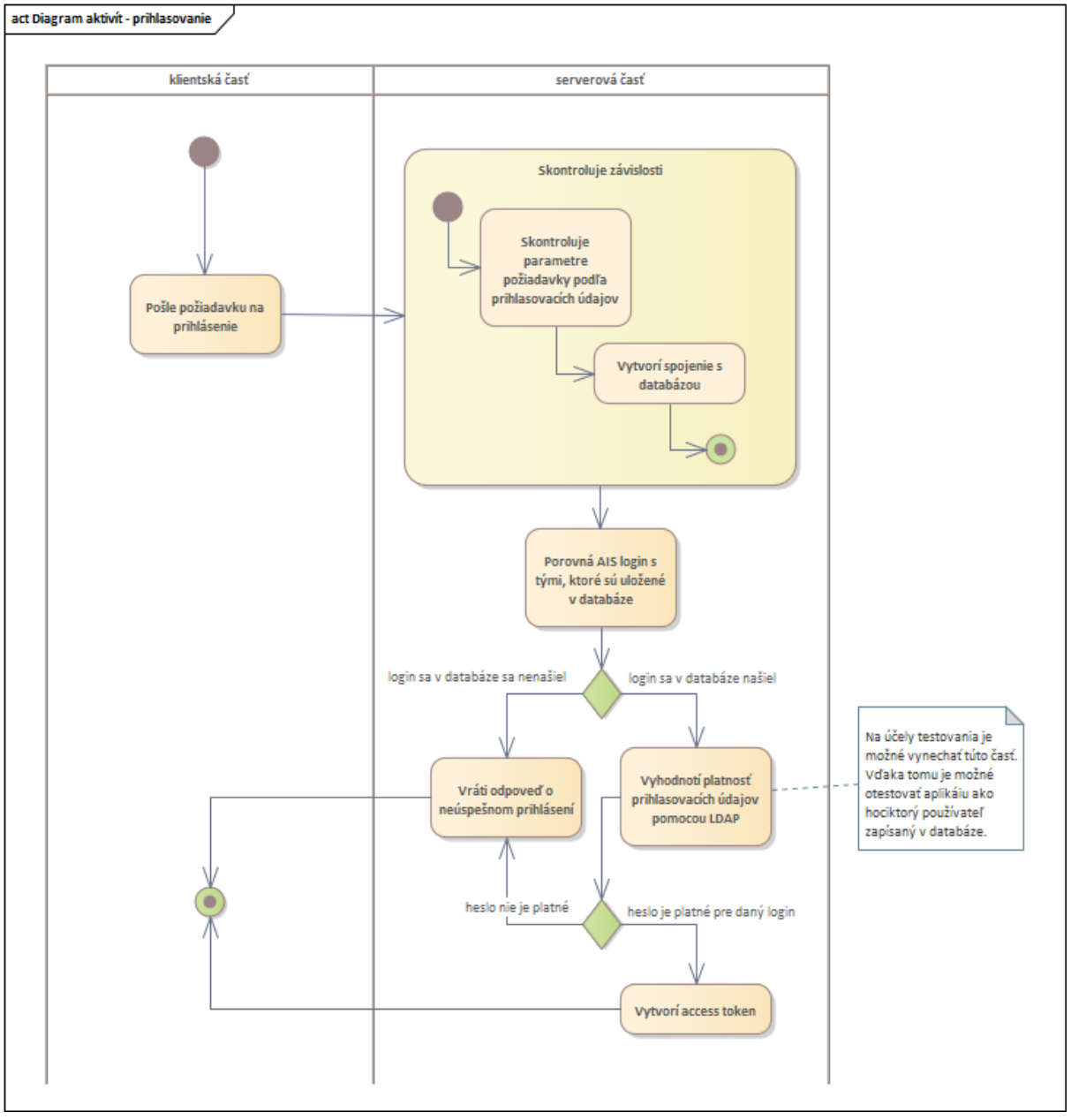
DEIMOS: Documentation of the server part (backend)

General functionalities

Login

The activity diagram for logging in is shown in the image below.



JWT web tokens are used to secure login. Tokens are valid for 1 day. After expiration, the token is not renewed. Tokens are encrypted with an algorithm *HS256*. These values are in the *.env* file.

Login is possible according to the AIS login and only for users who are registered in the database in the table *users*. (Administrators are the users Lukáš Kohútka and Admin 1, who are added to migrations when creating tables. More in the Migrations section.)

The correctness of the password is checked according to AIS using LDAP. In addition to the correctness evaluation, other data are returned such as e.g. role.

The application distinguishes three roles: *admin*, *teacher*, *student*.

The evaluation of the role is divided into two parts. Firstly, it checks what is written in the database for a particular user. Then, there is a compare check on whether it fitts roles returned from AIS. (Since the student himself can also be a teacher - e.g. OPP subject.)

Access to the application is also allowed for sample users according to the table.

AIS id	**less**	**login**	**password (for day January 1st)**
1	Admin 1	a1	a_deimos_1.1
2	Teacher 1	t1	t_deimos_1.1
3	Teacher 2	t2	t_deimos_1.1
4	Student 1	s1	s_deimos_1.1
5	Student 2	s2	s_deimos_1.1
6	Student 3	s3	s_deimos_1.1

For sample users, the password is not checked against AIS, but directly in the application. They have a password according to this template:

```
[a|t|s - by user role]_deimos_[current day].[current month]
```

Passwords for sample users were created in such a way that they can also be used in production.

Feedback

Feedback can be created by users of all roles. Besides storing feedback it also stores the person who created it (according to his *id*).

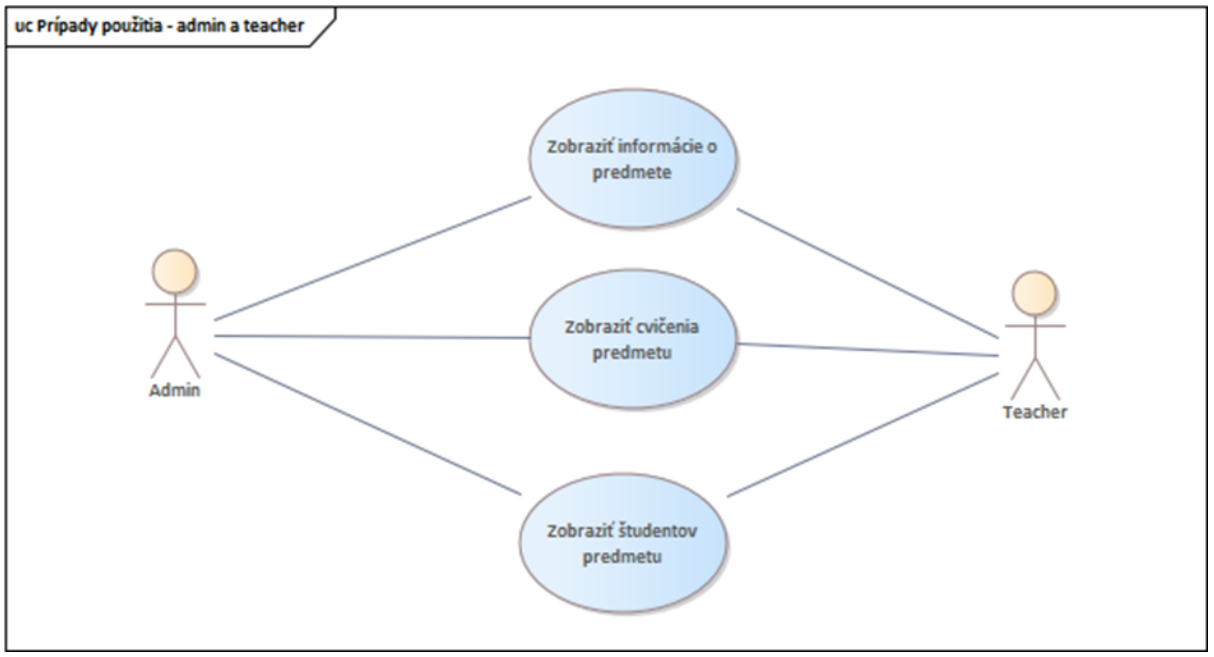
Only the role can watch it with their authors and evaluate it *admin*.

The data is stored in the database in the table *feedback*.

Functionality for the student role

Functionality for the role of admin and teacher

The use case diagram in the figure below shows all the implemented functionalities common to the role *admin* a *teacher*, which are described in more detail below.



Although the created APIs are different in prefix */admin* a */teacher*, but perform the same functionality common to both roles. The only difference is that for the role *teacher* only information about his seminars and students will be filtered out.

View information for admin and teacher roles

According to the selected subject, information about the selected subject, seminars, students, created tasks and solved tasks of a specific student is displayed.

View information about the subject

The information that is returned for display about the given subject is the name and abbreviation of the subject, semester, year and all teachers.

Information on whether students have already been imported for the given subject is also returned (more about importing students is in the section Managing seminars and students). If they were already imported, it is not possible to re-import students for the given subject.

The API route is divided according to roles (*admin*, *teacher*), but the same function is called to process the request.

The content of the function *get_subject_header()* in *subjects_controller.py* is checking the role and correctness of the subject *id* and there is a query to retrieve data from the database. There are two cases to be considered. When there are and are not teachers imported to the database. The reason is that the query concatenates teachers with the subject using *JOIN* to get everything needed. When there are no teachers yet, it returns only basic information about the subject. The output is serialized to the desired structure before the response is returned.

View subject seminars

The information that will be returned is the day and time of the seminar, the number of students participating in the seminars, and their teachers.

The API route is divided according to roles (*admin*, *teacher*), but the only difference in processing the request is that for the role *teacher* are filtered out only the seminars that are teached by the given teacher.

Functions *list_all_seminars_admin()* and *list_all_seminars_teacher()* perform a role check and a query that returns all seminars of the subject, with the number of students in each seminar counting from the table *users_seminars*. The response is returned serialized.

View subject assignments

View students of subject

The information that is returned is the maximum number of tasks and points that the students could get and basic information about the student (name, login) and for each student how many tasks he solved and how many points he got as well as information about his seminar (day and time)

The API route is divided according to roles (*admin*, *teacher*), but the only difference in processing the request is that for the role *teacher* are filtered out only the students teached by the given teacher.

Display all students of the subject perform functions *list_subject_students_admin()* a *list_subject_students_teacher()*.

At the beginning, there is a query that counts all task assignments and their points for a given subject. Next, a second query is executed based on what the first query returned.

- If there **are no assignments** yet for the subject**,** the result of the second query is only a list of students with their seminars. To facilitate the serialization of the result in this case, the query contains 2 columns, which will be created by using *literal(0)*. These represent the number of solved tasks and the number of points the student received, which will be filled with zeros because there is no assignment yet.
- If there **are any assignments** for the subject, the result of the query will return, in addition to the students, the number of points and assignments that the students have solved. To interconnect projects (solved only) of the student and assignments (of solved projects) of the given subject properly, it was necessary to connect the tables *users* and *assignments* with a table *projects* based on two parameters:
    - *users.id = projects.created_by*
    - *assignments.id = projects.assignment_id*

The reason for the complexity of the query in the case of the existence of assignments is that, on the one hand, it is necessary to find all the projects of the given student, but at the same time only those that are the solution for the assignments of the given subject.

View students of seminar

The information that will be returned is the time and day of the seminar, the maximum number of tasks and points that the students could get and basic information about the student with the number of tasks that he solved and how many points he got for them.

The API route is divided according to roles (*admin*, *teacher*), but the only difference in processing the request is that for the role *teacher* is returned a list of students, when the seminar belongs to the given teacher.

To retrieve students from the seminar perform functions *list_students_of_seminar_admin()* and *list_students_of_seminar_teacher()*.

At the beginning, there is a query that returns basic information about the seminar and counts all assignments and their points for the given subject. Next, a second query is executed based on what the first query returned.

- If there **are no assignments** yet for the subject, the result of the second query is just a list of students. To facilitate the serialization of the result in this case, the first 2 columns are created by using *literal(0)*. These represent the number of solved tasks and the number of points the student received, which will be filled with zeros because there is no assignment yet.

- If there **are any assignments** for the subject, the result of the query returns, in addition to the students, the number of points and assignments that the students have solved. To interconnect projects (solved only) of the student and assignments (of solved projects) of the given seminar, the seminar is filtered out according to its *id* and at the same time the tables are joined *users* and *assignments* with a table *projects* based on two parameters:
    - *users.id = projects.created_by*
    - *assignments.id = projects.assignment_id*

The reason for the complexity of the query in the case of the existence of assignments is that, on the one hand, it is necessary to find all the projects of the given student, but at the same time only those that are the solution for the assignments of the given subject.

Running code and evaluating it

Both Admin and Teacher can compile and run student's code, run tests for this code and update points achieved by a student for this particular code (assignment).

For details about code compilation and running refer to *Code Compilation and Running* inside *Technical and Other Functionalities* chapter.

Code responsible for updating assignments points functionality can be found **routes.admin_router.admin_projects.update_project**, **routes.teacher_router.teacher_projects.update_project**. This is a standard UPDATE operation on Project entity but for now only points_achieved field update is available.

Changing application settings

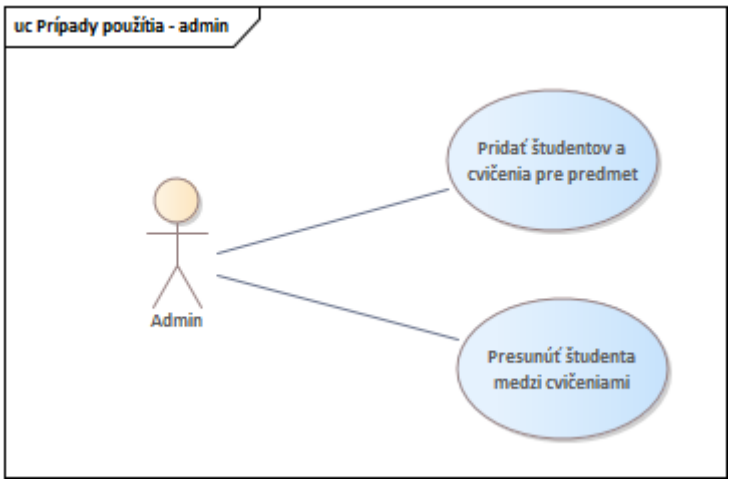
Admin is capable of listing and changing application settings. Both functionalities are implemented inside **routes/admin_router/admin_settings.py.** This is the list of application's settings:

global_run_timeout - maximum time that student's code can run in seconds
max_feedback_count_per_user - maximum amount of feedback that every user can send (to prevent spam attacks)
container_memory_limit - maximum amount of memory that Docker engine can allocate to a single run-compile container. Units are defined by documentation od Docker Python SDK - Docker SDK for Python
global_memory_limit_heap - maximum amount of memory that can be allocated for heap for compilation and run in bytes
global_memory_limit_stack - maximum amount of memory that can be allocated for stack for compilation and run in bytes

Names of the settings cannot be changed as they are not dynamic and defined by behavior of the frontend.

Management functionalities

The management functionalities listed below are only possible for the admin role, which are also shown diagram use cases.



Subject management

Adding new subject

Task management

Adding an assignment

Administration of seminar and students

Adding students and seminars for a subject

This functionality is implemented by importing a type file.*csv* and not by manually adding seminars and students.

The mentioned file is exported from data from AIS. To successfully import data into the Deimos application, the file must contain information about the student, their teacher and the seminar.

After consultation with the AIS integrator of FIIT (Mrs Gnipova) it is possible to request a given export from AIS at the beginning of each semester for the selected subject.

The header.*csv* file must contain exactly the following fields: *STUDENT_ID, STUDENT, STUDENT_LOGIN, TEACHER_LOGIN, TEACHER_ID, TEACHER, DAY, TIME.* Field *STUDENT* or *TEACHER* represents the student's or teacher's full name. The required format of field *DAY* is a number (1 - Monday, 7 - Sunday). Format of field *CAS* is *hh.mm*. An example is shown below.

```
STUDENT_ID;STUDENT;STUDENT_LOGIN;TEACHER_LOGIN;TEACHER_ID;TEACHER;DAY;TIME
4;Student 1;s1;t1;2;Teacher 2;1;08.00
```

It is only possible to import students and seminars from a file once for a subject.

Information about students and teachers is stored in the table *users*. Information about the seminar is stored in the table *seminars*. The link between student and seminar is stored in the table *users_seminars*. If a record about the user already exists in the database (e.g. for another subject), then only the mentioned link will be created.

There is also an API for removing imported students and seminars, but it was mainly used for testing the import. If the students have already opened or completed a task, it will not work. It therefore represents an experimental functionality that is not even provided on the frontend (FE).

Transferring students between seminars

For the admin role, it is possible to move the selected student to another seminar. The only condition is that there has to always be at least one student per seminar. Otherwise, the change is not possible.

Technical and other functionalities

Migrations

We used the tool called *Alembic* to gradually track changes in the database. This tool was created for the Python tool *SQLAlchemy* to communicate with the database. During development, we created several migrations to create the tables in the final form, which are connected to each other.

In addition, migrations are also used to write data to the database. The migration creates records in the table *users* for Lukáš Kohútka with a role *admin* and all sample users.

Run this command `alembic upgrade head` to create all the tables in the database. At the same time, the table *alembic_version* is created to track the current version that was lastly added.

This command `alembic revision --autogenerate -m "<name>"` is used to create a new migration that detects all changes in the models and creates commands to modify the

database. It may happen that when creating the mentioned migration (for *Alembic 1.8.1*) all constraints are created anew. The cause of the error is in *Alembic*, which cannot correctly recognize the database schema. However, it does not return an error. In addition, since those restrictions are already present in the database, nothing will really change. But we have removed all these redundant commands in existing migrations.

It is possible to use and edit tables even without using the *Alembic*, but then it would be necessary to manually modify the changes in the models in all databases as well.

Code uploading

Before a student's code can be compiled and run it needs to be uploaded to the server. This is done via FastAPI Request Files (https://fastapi.tiangolo.com/tutorial/request-files/). To prevent large file uploading **controllers.import_controller.valid_content_length** dependency is included at the endpoint definition. Optimally maximum file size should be limited by webserver configuration.

Code files are saved under a path that is defined in **docker-compose** files as a mount path for students' projects. External (from a backend container perspective) path for saving code is needed due to the fact that we want to keep those files even when the backend container dies. Single code file is saved under

${path_to_projects}/${student_name}/${assignment_id}/code.c.

The name of the code file should remain the same (code.c) as the possibility of multiple files compilation + run is not implemented yet.

Contain of project file can be retrieved either by

controllers.projects_controller.get_project_files_content or

controllers.projects_controller.return_project_files_binary.

First method returns the file's content in a text format while the second returns the code as UploadFile code that is used further in the code compile+run method. Second method is used when we don’t want to upload and save new code but just run existing one.

Code compilation and running

Depending on the called endpoint and the condition of the project (submitted / not submitted) project code file can be updated/uploaded again by the principles described above.

Code compilation and run is done with the help of Docker SDK for Python and done in a separate container due to security reasons. There are several application settings that are retrieved from database before code compilation:
global_run_timeout - maximum time that student’s code can run in seconds

container_memory_limit - maximum amount of memory that Docker engine can allocate to a single run-compile container. Units are defined by documentation od Docker Python SDK - Docker SDK for Python
global_memory_limit_heap - maximum amount of memory that can be allocated for heap for compilation and run in bytes
global_memory_limit_stack - maximum amount of memory that can be allocated for stack for compilation and run in bytes

Before compilation starts we are writing user input from a frontend to the file so we can use it later.

For code compilation and run we are using Docker **gcc** image and we are mounting volume where students projects are located. Compilation command is long, complicated and contains multiple parts that are connected by && operator. Let's break it down:

gcc -o compiled {file.filename} - standart simple gcc file compilation
ulimit -SH -d {memory_limit_heap} -s {memory_limit_stack} - we are using ulimit to set limits for a heap and stack memory
cat user_input | timeout {time_limit} ./compiled - we are parsing user input to the compiled program and using timeout to set the maximum amount of time that program can run.
echo - extra print line due to the unpredictability of Docker SDK. Without it run result is cut off and not complete.

If the run was successful we return the run result back to the client side.

There can be several errors during code compilation+run. Known return statuses are:
124 - timeout during run caused by timeout program
137 - memory limit for a single container was exceeded caused by Docker daemon.
139 - segmentation fault caused by ulimit

After compilation+run are completed we are removing user_input and compiled files in case there was an error.

Running tests

Test runs are basically compile+run operations done multiple times for each defined input. It can take a while and to speed up this process asynchronous processing with ***asyncio*** package was used. Depending on the run result for the single test evaluation is made.

Unit tests

There are several unit tests written under the test directory. To run tests you need to set variable **testing** to **True** and run command **pytest.**

At the time of writing this document running unit test functionality is broken.

Logging

All APIs also include commands to log activities.

Information that is written in the logs:

- date and time
- log level (*debug* for regular statements, *error* for errors and exceptions)
- AIS login of the user who created the API request
- information about the request processing stage:
    - start: API route name and parameters (if needed)
    - during processing: function name and parameters, queries to the database (before running them)
    - conclusion: the answer to the given request

The predefined logging level is *debug*.

Possible future improvements

Comparing assignments against a character string is insufficient. When creating a new assignment, there should be an option to add source code to test students' projects more properly (mainly to test bigger ones).

According to the GCC version, there are libraries, functions, etc. which could or could not be used. Therefore set the correct GCC version to run the C code (also according to the agreement with the supervisor) to forbid some of libraries, functions, etc in students' projects. For example *sort()*.

Fixing pytest and including it in CI/CD pipeline.

Docker build optimization for both Frontend and Backend images (refer to https://docs.docker.com/build/cache/)

Creating mechanism for storing backend container logs externally.

Forbid system calls and C build-in functions (sort for example) inside student's code.

Database credentials

PROD

HOST: deimos.fiit.stuba.sk

PORT: 5437

DB_NAME: deimos

DB_USER: deimos

DB_PASSWORD: uJrhjb4bYdWxmF36

DEV

HOST: deimos.fiit.stuba.sk

PORT: 5437

DB_NAME: deimos-dev

DB_USER: deimos

DB_PASSWORD: uJrhjb4bYdWxmF36