arraybuffer` or as a `blob`. The filename is passed in the `Content-Disposition` header of the response. The response body is parsed into `blob` object and stored in the `Window Object URL`.

The HTML anchor element is created in the memory and assigned to the variable. The `href` - hypertext reference is set to the URL generated before. The click event of the anchor tag is triggered and the file will be downloaded automatically. The object is revoked from the window once the download has been done. The sample implementation is shown in the Code 3.23.

3.6 SEE with Go Routine

3.6.1 Problem Statement

The SEAP has an automated execution system for the submitted file. The engine is also implemented with the Go programming language. Upon triggering the execution, all the files are executed with the selected plugins. There are many programming languages and many grading systems. The plugins are executed according to the created duty. If the duty for the Java assignment is created, then the Java SEE Plugin should be selected and all the submitted files will be executed by the Java SEE Plugin.

Different subjects have different syllabi and different assignment systems. In programming, some assignments just need the executed output and in other assignments, the implementation should also be examined. If the framework is fixed with only 1 feature such as executing the Java program only, then the duty cannot be published for the other programming languages. If the duty is related to Python and the framework supports Java only, then the implementation of the framework needs to be modified and go through the development process just for executing the Python files. If there is another programming language, then the framework needs to be modified again.

3.6.2 Script Execution Engine Implementation

The SEE is an engine connected with the SEAP. SEE supports plugin development and SEAP can use the SEE to execute the plugin. If the duty is related to

Java, then the plugin for that duty can be developed. If there is a new programming language, then a new plugin has to be developed for that. When developing the plugin, the SEE has its own implementation structure and standard library. Every subject has different syllabi, so, the plugin for each subject can be implemented and used to execute the duty submitted by the students. In this way, instead of modifying the framework again and again for new programming languages, the plugin can be developed with a small amount of code and without touching the structure or the code of the implementation.

Plugin Loading

The `engine` package has the `Init()` for setting up the plugins. The plugins are located in the `plugins` package. When the engine has initialized, the plugins inside `plugins` package are parsed dynamically and stored in the local variable. Fortunately, Go has a library called `plugin` which is part of the standard library of Go and it supports dynamic loading of Go code during compilation. The `.go` files are parsed into the shared object files `.so` and dynamically loaded into the engine. Normally, there is a warning and concern regarding the usage of `plugin` library to load unknown `.go` files. As this project has its plugin template and structure, it will not be a problem to use the `plugin` library.

The plugins are discovered by walking through the `plugins` package folder and parsed into `.so` file to be able to load into the engine. The parsing of `.go` file into `.so` file can be performed with this command.

```
go build -buildmode=plugin -o pluginA.so pluginA.go
```

Code 3.24: .so File Generation Command

The function `DiscoverAndRegisterPlugins` is responsible for searching and loading the plugins. The Code 3.25 is the code snippet for executing the command 3.24 and loading the `.so` file and register in the engine.

```go
cmd := exec.Command("go", "build", "-buildmode=plugin", "-o",
↪  pluginBinaryName, pluginGoName)

err := cmd.Run()
if err != nil {  return err }

p, err := plugin.Open(pluginBinaryName)
if err != nil { return err }

symbol, err := p.Lookup("NewPlugin")
if err != nil { return err }

newPluginFunc, ok := symbol.(func() lib.Plugin)
if !ok { return errors.New("unexpected type from plugin symbol") }

RegisterPlugin(entry.Name(), newPluginFunc)
```

Code 3.25: Dynamic Plugin Loading

Plugin Concurrent Execution

A duty is associated with only one plugin. All the submitted files are executed and analysed only by that plugin. So, if there are 10 submissions, then the plugin should be executed 10 times with different submissions in each execution. Go specialises in concurrency and parallel programming. Go use the Go runtime thread instead of the operating system thread. This means it will not be an issue if tens of thousands of threads are created and run at the same time. The maximum number of threads used in this execution is 4. Each thread will evaluate each submission and if a thread has finished with a submission, the same thread with another submission will be evaluated.

Before executing, the metadata of the submission such as file location, and input parameters are added to the `channel`, in other programming languages, Queue, and the 4 threads are initialised with `go` and added to the wait group. A function is constructed to execute the plugin and that function is called parallelly in 4 threads.

The function is iterating the channel where the submission's metadata are stored. That channel is consumed by 4 threads parallelly. The thread consumes the first element of the channel and executes it. If the execution is done, the thread consumes another element of the channel until the channel becomes empty. Once the channel is empty, all the threads have stopped and return to the parent function.

```go
func ExecuteDuty(pluginName, dutyDir string) error {
  entries, err := os.ReadDir(dutyDir)
  if err != nil { return err }

  inputFileCh := make(chan string, len(entries))

  for _, entry := range entries {
    eachUserDir := filepath.Join(dutyDir, entry.Name())
    inputFileCh <- eachUserDir
  }
  close(inputFileCh)

  maxThreads := runtime.NumCPU() - 2
  var wg sync.WaitGroup; var mu sync.Mutex

  for i := 0; i < maxThreads; i++ {
    wg.Add(1)
    go Worker(inputFileCh, pluginName, &wg, &mu)
  }

  wg.Wait()

  fmt.Println("All executions completed")
  return nil
}
```

Code 3.26: Plugin Concurrent Execution

Plugin Report File

Plugin execution should produce only one report file in a given directory. The plugin can execute the submitted files and can write anything to the report file. The report file is `HTML` format and is not exposed to be controlled by the plugin. The report file is controlled inside the `SEEStandardPluginLibrary` and if the plugin wants to write the report file, the exposed methods starting with `Report` keyword by the `SEEStandardPluginLibrary` can be called.

In this version, the styling of the report file cannot be controlled by the plugin,

3.6.3 Plugin Template and Implementation

Plugins are compiled into shared object files `.so` which has a high risk if the plugin implementation does not have proper structure. To have a uniform implementation, to reduce the amount of code, and to ease the effort of plugin development, the plugin implementation is structured with a proper template.

Plugin Interface

The `Plugin` interface is the footprint of the plugin implementation. Every plugin should create a struct that implements all the functions of the `Plugin` interface. The struct is similar to the class in other languages. The struct created for the plugin can have member variables and state representation. In this version, the plugin interface will only have a few methods and later on, there can be more methods according to new features and additional requirements.

```go
type Plugin interface {
    Initialize(string) error
    Execute() error
    Name() string
}
```

Code 3.27: Plugin Interface

Plugin Template

The plugin file extension is a usual `.go`. Every plugin should start with this `//go:build plugin` to exclude from the build of the main project. Plugin files are executed during runtime and it is not necessary to compile together with the main project. The plugin `.go` file should have the package name as `main`.

```go
//go:build plugin
package main
```

Code 3.28: Mandatory Code for Plugin

The struct should be constructed for each plugin. That struct should implement the `Plugin` interface, so all the methods inside `Plugin` interface are implemented. The struct should also be composited with the `SEEPluginCommonLibrary` so that there will not be duplicated code and it will be much easier to develop new plugins. In the `Initialize` method, it is required to call the `InitializeLibrary()` method from the `SeePluginStandardLibrary`. When implementing the `Close()` method, it is mandatory to invoke the `CloseLibrary()` method to shut down the resources.

The most important part is to create a new public function `NewPlugiin()` to return the plugin struct. Without that function, the plugin import and loading will not work. The complete plugin template is shown in Code 3.29.

```go
type PluginTemplate struct {
  lib.SeePluginStandardLibrary
  // member variables goes here.
}

func NewPlugin() lib.Plugin {
  return &PluginTemplate{}
}

func (p *PluginTemplate) Initialize(inputDir string) error {
  p.InitializeLibrary(inputDir)
  /*
    Member variable initialization goes here.
  */
  return nil
}

func (p *PluginTemplate) Execute(targetDir string) error {
  p.SetUsername(targetDir)
  /*
    Business logic goes here.
  */
  return nil
}

func (p *PluginTemplate) Close() {
  /*
    Close everything here
  */
  p.CloseLibrary()
}

func (p *PluginTemplate) Name() string {
  return "PluginTemplate"
}

....
```

Code 3.29: Plugin Struct and Library Composition

3.6.4 Standard Library for Plugin

The development of the plugin is supported by the plugin standard library. Every plugin must follow a well-defined structure and template. If the same functionality is used in many plugins, that functionality can be integrated into the standard library. The purpose of the plugin standard library is to reduce the duplicated code and to increase the ease of plugin development. If the plugin standard library is easy to use and understandable, a developer who knows the `Go` syntax can easily develop a plugin.

A plugin is run concurrently, which means, multiple threads execute the same plugin at the same time. The implementation of the plugin should be concurrent and safe. Therefore, the functions that are essential and useful are developed as thread-safe in the standard library. The list of API methods by the `SeePluginStandardLibrary` are

- ReadFileConcurrentlySafe(string) ([]byte, error) - section 3.6.4
- ReadJSONFileAsStruct(string, any) error - section 3.6.4
- ReadDirectory(string) []os.DirEntry - section 3.6.4
- ReadProgrammingFileWithoutComment(string, string, string, string) ([]string, error) - section 3.6.4
- WriteLinesToFile([]string, string) error - section 3.6.4
- CreateAndWriteFileInTemp(lines []string, fileName string - section 3.6.4
- GetNewTemporaryDirectory() string - section 3.6.4
- ExecuteCommandWithTimeout(...string) (string, string, error) - section 3.6.4
- ReportAddHTMLTable(*[][]string) - section 3.6.4
- ReportAddMiniHeader(string) - section 3.6.4
- ReportAddParagraph(string) - section 3.6.4
- ReportAddHorizontalBar() - section 3.6.4
- ReportAddMiniHeaderAndParagraph(string, string) - section 3.6.4
- util package of SEAP - section 3.6.4

The API exposed by the `SeePluginStandardLibrary` struct are indicated with the method signature and the description as follows:

`ReadFileConcurrentlySafe(string) ([]byte, error)`

This method takes the file path as a string reads that file and returns the byte array. This method is also represented as a wrapper method of the `os.ReadFile()` function. The difference is the thread-safe and error-handling features. Two variables, `var fileReadMutexes = make(map[string]*sync. Mutex)` and `var fileReadMutexesMutex sync.Mutex` , are declared as a global cache to store the mutex of each file to be open. `ReadFileConcurrentlySafe` function also returns the empty byte array if the file path does not exist.

`ReadJSONFileAsStruct(string, any) error`

This function takes the file path as a string and the struct object to be transformed from the JSON file. This function calls the `ReadFileConcurrentlySafe()` function to read the JSON file and use JSON binding with the given struct. If there were any error, that error will be returned. If there is no error, the second parameter will be assigned with the valid struct representing that JSON file.

`ReadDirectory(string) []os.DirEntry`

This function is a wrapper function for the `os.ReadDir()` function which returns the empty list of `os.DirEntry` if the file path does not exist.

`ReadProgrammingFileWithoutComment(filePath string, singleLineComment string, multiLineStart string, multiLineEnd string) ([]string, error)`

Sometimes, in the plugin, the submitted files have to be read and analysed. In these cases, while reading the file, it is better to ignore the comment in the submitted file. Different programming language has different syntax for the comment of single line or multiline. Therefore, the parameters. `singleLineCommnet`, `multilineStart`, and `multiLineEnd`. The result string read from the file is returned as a list of String together with the error.

`WriteLinesToFile(lines []string, filePath string) error`

This function takes two parameters, the list of strings which are lines to be written in the file and the file path. If the file path does not exist, it will create a new file. The file path will be created even if the directory does not exist. This method is not a concurrent safe method, therefore if many threads invoke this function, it would occur unnecessary scenario.

`CreateAndWriteFileInTemp(lines []string, fileName string) string`

The plugin needs to create a new temporary file which will be deleted at the end of the plugin execution. This function takes two parameters, the list of strings, lines, and the file name to be created in the temporary directory. The temporary file with the given name will be created and written and the file path will be returned. The temporary directory is retrieved from the environment variable from the `util` package.

It is the consumer's responsibility to open and read the temporary file.

`GetNewTemporaryDirectory() string`

If the plugin needs a new temporary directory where the plugin can save some files or wants to cache the resources, this function can be applied.

`ExecuteCommandWithTimeout(command ...string) (string, string, error)`

Regarding the operating system command execution, regardless of bash or shell, this function will invoke the library `exec.Command` which supports multiple methods of command execution differently, such as synchronous execution, asynchronous execution and retrieving the output and error. This `ExecuteCommandWithTimeout` will configure the timeout and execute the given commands wait for the execution to be finished and retrieve the output. If the timeout occurs, the output before termination of the execution will be returned.

`ReportAddHTMLTable(tableHeadAndData *[][]string)`

If the plugin wants to add the table to the report file, this function can be called. In this initial version of SEE, only a few features will be included, so only the normal table (rows and columns) can be added and there is no subtable in the table cells.

`ReportAddMiniHeader(header string)`

This function will add the header to the report file.

`ReportAddParagraph(para string)`

This function will add the paragraph to the report file.

`ReportAddHorizontalBar()`

This function will add the horizontal bar to the report file.

`ReportAddMiniHeaderAndParagraph(header string, param string)`

This function will add the combination of the header and the paragraph to the report file.

`util` **package**

The SEAP and SEE have the same functionality in some areas. Therefore, the `util` package is created to reduce the code smell. The `util` package is consumed by both SEAP Rest API and the SEE and can also be applied in the plugin development.

The `SEEStandardPluginLibrary` is created to help developers implement the plugin easily. In this initial version of SEE, there are many weak points and a lack of useful features in terms of user and developer experience. The styling of the HTML report file cannot be controlled and only the elements supported by the `SEEStandardPluginLibrary` can be added to the report file. In the future, the SEEStandardPluginLibrary will be full of features, useful and easy access and aim to publish as a third party `Go` library to be used by other developers in other projects. For now, let us move on with the unit test of the SEAP and SEE.

3.7 Testing

Testing is the most important phase after the development. There are two popular testing types, integrated testing and functional testing. The unit testing is the first level of the testing after the development. It is also the developer's responsibility to come up with the unit testing together with the implementation. It is a win for the tester if the source code crashes on the test case.

Integrated testing means testing with the dummy data to verify the expected results. The dummy data is passed into the source code and the result returned by the source code is verified with the expected result. Integrated testing ensures that the source code is performing correctly without any weakness. The dummy data must cover all possible scenarios.

Functional testing means the functionality of the project in a real-world scenario (for example: UI testing, verifying if the button is working as expected). Functional testing also checks if the features are working according to the requirements. In this project, functional testing is not implemented as this is the initial version of the project.

The black-box and white-box testing are the testing concepts applied in the unit testing. This SEAP and SEE came with the unit testing package which examines the implementation of the source code, a combination of three concepts, integrated testing, and black-box and white-box testing. The unit test is constructed for each layer of the architecture, controller, service and repository, and additionally for the SEE as well. The implementation of the test cases and how the unit tests are written is specified in the subsection 3.7.1.

3.7.1 Unit Testing

The unit testing is a standalone isolated testing for the implementation. The unit testing for the SEAP Rest API is separated into different packages for different layers. All the test package are located under the `test` directory. The test cases can be executed with the `go test module_name`. The full command for the test case execution is

```
go test -v -cover ./test/controller
```

Code 3.30: Command for Test Case Execution

If the `-v` flag is included, all the details for each test case will be shown. `-cover` flag will show the coverage of the test cases comparing with the source code. The last parameter of the command must be the test package directory relative to the `main.go` entry.

Each layer of the design are constructed with the interface which allows the polymorphism principle. The mock layer is established and injected to the target layer in each test case. The unit test package for each layer is specified below.

Controller Layer Testing

In the controller test package, the test cases are separated into multiple file according to the source code. The controller methods are taking only one parameter which is the `gin.Context`. These controller methods are consumed inside each test case. A controller method can have one or more test cases based on the possible scenario. Sometimes, there are three test cases just for one controller method.

As the controller method are called inside the test case, the `gin.Context` needed to be mock. The benefit of using `Gin` framework is that `Gin` has API functions for testing. The dummy `gin.Context` can be created with the `gin.CreateTestContext(http.ResponseWriter)` function. The dummy http request is created and sent to the context. After that, the controller method is invoked.

The controller is connected with the service layer. In the controller, everything related to the HTTP requests is handled and consumes the functions from the service layer to return the response to the client. Therefore, it is also necessary to create a mock service for the controller. Due to the maintainability of the source code, the mock service can be easily passed into the controller layer, similar to the dependency injection.

The sample implementation of the mock service for the `MemberService` is as follows:

```go
type MockMemberService struct {
  Data dto.Response
  Err  error
}

func (m *MockMemberService) GetMemberByIdResponse(s string)
 ↪  (dto.Response, error) {
  return m.Data, m.Err
}
.
.
.

func (m *MockMemberService) GrantRoleResponse(s string, i int)
 ↪  (dto.Response, error) {
  return m.Data, m.Err
}
```

Code 3.31: Mock Member Service

The `MockMemberService` must implement the `MemberService` interface so that `MockMemberService` object can be injected into the controller to replace the actual `MemberService`.

The controller method for the `GetAllMember` is shown below,

```go
func (mc *memberControllerImpl) GetAllMembers(context *gin.Context)
↪  {
  size, page := paginated(context)
  response, err := mc.ms.GetAllMembersResponse(size, page)
  if err != nil {
    context.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})
    return
  }
  context.JSON(http.StatusOK, response)
}
```

Code 3.32: Sample Controller Method

The unit test for the `GetAllMember`, Code 3.32, contains two test cases, the success scenario and the error scenario. The context to be passed to the controller function is mocked and the controller is initialized with the mock service layer which is injected.

In the Code 3.33 test case, the context is created with the custom http writer and set the necessary parameter. The dummy service layer is initialized with the `MockMemberService` (see Code 3.31), and some dummy data are injected. After that, the controller object is invoked with the `NewMemberController()` function with the injection of `MockMemberService`. The target controller method is called with the previous created context. After consuming the controller function, the assertion is implemented. The http status has to be verified and then the returned response by the controller function is asserted with the expect value.

```go
func TestGetAllMembers_Success(t *testing.T) {
  w := httptest.NewRecorder()
  context, _ := gin.CreateTestContext(w)
  context.Request = httptest.NewRequest(http.MethodGet, "/members",
  ↪  nil)
  context.Set("size", "10")
  context.Set("page", "1")

  dummyMembers := []dao.Member{
    {Username: "johndoe", FirstName: "John", LastName: "Doe"},
    {Username: "janedoe", FirstName: "Jane", LastName: "Doe"},
  }

  mockService := &service.MockMemberService{
    Data: dummyMembers,
    Err:  nil,
  }
  controller := controller.NewMemberController(mockService)

  controller.GetAllMembers(context)

  if context.Writer.Status() != http.StatusOK {
    t.Errorf("Expected status code %d, got %d", http.StatusOK,
    ↪  context.Writer.Status())
  }

  var response []dao.Member
  err := json.Unmarshal(w.Body.Bytes(), &response)
  assert.NoError(t, err)

  assert.Equal(t, dummyMembers, response)
}

```

Code 3.33: Sample Test Case of Success Scenario for Controller

The Code 3.34 is the test case for the error scenario of the controller method. The test case is implemented in the same approach with the Code 3.33, except the differet mock service object. Here, the `MockMemberService` struct is initialized with the `nil` in the `Data` and the error object in the `Err`. The response status by the controller method is asserted with the `http.StatusBadRequest` which is the http code 400.

```go
func TestGetAllMembers_Error(t *testing.T) {
  context, _ := gin.CreateTestContext(httptest.NewRecorder())
  context.Request = httptest.NewRequest(http.MethodGet, "/members",
   ↪  nil)

  mockService := &service.MockMemberService{
    Data: nil,
    Err:  errors.New("mock error"),
  }
  controller := controller.NewMemberController(mockService)
  controller.GetAllMembers(context)

  if context.Writer.Status() != http.StatusBadRequest {
    t.Errorf("Expected status code %d, got %d",
     ↪  http.StatusBadRequest, context.Writer.Status())
  }
}
```

Code 3.34: Sample Test Case of Error Scenario for Controller

Normally, the test case for the controller are quite similar except the mock service layer. There still exists the extrarodinary test cases for the file downloading controller. The controller function for the file downloading is mentioned in the Code 3.14. In the test case, instead of creating the dummy context, the dummy gin web server is initialized and assigned with the target controller function. The http request is sent to that dummy web server and the response is asserted. When the server layer is mocked, the temporary file is created and the controller function will return that temporary file as a response.

```go
file, err := os.CreateTemp("", "mock-image-*.txt")
if err != nil {
    t.Fatalf("Failed to create temporary file: %v", err)
}
defer os.Remove(file.Name())
```

Code 3.35: Create Temporary File for Test Case

The Code 3.35 is the statement for creating the temporary file inside the test case.

```go
mockService := &service.MockDutyService{
    FilePath: file.Name(),
    Err:      nil,
}
cc := controller.NewCDNController(mockService, nil)
```

Code 3.36: Mock Service Injection

In the Code 3.36, the previously created temporary file path is passed into the `FilePath` variable of the `service.MockDutyService` and then, the `MockDutyService` object is injected into the controller.

```go
router := gin.Default()
router.GET("/your-endpoint/:dutyId/:fileId", cc.DownloadGivenFile)

w := httptest.NewRecorder()

req, err := http.NewRequest("GET", "/your-endpoint/123/123", nil)
if err != nil { t.Fatal(err) }

router.ServeHTTP(w, req)
```

Code 3.37: Dummy Web Server for Test Case

The Code 3.37 shows the dummy web server created with the `gin.Default()` and the target controller function is passed. The http request is initialized and invoked with the dummy web server.

```go
assert.Equal(t, http.StatusOK, w.Code)
assert.Equal(t, "blob", w.Header().Get("Content-Type"))
```

Code 3.38: Assertion of the Response by Dummy Web Server

The response by the dummy web server is verified in the Code 3.38.

Service Layer Testing

The service layer is a bridge between the controller and the repository.This layer retrieves the data from the repository and perform business logic and returns to the controller. The unit testing for the service layer is quite close to the unit testing of the imperative programming. There is no complicated scenario in the function. The service layer is based on the repository layer, therefore, it is necessary to have a mock repository layer for the unit testing.

```go
func (ms memberServiceImpl) GetMemberByIdResponse(id string)
 ↪  (dto.Response, error) {
  var member *dao.Member = &dao.Member{
    Username: id,
  }
  err := ms.mr.GetMemberByUsername(member)
  if err != nil {
    return BeforeErrorResponse(PrepareErrorMap(404, "Username does
     ↪  not exist.")), err
  }

  newResp := BeforeDataResponse[dao.Member](&[]dao.Member{*member},
   ↪  1)
  return newResp, nil
}
```

Code 3.39: Sample Function of the Service Layer

The Code 3.39 function is the service function for getting the member record by the id. This function is being consumed by the controller layer. When this function is called, the id of string type must be passed and the value will be returned if the record is found. The unit test for that service function can be as follows:

```go
func TestGetMemberByIdResponse_Success(t *testing.T) {
  mockRepo := &repository.MockMemberRepository{
    Member: &dao.Member{ Username: "mockUsername" },
    Err: nil,
  }

  ms := service.NewMemberServiceForTest(mockRepo, nil)

  response, err := ms.GetMemberByIdResponse("mockId")

  assert.Nil(t, err)
  assert.NotNil(t, response)
}
```

Code 3.40: Sample Test Case for the Service Function

In the Code 3.40, the mock repository is initialized and injected in creating the MemberService object. After that, the target function is invoked and the assertion is executed.

Repository Layer Testing

The repository layer is responsible for executing the query to the database. This is the layer that has a control over the database. `Gorm`, Go Object Relation Mapping, library is used to connect with the database. Similar to the service layer, repository layer is also made up of the interface and the struct.

```go
func (m MemberRepositoryImpl) GetAllMembers(offset, limit int)
 ↪  *[]dao.Member {
  var members []dao.Member
  dc.GetAllByPagination(&members, offset, limit, &dao.Member{},
   ↪  "Role")
  return &members
}
```

Code 3.41: Sample Function of the Repository

The Code 3.41 is the method of the `MemberRepositoryImpl` struct that implements the `MemberRepository` public interface. This `GetAllMembers` method is consumed by the service layer to retrieve the record from the database. `GetAllMembers` method calls the `GetAllByPagination` method of the `DataCenter` interface which has the private implementation of `seapDataCenter` struct. For the unit testing for this method, it is not necessary to know how the `seapDataCenter` is implemented.

```go
func TestGetAllMembers(t *testing.T) {
  mockDataCenter := &database.MockDataCenter{}
  repository.InitializeDataCenter(mockDataCenter)

  memberRepo := repository.MemberRepositoryImpl{}
  offset := 0; limit := 10
  result := memberRepo.GetAllMembers(offset, limit)

  assert.NotNil(t, result)
  assert.Equal(t, reflect.TypeOf(result).String(), "*[]dao.Member")
}
```

Code 3.42: Sample Test Case for the Repository Function

In the Code 3.42, the mock `DataCenter` struct is initialized and injected into the repository layer. After that, the `MemberRepositoryImpl` struct is initialized and the target method, codeGetAllMembers, is invoked. Finally, the results are verified with the assertion.

The testing phase is part of the software development lifecycle and this thesis covers the unit testing only. In the future, if the project is applied in the real-world, the SEAP must go through the integrated testing and functional testing phase. During the development, it is better to setup the continuous integration and continuous delivery, tracking with the version control system.

Conclusion

The SEAP and SEE were developed through many development phases, the analysis and research, the design and architecture, the implementation, the testing and the documentation. The SEAP is built as an enterprise application used in the real world and with a software development lifecycle applied in today's business.

The SEAP and SEE cannot replace human work for correcting assignments, however, the amount of workload and effort can be reduced. It would be time-saving that all the submitted files could be executed with just one click. It is quite impossible to produce software that covers all the aspects. That is why the plugin development feature is added, so the SEAP is open for extension but closed for modification.

Instead of creating different apps for the different programming languages, we just built one main platform and added the plugin extensible feature. Even though the SEAP and SEE have many security weak points in this version, all the recommended solutions and the cautions are mentioned in this thesis, so those solutions will be the future work to keep improving this SEAP.

As the development of AI is progressive, we cannot imagine how powerful this SEAP is combined with AI. This SEAP can be used for the Computer Science field and other fields such as the recruiting process in businesses. By developing a plugin that can analyse the submitted files, instead of compiling and executing, the SEAP can be used in other major studies in the educational field.

This thesis combines web development, software engineering, and automation. In software engineering, duplicated codes, and repeated functionality with different implementations or repetitions are forbidden. The SEAP applies the Open-closed principle (i.e. open for extension, closed for modification) at a software level by introducing the plugin development feature. Therefore, the SEAP also proved that software engineering principles can be used in coding and the actual world in our daily lives.

Future Work

All the difficulties I encountered while making this thesis were connected to making a detailed solution in a constrained timeframe. This section is intended to give a foreknowledge of the potential issue of the thesis work. It reveals some aspects of its creation story as well as what started it all. I aim to provide a deeper understanding of the context in which the thesis was conceived and executed.

This practical project has some issues regarding security. As mentioned in section 2.1, this software is aimed to reduce the human effort for assignment management with the automatic execution and generation of reports.

There are two main issues in terms of security. First, the authentication mechanism in this project is not complete enough. Second, the execution of the submitted files should be taken as a serious matter as the environment in which the execution took place. A user can submit any files so far and during execution, what if the user wrote a script that can bypass retrieve all the system information and attack the software? Moreover, the author noticed that the authentication mechanism used in this project completely lacks security.

The author already has a solution to prevent those from happening. The allocated timeframe for this project does not afford to adequately address all aspects of the application's architecture and implementation. So, the author could not implement the solution in the practical project, however, included the research in this documentation.

The author did a lot of research on the authentication method of Canvas Learning Management System, TMS and Neptun. All of them used the same authentication server which is Neptun. So, both Canvas and TMS can be entered with the Neptun credentials. Even though they are using the same authentication server, they have different authentication methods. Canvas and Neptun are storing the token in the cookie together with the CSRF token meanwhile TMS is storing the token in the local storage which is similar to this SEAP.

In SEAP, the current implementation applied a JWT token with the expiration but without the refresh token. The token is stored in the session storage or the local storage. Storing the token in the front end can give a chance to the attacker to gain access to the unauthorised data. There are two proposed methods by the author: to use the rotational JWT combined with a Backend-for-Frontend proxy server[22], or to store the JWT token in the cookies together with the CSRF token[16]. All the details are explained in the developer documentation (see chapter 3).

Another issue is related to the script execution. When executing the submitted file, the environment in which the execution took place should be isolated, otherwise, a user can easily retrieve all the system data. Of course, here, the containerization, docker, should be used to create a sandbox environment. Container isolation is still not enough to prevent the injection if the software meets the attacker who can do a memory attack. GVisor is the software developed by Google that can create a fake kernel for the container[23]. This is the solution. A combination of docker and gvisor could create a strongly isolated sandbox environment.

When the project reaches the production level, it is recommended to use the proper file storage and database such as an S3 bucket and RDS from Amazon Web Service[24, 25]. It is the author's goal to improve these features in this SEAP in the future and, surely, this SEAP will be a nice perfect secure software.

Acknowledgements

When developing this thesis, I encountered challenges and it was a tough development process. Starting from scratch, until producing the enterprise application, every step was challenging. First of all, I would like to thank my supervisor, Dr Zsók Viktória, who pushed me to be progressive during development and guided me with this documentation, and all the professors in the ELTE for teaching me to be able to develop this SEAP and SEE.

I would not be here studying at ELTE if I did not receive the Stipendium Hungaricum Scholarship. I would like to indicate my thankfulness and recognition to the foundation and the Hungarian Government that awarded me the scholarship and gave me a chance to establish a brighter future.

I extend my sincere appreciation to my manager and my mentor at Nokia Budapest, for allowing me to keep working on this thesis and for giving me a chance to grow. I would not become who I am today if my mentor from Nokia had not guided me and taught me how to be professional.

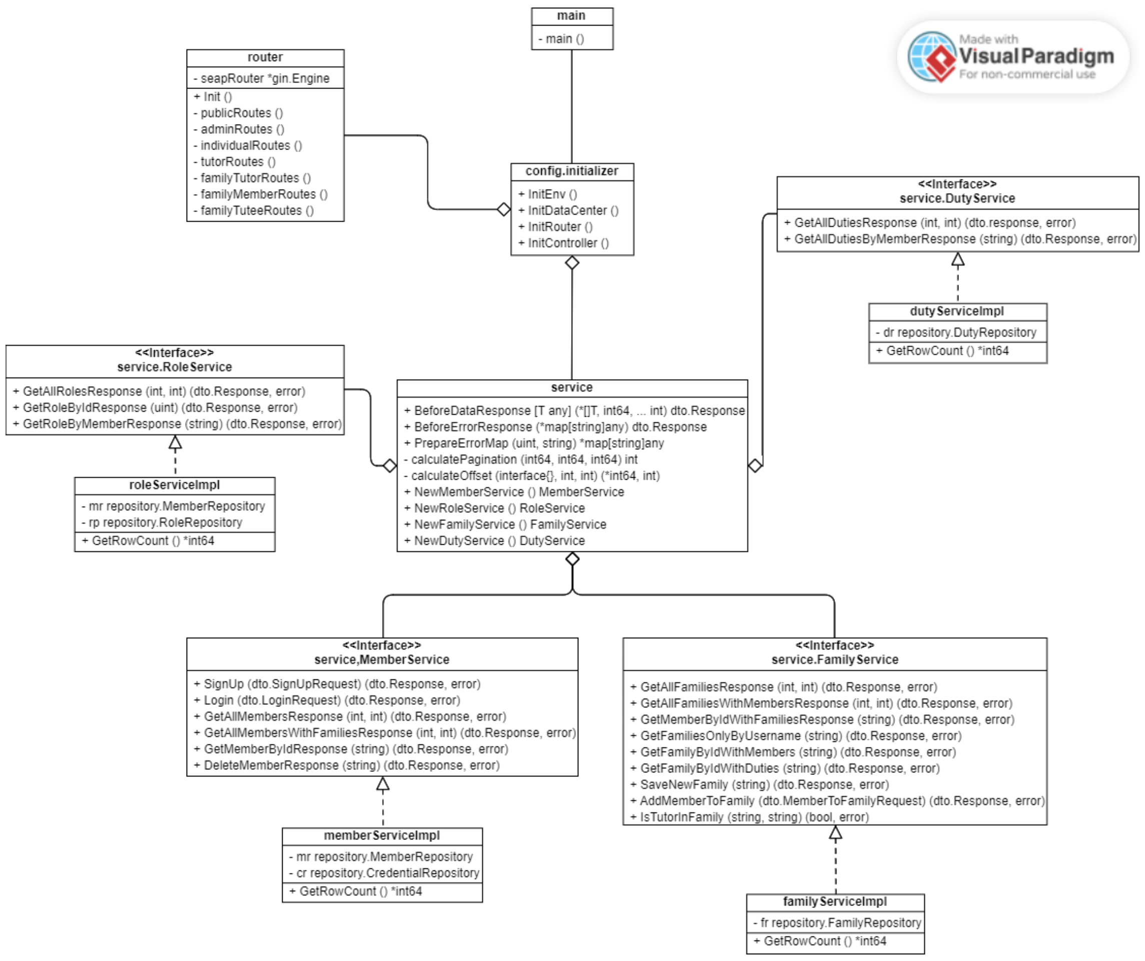
Here, I express my gratitude to my family, Kyi Shwin and Thida Win who supported me from my home country through my entire BSc education journey. I appreciate my parents for letting me choose this Computer Science career and respecting my decision without any doubts.

While it is impossible to individually acknowledge everyone, I express my gratitude here with this thesis to all who have contributed indirectly to this thesis completion and my education journey.

Appendix A

UML Diagrams

Service Layer UML Diagram



main
- main ()
router
- seapRouter *gin.Engine
+ Init ()
- publicRoutes ()
- adminRoutes ()
- individualRoutes ()
- tutorRoutes ()
- familyTutorRoutes ()
- familyMemberRoutes ()
- familyTuteeRoutes ()
config.initializer
+ InitEnv ()
+ InitDataCenter ()
+ InitRouter ()
+ InitController ()
<<Interface>>
service.DutyService
+ GetAllDutiesResponse (int, int) (dto.response, error)
+ GetAllDutiesByMemberResponse (string) (dto.Response, error)
dutyServiceImpl
- dr repository.DutyRepository
+ GetRowCount () *int64
<<Interface>>
service.RoleService
+ GetAllRolesResponse (int, int) (dto.Response, error)
+ GetRoleByIdResponse (uint) (dto.Response, error)
+ GetRoleByMemberResponse (string) (dto.Response, error)
roleServiceImpl
- mr repository.MemberRepository
- rp repository.RoleRepository
+ GetRowCount () *int64
service
+ BeforeDataResponse [T any] (*[]T, int64, ... int) dto.Response
+ BeforeErrorResponse (*map[string]any) dto.Response
+ PrepareErrorMap (uint, string) *map[string]any
- calculatePagination (int64, int64, int64) int
- calculateOffset (interface{}, int, int) (*int64, int)
+ NewMemberService () MemberService
+ NewRoleService () RoleService
+ NewFamilyService () FamilyService
+ NewDutyService () DutyService
<<Interface>>
service,MemberService
+ SignUp (dto.SignUpRequest) (dto.Response, error)
+ Login (dto.LoginRequest) (dto.Response, error)
+ GetAllMembersResponse (int, int) (dto.Response, error)
+ GetAllMembersWithFamiliesResponse (int, int) (dto.Response, error)
+ GetMemberByIdResponse (string) (dto.Response, error)
+ DeleteMemberResponse (string) (dto.Response, error)
memberServiceImpl
- mr repository.MemberRepository
- cr repository.CredentialRepository
+ GetRowCount () *int64
<<Interface>>
service.FamilyService
+ GetAllFamiliesResponse (int, int) (dto.Response, error)
+ GetAllFamiliesWithMembersResponse (int, int) (dto.Response, error)
+ GetMemberByIdWithFamiliesResponse (string) (dto.Response, error)
+ GetFamiliesOnlyByUsername (string) (dto.Response, error)
+ GetFamilyByIdWithMembers (string) (dto.Response, error)
+ GetFamilyByIdWithDuties (string) (dto.Response, error)
+ SaveNewFamily (string) (dto.Response, error)
+ AddMemberToFamily (dto.MemberToFamilyRequest) (dto.Response, error)
+ IsTutorInFamily (string, string) (bool, error)
familyServiceImpl
- fr repository.FamilyRepository
+ GetRowCount () *int64
Made with Visual Paradigm For non-commercial use

Figure A.1: Service Layer Design of SEAP Rest API

Repository Layer UML Diagram

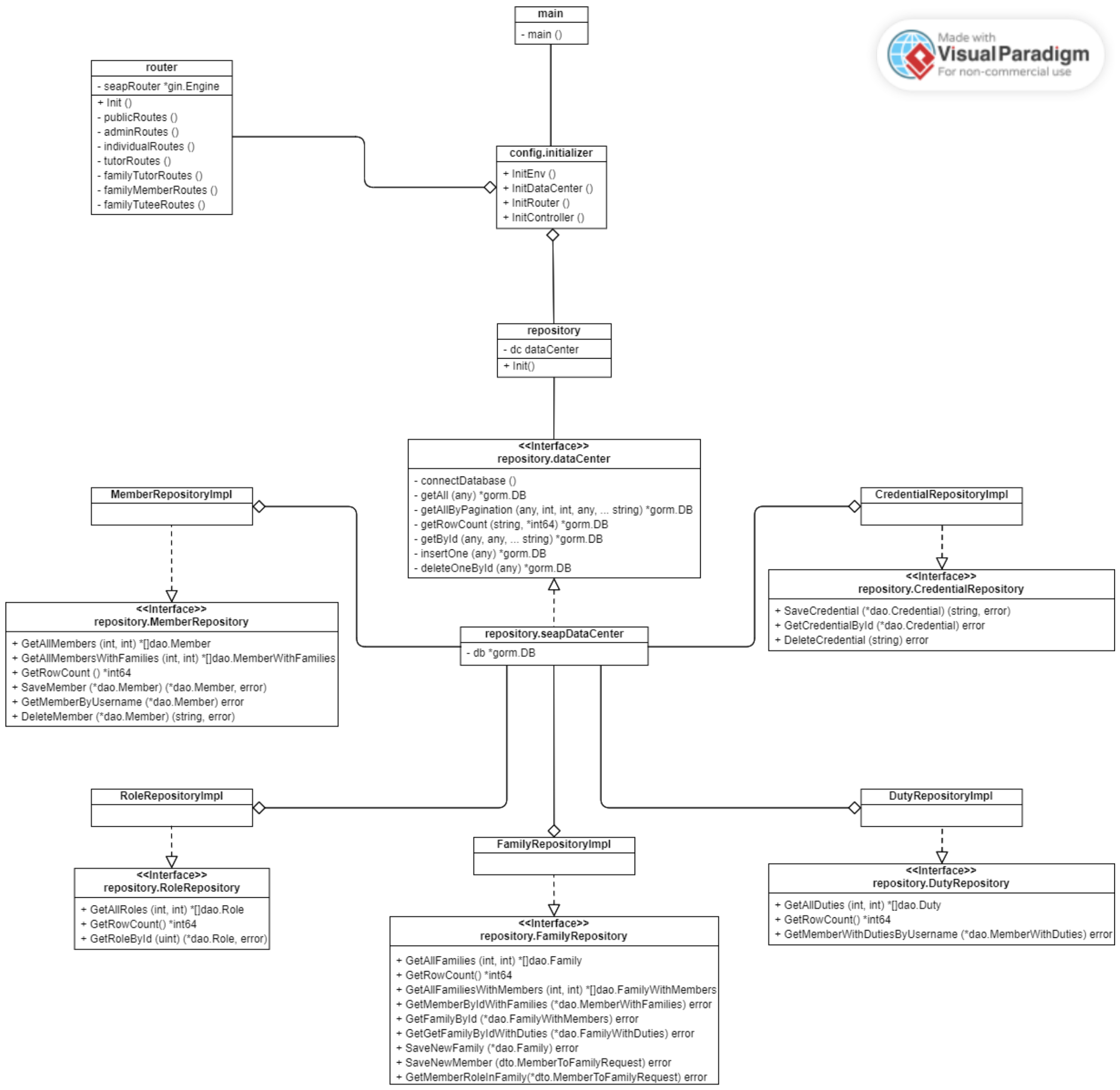

main
- main ()
router
- seapRouter *gin.Engine
+ Init ()
- publicRoutes ()
- adminRoutes ()
- individualRoutes ()
- tutorRoutes ()
- familyTutorRoutes ()
- familyMemberRoutes ()
- familyTuteeRoutes ()
config.initializer
+ InitEnv ()
+ InitDataCenter ()
+ InitRouter ()
+ InitController ()
Made with Visual Paradigm For non-commercial use
repository
- dc dataCenter
+ Init()
<<Interface>>
repository.dataCenter
- connectDatabase ()
- getAll (any) *gorm.DB
- getAllByPagination (any, int, int, any, ... string) *gorm.DB
- getRowCount (string, *int64) *gorm.DB
- getById (any, any, ... string) *gorm.DB
- insertOne (any) *gorm.DB
- deleteOneById (any) *gorm.DB
MemberRepositoryImpl
<<Interface>>
repository.MemberRepository
+ GetAllMembers (int, int) *[]dao.Member
+ GetAllMembersWithFamilies (int, int) *[]dao.MemberWithFamilies
+ GetRowCount () *int64
+ SaveMember (*dao.Member) (*dao.Member, error)
+ GetMemberByUsername (*dao.Member) error
+ DeleteMember (*dao.Member) (string, error)
CredentialRepositoryImpl
<<Interface>>
repository.CredentialRepository
+ SaveCredential (*dao.Credential) (string, error)
+ GetCredentialById (*dao.Credential) error
+ DeleteCredential (string) error
repository.seapDataCenter
- db *gorm.DB
RoleRepositoryImpl
<<Interface>>
repository.RoleRepository
+ GetAllRoles (int, int) *[]dao.Role
+ GetRowCount() *int64
+ GetRoleById (uint) (*dao.Role, error)
FamilyRepositoryImpl
<<Interface>>
repository.FamilyRepository
+ GetAllFamilies (int, int) *[]dao.Family
+ GetRowCount() *int64
+ GetAllFamiliesWithMembers (int, int) *[]dao.FamilyWithMembers
+ GetMemberByIdWithFamilies (*dao.MemberWithFamilies) error
+ GetFamilyById (*dao.FamilyWithMembers) error
+ GetGetFamilyByIdWithDuties (*dao.FamilyWithDuties) error
+ SaveNewFamily (*dao.Family) error
+ SaveNewMember (dto.MemberToFamilyRequest) error
+ GetMemberRoleInFamily(*dto.MemberToFamilyRequest) error
DutyRepositoryImpl
<<Interface>>
repository.DutyRepository
+ GetAllDuties (int, int) *[]dao.Duty
+ GetRowCount() *int64
+ GetMemberWithDutiesByUsername (*dao.MemberWithDuties) error

Figure A.2: Repository Layer Design of SEAP Rest API

Controller Layer UML Diagram

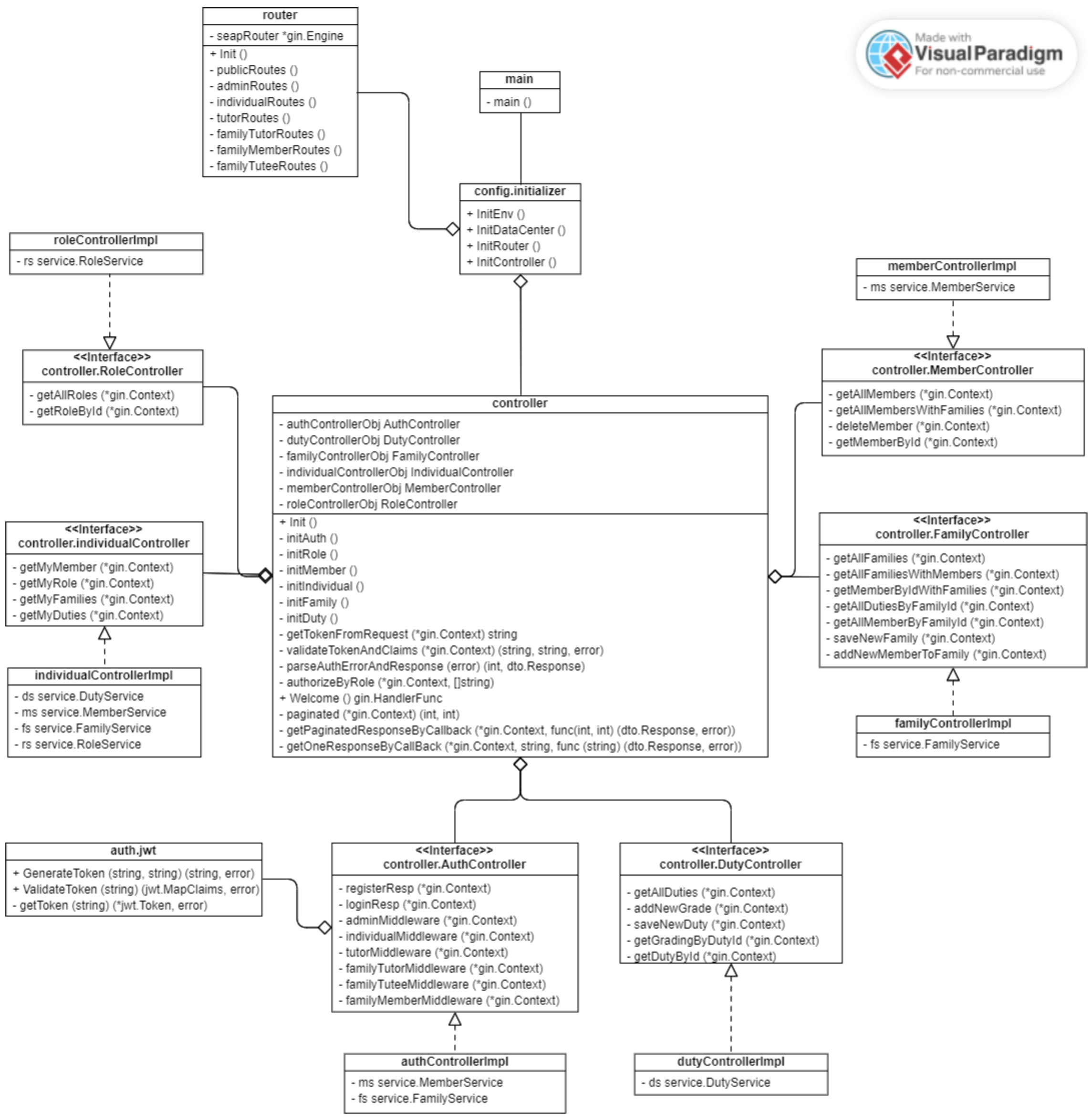

router
- seapRouter *gin.Engine
+ Init ()
- publicRoutes ()
- adminRoutes ()
- individualRoutes ()
- tutorRoutes ()
- familyTutorRoutes ()
- familyMemberRoutes ()
- familyTuteeRoutes ()
main
- main ()
config.initializer
+ InitEnv ()
+ InitDataCenter ()
+ InitRouter ()
+ InitController ()
roleControllerImpl
- rs service.RoleService
<<Interface>>
controller.RoleController
- getAllRoles (*gin.Context)
- getRoleById (*gin.Context)
memberControllerImpl
- ms service.MemberService
<<Interface>>
controller.MemberController
- getAllMembers (*gin.Context)
- getAllMembersWithFamilies (*gin.Context)
- deleteMember (*gin.Context)
- getMemberById (*gin.Context)
controller
- authControllerObj AuthController
- dutyControllerObj DutyController
- familyControllerObj FamilyController
- individualControllerObj IndividualController
- memberControllerObj MemberController
- roleControllerObj RoleController
+ Init ()
- initAuth ()
- initRole ()
- initMember ()
- initIndividual ()
- initFamily ()
- initDuty ()
- getTokenFromRequest (*gin.Context) string
- validateTokenAndClaims (*gin.Context) (string, string, error)
- parseAuthErrorAndResponse (error) (int, dto.Response)
- authorizeByRole (*gin.Context, []string)
+ Welcome () gin.HandlerFunc
- paginated (*gin.Context) (int, int)
- getPaginatedResponseByCallback (*gin.Context, func(int, int) (dto.Response, error))
- getOneResponseByCallBack (*gin.Context, string, func (string) (dto.Response, error))
<<Interface>>
controller.individualController
- getMyMember (*gin.Context)
- getMyRole (*gin.Context)
- getMyFamilies (*gin.Context)
- getMyDuties (*gin.Context)
individualControllerImpl
- ds service.DutyService
- ms service.MemberService
- fs service.FamilyService
- rs service.RoleService
<<Interface>>
controller.FamilyController
- getAllFamilies (*gin.Context)
- getAllFamiliesWithMembers (*gin.Context)
- getMemberByIdWithFamilies (*gin.Context)
- getAllDutiesByFamilyId (*gin.Context)
- getAllMemberByFamilyId (*gin.Context)
- saveNewFamily (*gin.Context)
- addNewMemberToFamily (*gin.Context)
familyControllerImpl
- fs service.FamilyService
auth.jwt
+ GenerateToken (string, string) (string, error)
+ ValidateToken (string) (jwt.MapClaims, error)
- getToken (string) (*jwt.Token, error)
<<Interface>>
controller.AuthController
- registerResp (*gin.Context)
- loginResp (*gin.Context)
- adminMiddleware (*gin.Context)
- individualMiddleware (*gin.Context)
- tutorMiddleware (*gin.Context)
- familyTutorMiddleware (*gin.Context)
- familyTuteeMiddleware (*gin.Context)
- familyMemberMiddleware (*gin.Context)
<<Interface>>
controller.DutyController
- getAllDuties (*gin.Context)
- addNewGrade (*gin.Context)
- saveNewDuty (*gin.Context)
- getGradingByDutyId (*gin.Context)
- getDutyById (*gin.Context)
authControllerImpl
- ms service.MemberService
- fs service.FamilyService
dutyControllerImpl
- ds service.DutyService

Figure A.3: Controller Layer Design of SEAP Rest API

Bibliography

[1] *TMS ELTE*. URL: `https://tms.inf.elte.hu/` (visited on 05/14/2024).

[2] *Clean*. URL: `https://wiki.clean.cs.ru.nl/Clean` (visited on 05/14/2024).

[3] *Google Chrome - The Fast & Secure Web Browser Built to be Yours*. URL: `https://www.google.com/chrome/` (visited on 05/14/2024).

[4] *Get Firefox browser — Mozilla (US)*. Mozilla. URL: `https://www.mozilla.org/en-US/firefox/` (visited on 05/14/2024).

[5] *Safari*. Apple. URL: `https://www.apple.com/safari/` (visited on 05/14/2024).

[6] *Download Microsoft Edge*. URL: `https://www.microsoft.com/en-gb/edge/download` (visited on 05/14/2024).

[7] *Opera Web Browser | Faster, Safer, Smarter | Opera*. URL: `https://www.opera.com/` (visited on 05/14/2024).

[8] *Postman Collections: Organize API Development and Testing*. Postman API Platform. URL: `https://www.postman.com/collection/` (visited on 05/14/2024).

[9] *MySQL :: Download MySQL Installer*. URL: `https://dev.mysql.com/downloads/installer/` (visited on 05/14/2024).

[10] *MySQL :: Download MySQL Workbench*. URL: `https://dev.mysql.com/downloads/workbench/` (visited on 05/14/2024).

[11] *The Go Programming Language*. URL: `https://go.dev/` (visited on 05/14/2024).

[12] *Node.js — Download Node.js®*. URL: `https://nodejs.org/en/download` (visited on 05/14/2024).

[13] *Postman API Platform | Sign Up for Free*. Postman. URL: `https://www.postman.com` (visited on 05/14/2024).

[14] *Gin Web Framework*. Gin Web Framework. URL: `https://gin-gonic.com/` (visited on 05/14/2024).

[15] Jinzhu. *GORM*. GORM. Apr. 23, 2024. URL: `https://gorm.io/index.html` (visited on 05/14/2024).

[16] *What are Refresh Tokens and How They Interact with JWTs?* URL: `https://www.loginradius.com/blog/identity/refresh-tokens-jwt-interaction/` (visited on 05/14/2024).

[17] *Using middleware*. Gin Web Framework. Section: docs. URL: `https://gin-gonic.com/zh-tw/docs/examples/using-middleware/` (visited on 05/14/2024).

[18] *Cross-Origin Resource Sharing (CORS) - HTTP | MDN*. May 8, 2024. URL: `https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS` (visited on 05/14/2024).

[19] *Angular - CanActivate*. URL: `https://angular.io/api/router/CanActivate` (visited on 05/14/2024).

[20] *Angular - Using Angular routes in a single-page application*. URL: `https://angular.io/guide/router-tutorial` (visited on 05/14/2024).

[21] *Angular - HttpInterceptor*. URL: `https://angular.io/api/common/http/HttpInterceptor` (visited on 05/14/2024).

[22] *alert('OAuth 2.0'); // The impact of XSS on OAuth 2.0 in SPAs*. URL: `https://pragmaticwebsecurity.com/talks/xssoauth.html` (visited on 05/14/2024).

[23] *google/gvisor*. original-date: 2018-04-26T21:28:49Z. May 14, 2024. URL: `https://github.com/google/gvisor` (visited on 05/14/2024).

[24] *Amazon S3*. Amazon Web Services, Inc. URL: `https://aws.amazon.com/pm/serv-s3/` (visited on 05/14/2024).

[25] *Managed SQL Database - Amazon Relational Database Service (RDS) - AWS*. Amazon Web Services, Inc. URL: `https://aws.amazon.com/rds/` (visited on 05/14/2024).

List of Figures

List of Tables

List of Codes