



POLITECNICO
MILANO 1863

Automation and Control Engineering

Implementation Document

Project of Software Engineering (for Automation)
Professors: Rossi Matteo Giovanni, Lestingi Livia

SchedulEx 

26/07/2023

Authors:

Petulicchio Lorenzo (10639923) – Talacci Mattia (10647582)

Sommario

| | |
|---|----|
| 1 - CODE STRUCTURE..... | 3 |
| FRONT-END..... | 3 |
| main.dart | 3 |
| UserState.dart..... | 3 |
| ProblemSessionState.dart | 4 |
| UnavailState.dart | 6 |
| BackEndMethods.dart | 7 |
| utils.dart..... | 10 |
| LoginPage.dart | 11 |
| SelectPage..... | 12 |
| SessionPage.dart..... | 13 |
| UnavailPage.dart..... | 13 |
| SettingsPage.dart..... | 14 |
| CalendarPage.dart | 14 |
| BACK-END | 15 |
| backend.py..... | 15 |
| Optimization_Manager.py..... | 16 |
| Optimizer.py | 18 |
| Utils.py | 19 |
| DATABASES | 20 |
| Database Problem | 20 |
| Database Exam | 20 |
| 2 - REQUIREMENTS IMPLEMENTED | 21 |
| 3 - USER MANUAL..... | 22 |
| User Log in | 22 |
| Session of work management | 23 |
| School and exam session dates and selection | 25 |
| Problem Session Savings..... | 26 |
| Selecting start and end date..... | 26 |
| Comment adding | 27 |
| Unavailability management..... | 27 |
| Settings management..... | 31 |
| Calendar scheduling starting | 33 |
| Optimization status checking | 35 |
| Calendar Download | 35 |

Notes.....36

4 - INSTALLATION GUIDE37

1 - CODE STRUCTURE

FRONT-END

For the front-end, Flutter was chosen as the framework for several reasons. Firstly, Flutter is an open-source framework developed by Google that enables the creation of high-quality cross-platform applications with a single code base. This allows the application to be used on other devices in the future.

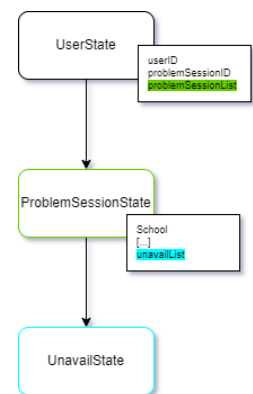
Secondly, Flutter offers a wide range of customisable and predefined widgets that facilitate the creation of an attractive and responsive user interface. Flutter's widgets are fully customisable, making it possible to create a valuable user experience.

In addition, Flutter uses the Dart programming language, which offers a simple and intuitive syntax. This made it possible to develop the application efficiently and to keep the code clean and readable.

Finally, another advantage of Flutter is its excellent performance. Thanks to its responsive architecture based on widget rendering, Flutter offers a smooth and lag-free user experience, ensuring a fast and responsive application.

Considering all these advantages, it was felt that Flutter was the ideal choice for developing the front-end application.

For the FrontEnd each script corresponds to the respective page visualized by the user. To keep track of the user selection through the webapp it is used an app state management logic, implemented in three different states which correspond to deeper hierarchical level of interaction between each other. These scripts are grouped in the '/model' folder. Their main objective is to manage a local copy of what is stored in database to visualize and modify it, as the backend do but without directly fetch the database.



main.dart

In this script, the app instance is specified and the **main()** function is used to make it run.

1. Within the **MyApp** widget we initialize the cascade of instances of object to have each state to be notified every change of the superior level one. These are called **Provider** and **ChangeNotifierProvider**. As we described before and can be seen in the figure above, each of these statements are contained within each other.
2. **ChangeNotifierProxyProvider<UserState, ProblemSessionState>** provides the connection between the **ProblemSessionState** widget and **UserState**, since for **UserState** we defined a List of **ProblemSession**, this is needed to be updated.
3. **ChangeNotifierProxyProvider<ProblemSessionState, UnavailState>** provides the connection between the **UnavailState** widget and **ProblemSessionState**, since for each **ProblemSessionState** we defined a List of **Unavail**, this is needed to be updated.
4. Then it's defined the **GoRouter** widget which manages all the paths to the corresponding pages and the initial location in which the user is set, once **SchedulEx** is opened.

UserState.dart

The **UserState** class is responsible for managing the state of the user and their associated problem sessions in the **SchedulEx** software. It extends the **ChangeNotifier** class, allowing it to notify listeners when the state changes and ensuring that the UI reflects the most up-to-date information.

► State Variables

It has two string variables, **userID** and **problemSessionID**, which store the current user's ID that logged in and the ID of the active problem session in which the user is making modification.

The class also contains a list, **problemSessionList**, which holds instances of the **ProblemSession** class. This list represents all the problem sessions that can be associated with the current user. At this level is the container of all the sessions stored in the database.

▶ *setUserID*

This function takes a user ID string as a parameter and updates the **userID** variable accordingly.

1. It then calls the **getSessionList** function, which belongs to **BackendMethods.dart** and retrieves from the backend a list of all session data stored in the database. (More about it in the chapters to follow)
2. Once the list of problem sessions is retrieved, it is assigned to the **problemSessionList** variable.

▶ *createSession*

The **createSession** function is responsible for creating a new problem session for the current user.

1. It calls the **saveSession** function, which belongs to **BackendMethods.dart**, passing the user ID and the initial status of "NOT STARTED" as parameters. This function returns an id of the new Session created id a sessionID is empty. More about it in the chapters to follow.
2. Upon successful creation of the session, the function updates the **problemSessionID** variable with the newly generated session ID and adds a new **ProblemSession** instance to the **problemSessionList**.

▶ *update*

The **update** function is used to modify the properties of an existing problem session.

1. It takes a **Map<String, dynamic>** containing the session ID as 'id' and the updated properties as a string parameter.
2. The function searches for the session with the given ID in the **problemSessionList**.
3. **If found**, it updates the corresponding **ProblemSession** instance with the provided properties (e.g., **school** and **status**), since these are the only attributes to be visualized at this level.
4. **If the session does not exist** in the list, it means it is a new session, and a new **ProblemSession** instance is added to the list with the provided session ID.

▶ *delete*

The **delete** function is used to delete a problem session associated with the user.

1. It takes the session ID as a parameter and calls the **deleteSession** function to remove the session from the database.
2. After successful deletion, the corresponding **ProblemSession** instance is removed from the **problemSessionList**.

ProblemSessionState.dart

The **ProblemSessionState** class is a fundamental component of the SchedulEx software, responsible for managing the state of the selected problem session. Similarly, to **UserState**, it extends the **ChangeNotifier** class, enabling it to notify listeners about any changes and ensuring that the user interface always displays the latest information.

▶ *State Variables*

1. One of the major advantages of use **ChangeNotifierProvider** is that you can define as a updated state variable, the **UserState** of the upper level, and listen to all his changes.
2. **selectedSessionID**, which stores the ID of the currently chosen problem session. Additionally, there's a nullable **DateTimeRange sessionDates**, representing the start and end dates of the selected problem session. Using this type we can have other attributes, like the *duration* in terms of number of day.
3. **school** String variable holds the name of the school associated with the chosen problem session.
4. **unavailList**, a list containing instances of the *Unavail* class. This list is used to visualize the total unavailability associated to the specific problem session.
5. **ProblemSessionState** also handles various settings and configurations specific to the chosen session, even if they are visualized in another page (**SettingsPage.dart**). These include variables like

- a. **minDistanceExams**: stores the value in days of distance between exam of the same semester, that the user decides.
- b. **minDistanceCallsDefault**: stores the default value in days of distance between calls of the same exam.
- c. **numCalls**: stores the number of calls chosen by the user for the selected session.
- d. **currSemester**, and exceptions, each serving a unique role in shaping the behavior of the application for the selected session.
- e. **Exception** is a list of instances of object to associate ad each exam the custom distance between calls for that specific exam.

▶ *resetProblemSessionID*

This function resets all session-related state variables to their initial values. This function ensures a clean slate when switching between problem sessions in the selectPage.

▶ *updateSettings*

This function implements the modification of session-specific settings based on the provided **payload**.

1. The payload is an instance of *map<String,dynamic>* For which parameters it is necessary to verify their existence.
2. The function updates relevant class variables and saves the changes to the database using the **setSettings** function from *BackendMethods.dart* with the **modifiedpayload** as input parameter.

▶ *insertException and deleteException*

With these functions, users can manage the exceptions for specific exams, relaxing some of the constraints introduced. Is important to notice the use of the same function **setSettings** that is able to behaves differently if a spefication instance is provided in input. (See BackendMethods.dart)

▶ *setProblemSessionID*

This function plays an important role for the GUI, since it retrieves session data of the chosen one data from the database and updates class variables accordingly.

1. Firstly, the input **id** value is assigned to **selectedSessionID**.
2. Then the getSessionData is called and async response is a *dynamic* that can have or can not have all of the state variables specified in this widget. So it go through a check for all of them.
3. For the settings, since it has a more complicated structure the verifying is done from the **numCalls** value to each **exception's** instances.
4. At the end, the method **updates** of the upper level state is called

▶ *setStatus, setSchool and setStartEndDate*

The **setStatus** function sets the status of the problem session, receiving a *string* in input, while the **setSchool** function assigns a school to the session. For each of them is called the method **update** of the upper-level state but only for the setSchool function is called the **SaveSession** BackendMethod, since the status can be modified on the database only by the backend logic.

The **setStartEndDate** function sets the start and end dates for the selected problem session. If the dates are not null, it saves them to the database using the **saveStartEndDate** function, converting the datetime object in string.

▶ *UpdateUnavail*

The update function modifies the properties of an existing *Unavail* instance or adds a new instance to the unavailList based on the provided update. The user can thus manage unavailability information for the selected session effectively.

▶ *deleteUnavail*

The `deleteUnavail` function removes an `Unavail` instance from the `unavailList` based on the provided ID. It also deletes the corresponding unavailability information from the database using the `deleteUnavailability` function from `BackendMethods.dart`.

`showToast`

The `showToast` function assists with user notification, displaying a brief toast message with the **text** specified in input.

UnavailState.dart

This is the component of the lowest level, as more unavailability is defined for each problem session and is in fact responsible for managing the state of unavailability for problem sessions. As has been said previously for the other components, it extends the `ChangeNotifier` class, enabling it to notify listeners when any changes occur, ensuring that the user interface stays up to date.

▶ *State variables*

1. The `UnavailState` class has a significant association with the `ProblemSessionState` class, allowing it to access data and methods related to problem sessions. This association is made through the variable **sessionState**
2. The class contains several variables that define the unavailability: **id**, representing the unique ID of the unavailability. The integer variable **type** represents the **type** of unavailability, and the string variable **name** holds the name of the professor involved or the name of the classroom or the description of the polytechnic event to which the unavailability refers.
3. **Dates** is the list of date in which the unavailability occurs for the entity specified in **name**.

▶ *reset*

The `reset` function resets all unavailability-related state variables to their initial values. This function ensures a clean slate when managing different unavailability instances.

▶ *setCurrID*

The `setCurrID` function takes a **newID** as a parameter and sets it as the current unavailability ID.

1. If the provided ID is **not empty**, the function retrieves the unavailability data associated with that ID from the database using the `getUnavailData` async function from `BackendMethods.dart`.
2. After obtaining the data, the function updates the class variables **id**, **type**, **name**, and **dates** accordingly.
3. If the provided ID is **empty**, the function creates a new unavailability instance using the `createUnavail` function and sets the newly generated ID as the current ID.

▶ *createUnavail*

The `createUnavail` function is responsible for creating a new unavailability instance when an id isn't specified.

1. It calls the `saveUnavailability` function from `BackendMethods.dart`, passing the session ID and an empty payload, cause the user hasn't set anything yet.
2. Upon successful creation, the function updates the class variable **id** with the generated ID.
5. Then it's called the method **update** of the upper-level state

▶ *setType, setName*

The `setType` function sets the type of unavailability, updating the class variable **type** and saving the change to the database using the `saveUnavailability` function with the specified payload. Similarly, the `setName` function updates the name/description of the unavailability, modifies the class variable **name**, saving all the changes to the database.

▶ *addDates, deleteDates*

These functions allow managing dates to the current unavailability instance. They simply call the respective `BackendMethods.dart` function.

1. The first one calls the *addDatesToUnavail* function from *BackendMethods.dart*, passing the session ID, current unavailability ID, and a list of **newdates**, since from the GUI is possible to enter multiple types of unavailability all assignable to a list of dates. Upon receiving the updated list of dates from the backend, the function updates the class variable *dates*.
2. The second one calls the *deleteDateToUnavail* function from *BackendMethods.dart*, passing the session ID, current unavailability ID, and the date to delete. Upon successful deletion, the function updates the class variable *dates* and notifies listeners of the changes.

In summary, the *UnavailState* class plays a crucial role in managing the unavailability of problem sessions in the *ScheduleX* software. By allowing users to set, modify, and delete unavailability instances, it enhances the application's flexibility and adaptability to scheduling constraints. The class maintains close coordination with the *ProblemSessionState* class and communicates changes to its state to ensure a seamless user experience.

BackEndMethods.dart

This script contains various utility functions used for managing data and making HTTP requests with the 'package:http/http.dart' used for the *ScheduleX* front-end to instaurate successful communication with the backend. The **SERVER_URL** string specifies the ip, where our backend server is hosted. It's important to notice the for educational purposes some functions that handle similar operations are implemented differently and it was decided to keep these functions, which are within the operational scope of the backend, separate from the code that handles the GUI, to maintain it lighter and less confusing.

▶ *saveUserID*

Makes a **POST** request to the server with the **sessionID** and **userID** as parameters to save the user ID associated with a specific session. Handles the response from the server and prints appropriate success or error messages based on the response status code.

▶ *saveSchoolID*

Makes a **POST** request to the server with the **sessionID** and **schoolID** as parameters in the request body to save the school ID for a given session. Handles the response from the server and prints appropriate success or error messages based on the response status code.

▶ *saveStartEndDate*

Makes a **POST** request to the server with the **sessionID**, **startDate**, and **endDate** as parameters to save the start and end dates for a session. Handles the response from the server and prints appropriate success or error messages based on the response status code.

▶ *saveSettings*

Makes a **POST** request to the server with the **sessionID**, **distCalls**, and **distExams** as parameters to save scheduling settings for a session. Handles the response from the server and prints appropriate success or error messages based on the response status code.

▶ *saveUnavailability*

This is one of the crucial operations in the session scheduling. Is designed to be as versatile as possible to future developments, in fact it saves all properties specified in **payload** (for now only type and name)

1. Sends a **POST** request to the server with the **sessionID**, **unavailID**, and **payload** (a map of unavailability data) as parameters to save unavailability data related to a session.
2. Converts the **payload** into a request body, including the **sessionID** and **unavailID** (if not empty) along with other unavailability data. If it empty is not specified in the payload. In this way the backend must create a new one
3. Handles the response from the server and prints appropriate success or error messages based on the response status code.
4. Returns the decoded **JSON** response from the server.

▶ *addDatesToUnavail*

Since managing the actual dates of unavailability is more complicated the update of this value is done through a different function, that make the request to add a specific set of dates. The backend responds with the final list of dates after processing it.

1. Sends a **POST** request to the server with the **sessionID**, **unavailID**, and **newDates** (a list of `DateTime` objects) as parameters to add dates to an existing unavailability instance associated with a session.
2. Converts the **newDates** list to a string format suitable for the request payload, divided by a `'/'`.
3. Handles the response from the server and prints appropriate success or error messages based on the response status code.
4. Returns a `List<DateTime>` after decoding the response and check the actual existence of a valid string to parse into a `DateTime` containing the updated list of dates received from the server.

▶ *deleteDateToUnavail*

Since the button that can activate this function is defined for each date of a **Unavail**, the input is for sure only one `DateTime`, and doesn't need to be processed further.

1. Sends a **POST** request to the server with the **sessionID**, **unavailID**, and **dateToDelete** (`DateTime` object) as parameters to delete a specific date from an existing unavailability instance.
2. Handles the response from the server and prints appropriate success or error messages based on the response status code.

▶ *deleteUnavailability*

In this case we have as input an entire `Unavail` type variable, even if only the `id` property is used. This choice was made to be able to make possible reimplementations of the `Unavail` Class easier.

1. Sends a **GET** request to the server with the **sessionID** and **unavail** (`Unavail` object) as parameters to delete an unavailability instance associated with a session.
2. Handles the response from the server and prints appropriate success or error messages based on the response status code.

▶ *deleteSession*

Similarly for the previous function, delete an entire `ProblemSession` instance stored in the database, but receiving as input only the **sessionID**. In fact, it's used in the *SelectPage* where not all information of a `ProblemSession` instance is requested.

1. Sends a **GET** request to the server with the **sessionID** as a parameter to delete a session.
2. Handles the response from the server and prints appropriate success or error messages based on the response status code.

▶ *startOptimization*

Sends a **GET** request to the server with the **sessionID** as a parameter to initiate the optimization process for a session. Although seemingly very simple, it actually activates the optimizing core of `Schedulex`.

▶ *getStatus*

Once the scheduling process is started, it's needed to retrieve the status of the optimizer.

1. Sends a **GET** request to the server with the **sessionID** as a parameter to retrieve the status of a session.
2. Handles the response from the server and prints appropriate success or error messages based on the response status code.
3. Returns a `Map<String, dynamic>` containing the status data received from the server, that include the status of the `ProblemSession` ('STARTED', 'NOT STARTED', 'SOLVED', 'NOT SOLVED') and a short description of what the optimizer has been processed.

► *saveSession*

Similarly, to the *SaveUnavail* function, we introduce a versatile way to update each property of the *ProblemSession* instances stored in the database with the exception of the *settings* property.

1. Sends a **POST** request to the server with the **sessionID** and **payload** (a map of session data) as parameters to save session data.
2. **For each property** that is specified in the map, the corresponding value is converted in a string,
3. Handles the response from the server and prints appropriate success or error messages based on the response status code.
4. Returns the decoded **JSON** response from the server.

► *getSessionList*

This function obtains the list of all the problem session that are stored in the database and some of their relative info. Since user management was not the subject of this project, there is no actual validation, but the function is useful for future developments in which to obtain sessions that can only be accessed by certain users.

1. Retrieves a list of all problem sessions by sending a **GET** request to the server.
2. Handles the response from the server and parses every single property of the **JSON** data into a `List<ProblemSession>`, checking if they are null and, if so, assign them a empty string.
3. Returns the `List<ProblemSession>` containing the retrieved sessions.

► *getSessionData*

Once the **selectedSessionID** is set, this function is called to retrieve all the information regarding that *ProblemSession*.

1. Sends a **GET** request to the server and waits for a response.
2. If successful, the function proceeds to decode the JSON response body into the **responseData** variable. Initializes an empty list **result** to store *unavailability data* associated with the session.
3. Checks if the **'unavailList'** key in the **responseData** is **not null**, indicating that there is unavailability data for the session. If unavailability data is present, iterates through each entry in the 'unavailList' and extract the corresponding info.
4. Extracts the unavailability ID (id) and the associated data (data) for each instance. Checks if the data is not null and not empty to ensure there is valid information for the unavailability instance.
5. Parses the **'type'** field from the data, converting it to an integer using `int.TryParse()`. If the 'type' field is not present or is not a valid integer, it **defaults to 0**.
6. Parses the **'name'** field from the data, converting it to a string. If the 'name' field is not present, it defaults to an empty string.
7. Parses the **'dates'** field from the data, converting the list of date strings to a `List<DateTime>` object. If the 'dates' field is not present or is not a valid list of date strings, it defaults to an empty list.
8. Creates an **Unavail** object with the extracted **id**, **type**, **name**, and **dates**, and adds it to the results list. If there is no unavailability data (**unavailList is null**), sets the results list to an empty list.
9. Parses the **'startDate'** and **'endDate'** fields from the **responseData**, converting them to `DateTime` objects if they exist; otherwise, sets them to null.
10. Parses the **'settings'** field from the **responseData**, converting it to a `Map<String, dynamic>` if it exists; otherwise, sets it to an empty map.
11. Returns a Map containing two entries:
 - a. **'problemsession'**: A *ProblemSession* object initialized with the **sessionID**, **school**, **status**, **startDate**, **endDate**, and the unavailability data (results list).
 - b. **'settings'**: A `Map<String, dynamic>` containing scheduling settings data for the session.

► *getUnavailData*

Once the **currID** of the *ProblemSessionState* is set, this function retrieves the corresponding information. Similarly, to the previous function, in order to have the selected **unavail** always updated to the current database state.

1. Retrieves unavailability data for a specific problem session by sending a **GET** request to the server with the **sessionID** and **unavailID** as parameters.
2. Handles the response from the server and parses the JSON data.
3. Then performs a check for each property as it's done in the *getSessionData* function.
4. Returns the **Unavail** object with its data in the suitable format.

▶ *getProfessorList and getExamList*

These two functions interact with the *DatabaseExams* (Through the backend) from which they obtain all exams (necessary for the correct insertion of exceptions in settings, or all the name of professors

1. Retrieves the list by sending a GET request to the server.
2. Handles the response from the server and parses the JSON data into a **List<Exam>** or a **List<String>** respectively.
3. Returns the List containing the names of the professors or the **Exam** instances.

Professor could also be implemented as an object, but in order to make the effective recognition of the related exam more immediate, it was decided to keep a single string identifying name and surname of professors.

▶ *setSettings*

Similarly, to the previous similar function, we introduce a versatile way to update each property of the ProblemSession's settings instances stored. Unlike before, we use a new way of encoding the payload that is processed in one line using a cast to *Map<String, Object>*, which will be better explained in the corresponding method offered by the backend.

▶ *getJsonResults*

After the effective resolution of the scheduling a JSON file is retrieved from the backend, a list of **optExam** that will be processed.

1. Retrieves JSON results for a specific session by sending a **GET** request to the server with the **sessionID** as a parameter.
2. Handles the response from the server and returns the decoded JSON data as a dynamic object.

▶ *downloadExcel*

With this function is possible to obtain a Excel format file of the solution.

1. Downloads an Excel file for a specific session by sending a **GET** request to the server with the **sessionID** as a parameter.
2. Saves the downloaded file to the specified directory.
3. Handles potential errors during the download process and returns appropriate status messages.

utils.dart

The **utils.dart** script provides various utility functions and class definitions for managing problem sessions, unavailabilities and exams, in the SchedulEx front-end . It defines classes for ProblemSession, Unavail, and Exam, representing the respective data structures. Additionally, there are utility functions for date conversion, days range calculation, and hash code generation.

▶ *Unavail Class*

This class represents an unavailability instance. It contains information about the unique ID (**id**), type of unavailability (**type**), list of dates when the unavailability occurs (**dates**), and an optional name/description (**name**) associated with the unavailability.

1. The constructor initializes the **Unavail** object with the provided id, type, dates, and an optional name. The name is set to an empty string if not provided.
2. **setType** function sets the type of unavailability to the **newType** value passed as an argument.
3. **setName** function sets the name/description of the unavailability to the **newName** value passed as an argument.

4. **setDates** function sets the list of dates for the unavailability to the **newDates** value passed as an argument.

► *ProblemSession Class*

This class represents a problem session, containing information about the session's unique ID (**id**), associated school (**school**), status (**status**), description (**description**), user (**user**), start date (**startDate**), end date (**endDate**), and a list of unavailability instances (**unavailList**).

1. The constructor initializes the ProblemSession object with the provided id. Other fields like school, **status**, **description**, **user**, **startDate**, **endDate**, and **unavailList** are set to default values if not provided.
2. **setSchool** function sets the school associated with the problem session to the **newschool** value passed as an argument.
3. **setStatus** function sets the status of the problem session to the **newStatus** value passed as an argument.
4. **setDescription** function sets the description of the problem session to the **newDescription** value passed as an argument.
5. **setStartEndDate** function sets the start date (**startDate**) and end date (**endDate**) of the problem session to the start and end values passed as arguments, respectively.

► *Exam Class:*

This class represents an exam and contains information about its unique ID (**id**), **name**, associated course (**cds**), a list of assigned dates (**assignedDates**), and an optional **professor** (not included in the current implementation).

1. The constructor initializes the Exam object with the provided **id**, **name**, course (**cds**), and a list of assigned dates (**assignedDates**).
2. The **toString** method returns a string representation of the exam, displaying its id and name.
3. The **operator ==** method override the existing method to checks whether two Exam objects are equal based on their id and name.
4. The **hashCode** method calculates a hash code for the Exam object based on its id and name.

► *convertStrToDateList*

This function takes a list of dynamic items (**strList**) as input, where each item can be either a *DateTime* object or a string representing a date. The function converts each item to a *DateTime* object and returns a list of *DateTime* objects.

► *daysInRange*

This function takes two *DateTime* objects (first and last) as input and returns a list of *DateTime* objects from first to last, inclusive. The function creates a list of dates starting from first and incrementing day by day until last (inclusive), and then returns the list of *DateTime* objects representing the days within the specified range.

LoginPage.dart

The **LoginPage** is a stateless widget that represents the login page. It provides a simple user interface where users can enter their username to log in to the app. The widget consists of a `Scaffold` that contains an `AppBar` at the top and a `Padding` widget with the login content in the body.

1. `AppBar`: Displays the title "Login Page" at the top of the screen.
2. `Scaffold`: Provides a basic structure for the page, including the app bar and body content.
3. `Padding`: Adds padding around its child widget to create some spacing between the edges of the screen and the content.
4. `Center`: Centers its child widget within the available space.
5. `SizedBox`: Creates a fixed-size box with a width of 200 pixels. This is used to constrain the width of the login form.
6. `Column`: Arranges its children in a vertical column.
7. `TextFormField`: A text input widget that allows users to enter their username. It has an **onChanged** callback that updates the **inputUsername** variable whenever the user types in the field.

8. `ElevatedButton`: A button widget that triggers the login process when pressed. It has an `onPressed` callback that calls the `userState.setUserID(inputUsername)` function to set the user's ID and then navigates to the next page using `context.pushReplacement('/select')`.
 9. `userState`: It refers to an instance of `UserState` using the `context.watch<UserState>()` method. The `UserState` is likely a state management class that handles user-related state and actions.
 10. `inputUsername`: A local variable that stores the username entered by the user in the text field.
- When the user types in the username field (`TextFormField`), the `onChanged` callback updates the `inputUsername` variable with the entered value.
 - When the user presses the "Login" button (`ElevatedButton`), the `onPressed` callback first calls `userState.setUserID(inputUsername)` to set the user's ID using the value in `inputUsername`. After that, it navigates to the next page using `context.pushReplacement('/select')`.

SelectPage

The `SelectPage` is a `StatefulWidget` that represents the page in ScheduEx where users can select problem sessions. It provides a user interface with a list of problem sessions and options to create new sessions or delete them. The widget utilizes various Flutter components and animations to enhance the user experience.

▶ Widgets

1. `Scaffold`: Provides a basic structure for the page, including the app bar and body content.
2. `AppBar`: Displays the title "Select Page" at the top of the screen.
3. `Column`: Arranges its children in a vertical column.
4. `ElevatedButton`: A button widget with the label "new session" and an icon. When pressed, it creates a new problem session using `userState.createSession()` and sets the session ID in `problemSessionState` before navigating to the appropriate page using `context.pushReplacement`.
5. `Consumer`: A widget that rebuilds its child whenever `UserState` changes. It listens for changes in the `problemSessionList` and updates the UI accordingly.
6. `Expanded`: Takes all available vertical space in the `Column` for the list of problem sessions.
7. `ListView.builder`: A scrollable list that displays problem sessions fetched from `problemSessionList`. Each item in the list is represented by a `ListTile`.
8. `ListTile`: Represents an individual problem session in the list. It displays the school's name and an icon corresponding to the session status.
9. `IconButton`: Displays a delete icon. When pressed, it deletes the corresponding problem session using `userState.delete(element.id)`.
11. `onTap`: A callback function for each `ListTile`. When tapped, it sets the problem session ID in `problemSessionState` and navigates to different pages based on the session status.
12. `_rotationController`: An `AnimationController` used for the infinite rotation of the switch icon when a session is "STARTED."

▶ Functionality

- When the "new session" button is pressed, it creates a new problem session using `userState.createSession()` and sets the session ID in `problemSessionState`. The app then navigates to the `"/select/session"` page.
- Each `ListTile` represents a problem session in the `ListView.builder`. It displays the `school` and an icon representing the session status. The delete icon calls the `deleteSession` the corresponding problem session.
- When a problem session in the list is tapped, it sets the problem session ID in `problemSessionState`. Depending on the session status, the app navigates to different pages: `"/select/calendar"` if the status is "STARTED" or "SOLVED," and `"/select/session"` for other statuses.

SessionPage.dart

The SessionPage contains two main widgets: ProblemSessionPage, represents a page where users can manage problem sessions, and `UnavailViewer` that is contained in the first one that display a list of Unavail that belongs to those sessions.

▶ Widgets

1. **`Scaffold`**: Provides a basic structure for the page, including the app bar and body content.
2. **`AppBar`**: Displays the title "Session Page" at the top of the screen, along with a close icon that navigates back to the previous page.
3. **`Consumer`**: Listens for changes in **problemSessionState** and updates the UI accordingly.
4. **DropDownButton**: A dropdown menu to select a school from a list of predefined options ("AUIC", "Ing_Ind_Inf", "ICAT", "Design"). It sets the selected school in the **problemSessionState**. This widget requires a initial value not null to be visualized and it's set to "Ing_Ind_Inf".
5. **`ElevatedButton`**: When pressed, opens a *date range picker* to select session start and end dates, which are then set in the **problemSessionState** in the **sessionDates DateTimeRange**.
6. **`ListView.builder`**: Displays a list of unavailability associated to professors fetched from **unavailList** properties of **problemSessionState**. On click of each **'ListTile'** element, navigates to the "/select/unavail" page loading the corresponding Unavail to be visualized in *UnavailPage*.
7. **`FloatingActionButton`**: A small floating button with the "+" icon to add new unavailabilities. When pressed, it sets the current **unavailID**, trigger the **setCurrID** function to create a new unavail giving in input to an empty string in the *UnavailState* and navigates to the "/select/unavail" page.
8. **`ElevatedButton`**: When pressed, starts the optimization process if all required session settings are defined and navigates to the "/select/calendar" page. It displays different messages based on what properties of session settings is missing.

UnavailPage.dart

This widget represents a page where users can add and manage a single element of the **problemSessionState** for a selected problem session. Users can choose from three options to add unavailabilities: a single day, a date range, or a recurrent day of the week within a specific range. The widget provides options to select the type ("professor" or "Politecnico") and the name of the associated entity.

▶ Widgets

- **`Scaffold`**: Provides a basic structure for the page, including the app bar and body content.
- **`Consumer`**: Listens for changes in **UnavailState** and updates the UI accordingly.
- **`DropDownButton<int>`**: Allows users to select the type of unavailability (professor or Politecnico) and updates the selected type in the **UnavailState**.
- **`AutoCompleteProfessor`**: A custom widget that provides an autocomplete feature for selecting professor names. It is used when the unavailability type is set to "professor." When a name is selected, it updates the selected name in the **UnavailState**. It's composed by the following components:
 - **`Autocomplete<String>`**: The autocomplete widget that allows users to enter professor names and displays suggestions based on the **_profList** fetched in **initState**.
 - **optionsBuilder**: Provides suggestions based on the entered text and the list of professor names in **_profList**, that is the list retrieved from the backend function **getProfessorList**.
 - **onSelected**: Callback function triggered when a professor name is selected from the suggestions. It updates the selected professor name in the parent widget using the **setState** function provided by the parent widget.
- **`ElevatedButton`**: Allows users to add dates to the unavailability instance by opening date pickers or a recurrent date picker dialog, depending on the selected option. It invokes corresponding functions to add single-day, date range, or recurrent day of the week within a specific range inside the **sessionDates**.
- **`ListView.builder`**: Displays the list of dates in which the entity specified above is not available fetched from **unavailState.dates**.

- ``ElevatedButton``: When pressed, it resets the **UnavailState** and navigates back to the problem session page ("/select/session") if at least one dates has been entered. Otherwise, it shows a toast message indicating that all input fields should be completed.

SettingsPage.dart

This widget represents the settings page to configure various settings related to problem sessions. Users can input the number of calls, the current semester, the minimum distance for exams, and the default distance. Additionally, users can add exceptions to the default distance for specific exams that needs more days between calls.

▶ Widgets

- ``Scaffold``: Provides a basic structure for the page, including the app bar and body content.
- ``Consumer``: Listens for changes in **ProblemSessionState** and updates the UI accordingly.
- ``TextField``: Allows users to input the number of calls, the current semester, the minimum distance for exams, and the default distance.
- ``ElevatedButton``: When pressed, it saves the form by updating the corresponding settings in the **ProblemSessionState**.
- ``AlertDialog``: When the "Add Exception" button is pressed, it opens a dialog to add a custom distance exception for a specific exam. Users can input the exam name and distance, and the exception will be added to the list of exceptions in **ProblemSessionState**. In this Dialog we use a **AutocompleteExams** widget, similar to the one's used in **UnavailPage.dart**. It provides autocomplete functionality for selecting exam names. It receives a list of exams, fetches it once in its **initState**, and displays suggestions as users type their selection.
- ``ListView.builder``: Displays the list of exceptions fetched from **session.exceptions**.

CalendarPage.dart

This widget represents the calendar page of the **SchedulEx**. It allows users to view the exam schedule on a calendar. The page also includes functionality to download an Excel file with the exam schedule. After a **ProblemSession** is started, it fetches data from the server periodically to update the status and the progress to be visualized.

▶ Widgets

- ``Scaffold``: Provides a basic structure for the page, including the app bar and body content.
- ``Consumer``: Listens for changes in **ProblemSessionState** and updates the UI accordingly.
- ``ElevatedButton``: When pressed, it downloads the exam schedule as an Excel file.
- ``Center``: The main content of the page is displayed based on the problem session status. It shows different messages based on the status that fetches periodically or a circular progress indicator while waiting for the server response. When the status is 'SOLVED' it's visualized a **'TableResults'** that is a stateful widget representing the table results (exam schedule) for the selected problem session. It displays a table calendar that allows users to select a single day or a date range and view the exams scheduled for that day or range.:
 - ``TableCalendar``: Displays the table calendar with options to select a single day or a date range. It listens for date selection events using **_onDaySelected** and **_onRangeSelected**.
 - **_selectedExams**: is a *ValueNotifier* that holds the list of exams scheduled for the selected day(s). It is updated whenever a new day or date range is selected.
 - **_getExamsForDay** and **_getExamsForRange**: Helper functions to get the list of exams scheduled for a single day or a date range.
 - ``ListView.builder``: Displays the list of exams for the selected day(s) using a *'ValueListenableBuilder'*.

▶ Functionality

- **Timer**: The **_pollingTimer** is used to fetch the problem session **status** periodically from the server. It is initialized in the **initState** and cancelled when the problem session status is not 'STARTED'.
- **stopPolling()**: A method to stop the polling when the page is disposed or the problem session status is not 'STARTED'.

BACK-END

It is chosen to use Python in combination with Flask for the back-end development of the application for several reasons. Firstly, Python is a versatile and powerful programming language that offers a wide range of libraries and frameworks for web development. Its intuitive and readable syntax makes writing code easier and more understandable.

Flask, on the other hand, is a lightweight and flexible framework that integrates perfectly with Python. It made it possible to create a RESTful API for the back-end quickly and efficiently. Flask offers a modular architecture, allowing specific endpoints to be created to handle requests according to the application's needs.

Another reason why it is chosen Python and Flask it is the large developer community that supports them. This made it possible to have access to a wide range of documentation more efficiently.

Python and Flask also offer great flexibility in the choice of databases. It is possible to easily integrate different relational or NoSQL databases into back-end, allowing to adapt to the specific needs of the application.

Finally, scalability is another advantage. It is possible to handle a high volume of requests without compromising the performance of the application.

For the back-end a config.xml file is present in order to be able to change easily some values as file path if necessary.

backend.py

This script defines a Flask web application to manage optimization sessions with data stored in a Firebase Realtime Database. The web application provides several API endpoints to handle various tasks related to optimization sessions. Now the implementation step-by-step is explained:

The necessary libraries/modules are: **json** (for JSON data handling), **os** (for operating system-related functions), **utils** (custom utility functions), **Flask** (the Flask web framework for creating the application), **request** (for handling HTTP requests in Flask), **send_file** (for serving files as responses), **jsonify** (for JSON response creation), **firebase_admin** (for interacting with Firebase services), **datetime** (for working with dates and times), **db** (Firebase Realtime Database module for database operations), **pandas** (for data manipulation and analysis), **Thread** (for multi-threading functionality), **openpyxl** (for working with Excel files), **Optimization_Manager** (custom module for optimization management).

Set up Firebase credentials and initialize the Firebase app:

The script uses a service account key file named "schedulex-723a8-firebase-adminsdk-mau2x-c93019364b.json" to access Firebase services.

The app is initialized with the given credentials and a custom app name "ScheduEx".

The Firebase Realtime Database URL is set to "https://schedulex-723a8-default-rtdb.firebaseio.com/".

Create a Flask app instance:

The Flask app is created using Flask(__name__).

A reference to the Firebase Realtime Database is created using db.reference("/").

Implement various API endpoints for the web application, including:

▶ startOptimization

Starts the optimization process for the specified session ID.

Retrieves the list of all session IDs from the Firebase database and updates the status list.

Checks if the optimization process is already running for another session, and if not, it sets the status to "STARTED" and starts a new optimization thread using runOptimizationManager as the target function.

▶ `askStatus`

It retrieves the status and progress of the optimization process for the specified session ID from the `status_list` and returns the result as a JSON response.

▶ `setUserID`

It sets the user ID for a specific session in the Firebase database.

▶ `setStartDate`

It sets the start and end dates for a specific session in the Firebase database.

▶ `getSessionList`

It retrieves a list of all sessions from the Firebase database and returns the result as a JSON response.

▶ `getSessionData`

It retrieves the data for a specific session from the Firebase database and returns the result as a JSON response.

▶ `getUnavailData`

It retrieves unavailability data for a specific session and unavailability ID from the Firebase database and returns the result as a JSON response.

▶ `setSettings`

It saves optimization settings for a specific session in the Firebase database.

▶ `saveUnavailability`

It saves unavailability data for a specific session in the Firebase database.

▶ `delete_unavail`

It deletes unavailability data for a specific session and unavailability ID from the Firebase database.

▶ `deleteUnavailabilityDate`

It deletes a specific date from the unavailability data for a specific session in the Firebase database.

▶ `getProfessorList`

It retrieves a list of professors from an Excel file named "Database esami_Ing_Ind_Inf.xlsx" and returns the result as a JSON response.

▶ `getExamList`

It retrieves a list of exams from the same Excel file and returns the result as a JSON response.

▶ `saveSession`

It saves session data in the Firebase database.

▶ `deleteSession`

It deletes a specific session from the Firebase database.

▶ `downloadExcel`

It allows users to download an Excel file containing the optimization results for a specific session.

▶ `getJSONresults`

It retrieves the optimization results in JSON format for a specific session and returns the result as a JSON response.

The main block of the script runs the Flask application with debugging enabled.

Optimization_Manager.py

Here it is implemented the management of the optimization process for exam scheduling. The script imports necessary modules like "firebase_admin," "pandas," "utils," and "optimizer," as well as "datetime" and "time" modules. The functions can takes as input different parameters; they are **cds_id** (programme study id), **percentage** (a value that take into account of the progression of the optimization), **sessionID** (id of the problem session to work with), **school** (school name to work with), **ExamList** (list of exam to work with), **unprocessedExamList** (list of exam to

schedule), **semester** (value of semester for the scheduling exam session), **problem_session** (data stored in Database Problem) and **callback** (function to call after the optimization process). These inputs are defined in the core function `runOptimizationManager`.

▶ `getDatabaseProblemData`

It takes `sessionID` as input.

It retrieves the data related to the scheduling problem from Firebase.

It extracts the data from `unavail_list`, which contains the unavailable dates for exams, and the settings related to exam weights and minimum distances.

The data is used to create a `ProblemSession` object and returned as the result.

▶ `getDatabaseExam`

It takes `cds_id`, `sessionID`, `school` and `percentage` as input.

It retrieves exam data from an Excel file based on the school and course code (`cds_id`)

It creates a list of "Exam" objects containing details such as course code, semester, location, professor, etc.

▶ `createOptExamList`

It takes as input `ExamList`, `sessionID` and `percentage`.

With the list of "Exam" objects creates a corresponding list of "optExam" objects.

It initializes the additional attributes for optimization purposes like `effortWeight`, `timeWeight`, and `assignedDates`.

▶ `createWeight`

It takes as input `unprocessedExamList`, `semester`, `sessionID` and `percentage`.

It calculates `effortWeight` and `timeWeight` for each "optExam" object based on its attributes and semester. The results are updated within the "unprocessedExamList."

▶ `addDistances`

It takes as input `unprocessedExamList`, `problem_session`, `sessionID` and `percentage`.

It adds information about minimum distances for exams and calls from the `problem_session` settings to each "optExam" object in the "unprocessedExamList."

▶ `addUnavailability`

It takes as input `unprocessedExamList`, `problem_session`, `sessionID` and `percentage`.

It merges the unavailability data (professors' or university unavailability) from the `problem_session` settings with each corresponding "optExam" object in the "unprocessedExamList."

▶ `runOptimizationManager`

It takes as input `sessionID` and the callback function defined in backend script.

It manages the overall optimization process.

It iterates through various schools' courses (CdS), retrieves data and performs optimizations.

It uses a callback function to handle the results of the optimization process.

The code uses the global "status_list" object to manage and track the progress of the optimization process. It updates the progress and status at different stages of the optimization using the "setStatus" and "setProgress" methods defined in the "optStatus" class.

▶ `solveScheduling`

It takes as input `unprocessedExamList` after weight, unavailability and distances addition and `problem_session`

It is used to perform the actual optimization of exam scheduling, it is explained in `Optimization.py`

In summary, the code retrieves problem and exam data from the Firebase database and an Excel file, respectively. It then creates a list of "optExam" objects for optimization, calculates weights and distances, and incorporates unavailability data. Finally, it performs an optimization process using the "solveScheduling" function and updates the status and results accordingly. The overall process aims to schedule exams efficiently and handle unavailability constraints.

Optimizer.py

This code implements a scheduling problem to assign exam dates optimally for a set of exams. It solves the problem as an Integer Linear Programming (ILP) using the MIP (Mixed Integer Programming) library with the GUROBI solver.

The necessary libraries are imported: mip, numpy, pandas, datetime, timedelta, and utils.

▶ solveScheduling

This function takes as input **exams**, a list of *optExam*, and **problem_session**, which is a Problem Session object in which are defined all the parameters that need to be included in the problem.

Now, like we did in the Design Document, we address to the constraint's implementation for the problem.

The major problem that we encounter is the definition of the function that indicates how any days, two assigned dates distances between each other, since it must not be linear to be included in a Linear Programming model. To do so, we create the following variables:

- $s_{i,k}$ which is equal to 1 if exam i is assigned to day k , 0 otherwise.
- $x_{i,k,j,t}$ which is equal to 1 if the exam i is assigned in the k date AND if the exam j is assigned in the t , 0 otherwise.
- D is the total number of days between start and end date computed and added in the **availDates** list. k and t indicate the indexes of elements of this list.
- N is the length of the list of *optExam* given in input. i and j indicate the indexes of elements of this list.

The objective function is:

$$\max \sum_i^N \sum_j^N \sum_k^D \sum_k^D e_i.effortWeight * e_j.effortWeight * e_i.timeWeight[j] * x_{i,k,j,t}(t - k)$$

(1)

The constraints are formulated as following:

- It's introduced a logical constraint to connect the decision variable and the auxiliary variable. This is a linear implementation of the logical operator AND.

$$\forall i, j, k, t \quad \text{con} \quad t > k \quad 0 \leq s_{i,k} + s_{j,t} - 2x_{i,k,j,t} \leq 1$$

- Moreover, it's needed to set the number of dates assigned to a exam equal to the settings parameter **numcalls**. This is obtained with this constraint:

$$\forall i \quad \sum_k^S s_{i,k} = numCalls$$

- The constrained is now focus on each element of the assigned dates of the single exam:

$$\forall i, j, k, t \quad \text{con} \quad t > k \quad x_{i,k,j,t}(t - k - minDistanceExamToExam) \geq 0$$

- For each exam it must be verified that elements of **assignedDates** distance from each other the number of days specified in input by the user for that exam or the standard value of 14 days.

$$\forall i, k, t \quad \text{con} \quad t > k \quad x_{i,k,i,t}(t - k - e_i.minDistanceCalls) \geq 0$$

- For the unavailability, we set to zero the variable that involves those dates, that are included in unavailDates property for each exam, in those dates.

$$\forall i, k \quad \text{con} \quad \text{date}_k \in e_i.unavailDates \quad s_{i,k} := 0$$

In a similar way are also set the unavailability related to Sunday day and when a exam has already been scheduled in the previous iteration, but setting the value of the variable to 1

Once the problem is solved or not the assignedDates property is updated for each exam of the initial list based on the obtained results. Then this results are returned with the status of the optimization accordingly.

Utils.py

In this file the code defines several classes and helper functions related to managing exam scheduling and status tracking. Each class and its purpose is explain in the following paragrhaps.

1. **Unavail class** represents unavailable time slots for exams. Its attributes are 'id', 'type', 'dates' and 'name' that are respectively a unique identifier for the unavailability, type of unavailability (professor/university), list of specific dates when the unavailability occurs and name of the unavailability (for example name of a professor). Moreover, there is the method 'toString' that returns a formatted string representation of the object.
2. **ProblemSession class** represents a session where exams are scheduled and optimized. It's attributes are 'id' (unique identifier for the session), 'school' (name of the school), 'status' (status of the session), 'description' (description about the problem session), 'user', 'startDate' (start date of the session), 'endDate' (end date of the session), 'unavailList' (list of Unavail objects representing unavailable time slots for exams) and 'settings' (distances between calls or exams for the session).
3. **Exam class** represents an individual exam. Its attributes hold specific information about the exam and are the one contained in Database Exam. It has also the method 'toString' that returns a formatted string representation of the object.
4. **optExam class** (inherits from Exam class) represents the exam object for the optimization. Inherits all attributes from the Exam class and adds the following attributes: 'unavailDates' (list of dates that are unavailable for scheduling the exam), 'effortWeight' (weight associated with the effort required for the exam., 'timeWeight' (weight associated with the scheduling time of the exam), 'minDistanceExams' (minimum distance between exams in terms of time. 'minDistanceCalls' (minimum distance between exam calls, 'assignedDates' (dates when the exam is scheduled). As the Exam class it ha a method 'toString'.
5. **optStatus class** represents the status and progress of an optimization session. Its attributes are '__progress' (private), '__status' (private) and 'sessionID', that respectively indicates tracking of the progress of the session, hold the status of the session and identifier for the optimization session. It has several methods: getStatus, getProgress, setStatus, setProgress toString. The last one is the same of the previous classes while the 'get...' method retrieves the status or the progress and the 'set...' update the status or the progress (the status in Firebase database).
6. **Sessions_status_list class** manages a list of optStatus objects and provides methods to retrieve and update their status and progress. As attribute it has a list of optStatus objets while the provided method are 'getStatus' to get the status of an optimization session using its sessionID, 'getProgress' to get the progress of an optimization session using its sessionID 'setStatus' update the status of an optimization session using its sessionID in the Firebase database, 'setProgress' to update the progress of an optimization session using its sessionID and 'toString' that returns a formatted string representation of the object.
7. Finally, the code creates an empty **Sessions_status_list** object named status_list to manage the status of various optimization sessions.

DATABASES

A Firebase database was chosen for the Database Problem for several reasons.

Extensive documentation is available regarding its use, so it was easy to understand its use.

For this situation we didn't need a relational database as in fact firebase isn't and it allowed us to save the data with a structure that was convenient for the needs of the problem to be faced.

Regarding the Database Exam, an excel file was chosen even if it is not a real database. This choice is due both to the fact that the data concerning the exams were already present in an excel file but also because it was easier to reorganize them according to needs. This was an important factor because during the development of the application they had to be rearranged several times.

Furthermore, Python offers useful libraries for working with Excel files and this has improved software development.

Database Problem

This is the structure of the database on firebase, data are read and written by specific function of backend.py depending on which action is needed to do.

sessionID

```
description: "string",
startDate: "string",
endDate: "string",
school: "string",
settings: {
  currSemester: int
  minDistanceCalls:
    Default: int,
    Exception:
      ID: {
        distance: int,
        id: "string"},
  minDistanceExams: int,
  numCalls: int},
status: "string",
unavailList:
  ID: {
    dates
      0: "string"
      .....
    name: "string",
    type: "string"},
userID: "string"
```

The meaning of each field is explained in Design Document

Database Exam

This database is structured as a table. Here a picture of how it appears.

| Course Name | Semester | Year | SEM | Location | Exam Head | Professor | Section | Enrolled number | CFU | Passed % | Average Mark |
|--|----------|------|-----|----------|-----------|---|---|-----------------|-----|----------|--------------|
| CALCOLO DELLE PROBABILITÀ E STATISTICA | 1 | 3 | 5 | MI | MAT1 | Ladelli Lucia Maria-Scarpa Luca | A M-M ZZZZ | 176 | 5 | 10 | 29 |
| FONDAMENTI DI INFORMATICA | 1 | 1 | 1 | MI | ELN1 | Bolchini Cristiana-Negri Mauro- Braga Daniele Maria-Miele Antonio Rosario-Loiacono Daniele- Caglioti Vincenzo-Matera Maristella-Mirandola Raffaella | A BRA-BRA COM-COM FEI- FEI IMA-IMA MEZ-MEZ PEZ- PEZ SAZ-SAZ ZZZZ | 176 | 5 | 10 | 29 |
| SISTEMI INFORMATICI | 1 | 2 | 3 | MI | ELN1 | Gatti Nicola-Mottola Luca | A M-M ZZZZ | 176 | 5 | 10 | 29 |
| FONDAMENTI DI AUTOMATICA | 2 | 2 | 4 | MI | ELN1 | Tanelli Mara-Piroddi Luigi | A M-M ZZZZ | 176 | 5 | 10 | 29 |

For each line there is an exam, and for each parameter there is a column.

2 - REQUIREMENTS IMPLEMENTED

All the requirement specified in RASD document are implemented, so the SchedulEx let to log in and visualize all the sessions that has been created (R4), but also create (R1), modify (R2) or delete (R3) a session to compute the Optimal Calendar associated to a specific School. These requirements are implemented with **setUserID** and **getSessionList** for what concerns the backend and all the functionalities included in the **SelectPage** for what concerns the front end.

Once the session is chosen or created the user can insert School name (R5) for which the session is defined, *insert (R6) or change (R7)* exam session dates and insert comments (R8) for the relative session. This is possible thanks backend functions **saveSession** and **setStartDate** and through the **SessionPage** in the front end.

Regarding the unavailability, thanks to **SaveUnavailability** and **delete_unavail** the user can add (R9) or delete (R10) an unavailability, then he can select the type (R11), insert the appropriated professor or classroom name (R12 and R13).

Moreover, the possibility to add/delete a single day, a multiple day or a recurrent day of unavailability (R14, R15, R16, R17, R18 and R19) **deleteUnavailabilityDate**. All these features are possible through the **UnavailPage** in the front end.

While the code for the satisfaction of requirement that deal with the distance between the distance between calls and exam (R20, R21 and R22) is the backend function **setSettings** and can be inserted and modify through the **SettingsPage**, in particular the exceptions for each exam with **addException**.

Finally, due to Optimization_Manager (for the back-end part) and fromfor the front-end part), the requirements of optimization process (R23, R24 and R25) are implemented.

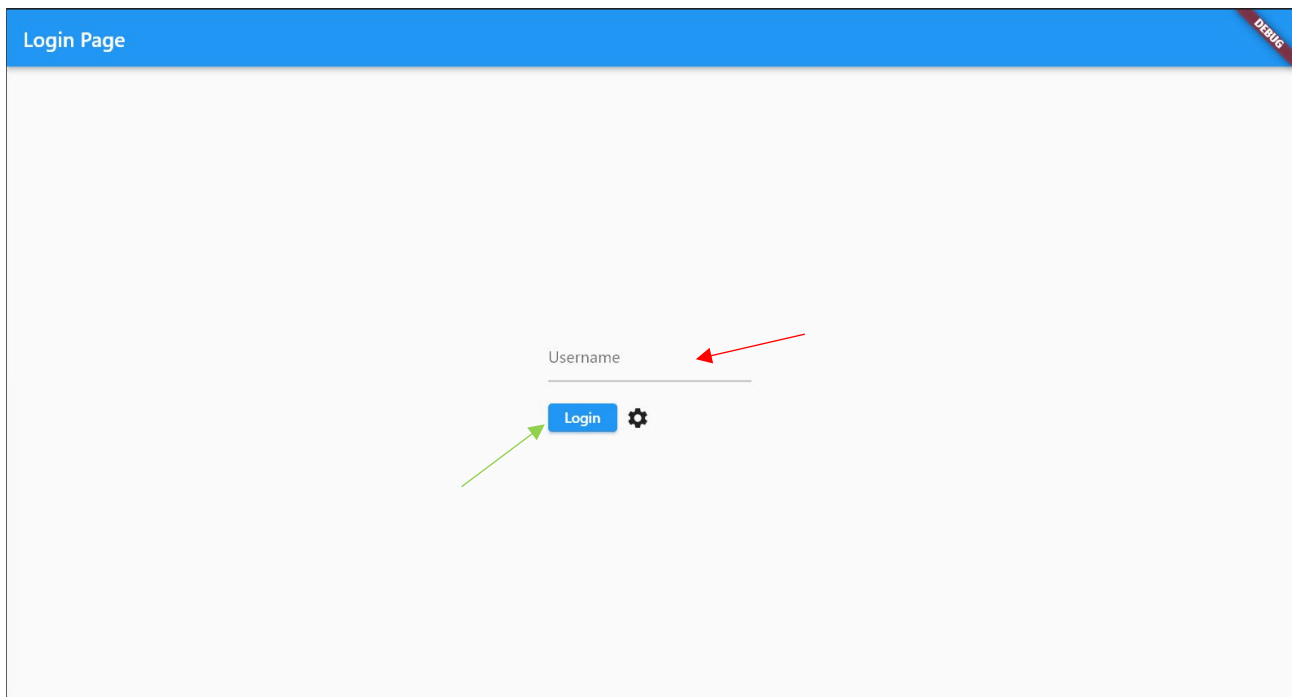
3 - USER MANUAL

In this section are listed all the actions that the user can do working with the software, these action regards the interaction user-webapp. Some of them are mandatory, other can be chosen by the user.

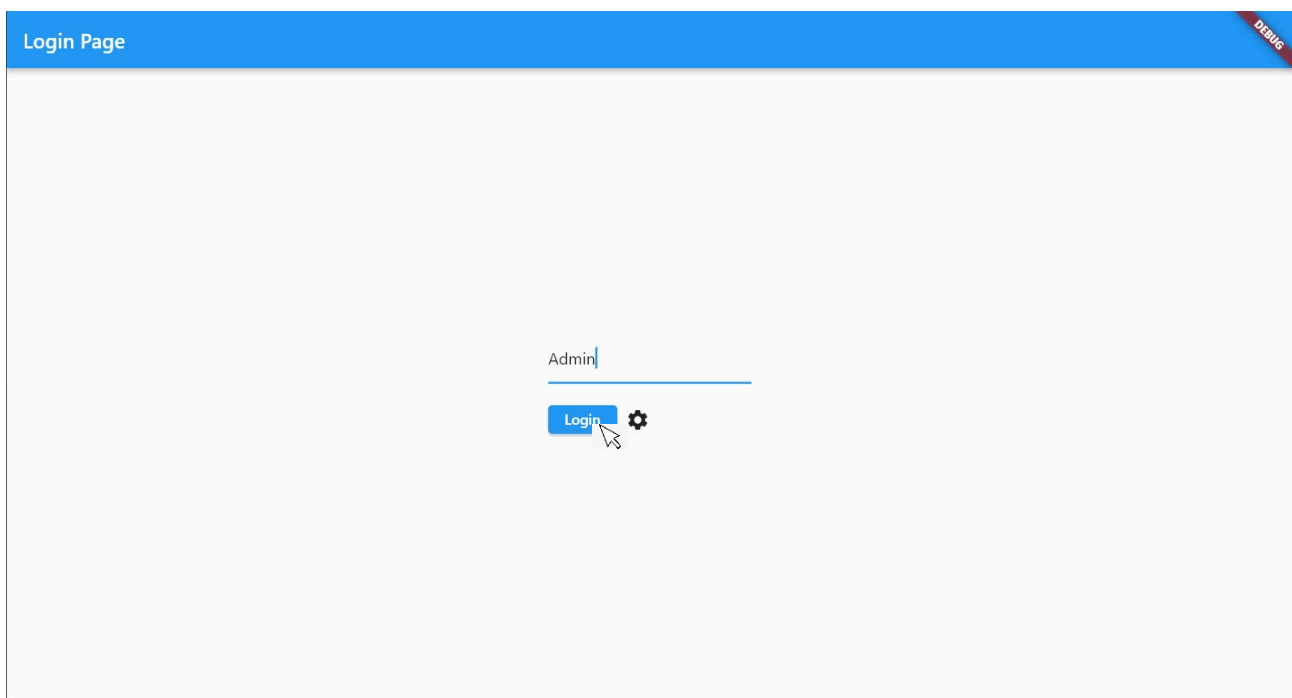
User Log in

First of all, the page that appears when the user opens the webapp is the login page in which he must insert the User ID to log in, to proceed the user must press the Log in button. These two actions are shown in the following figures.

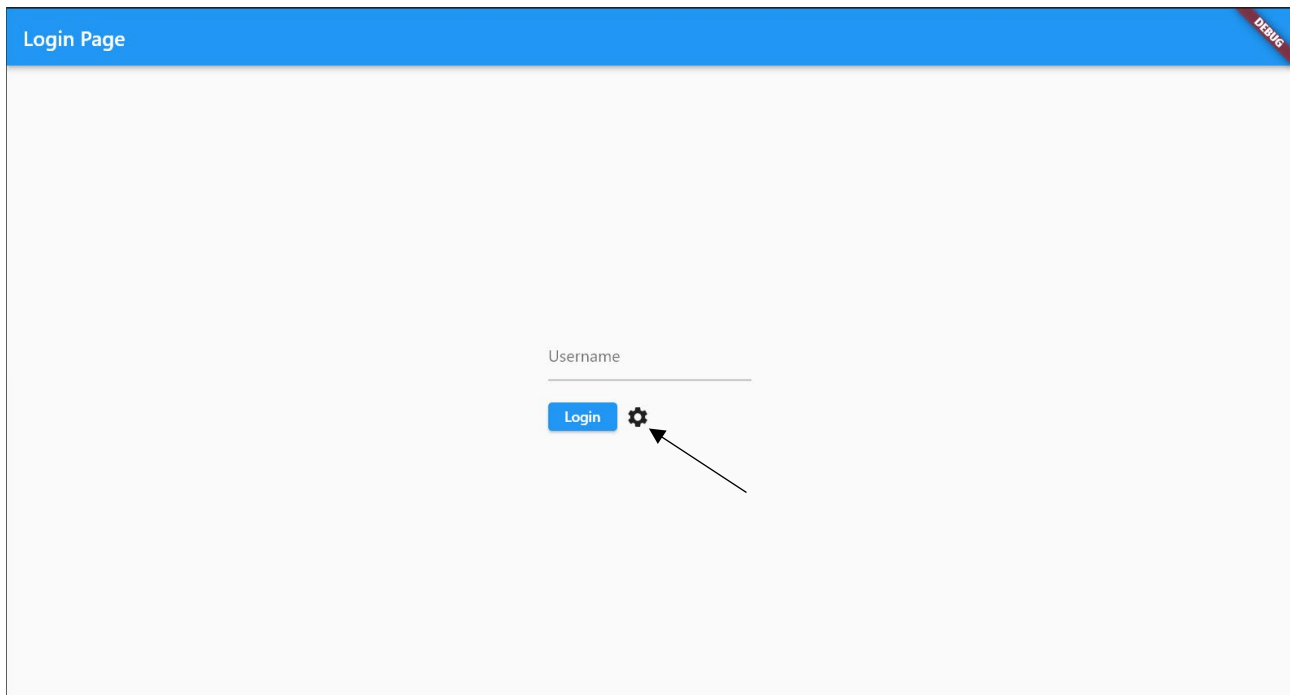
This is the login page; the field indicated with the red arrow is the one in which the user must insert the User ID while the green arrow indicates the button to click to log in.



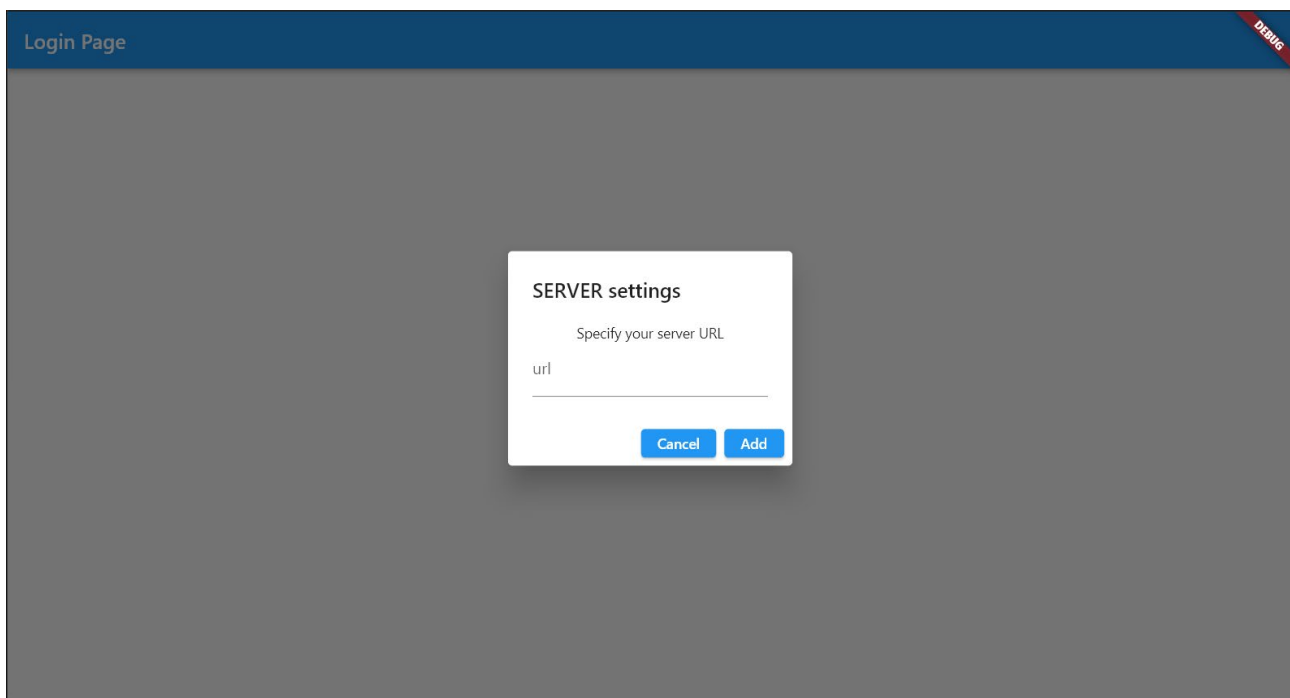
This is how the login page appear with the User ID inserted.



It is important to notice that in this page there is also a button for server url setting. The one indicated with the black arrow in the following figure.



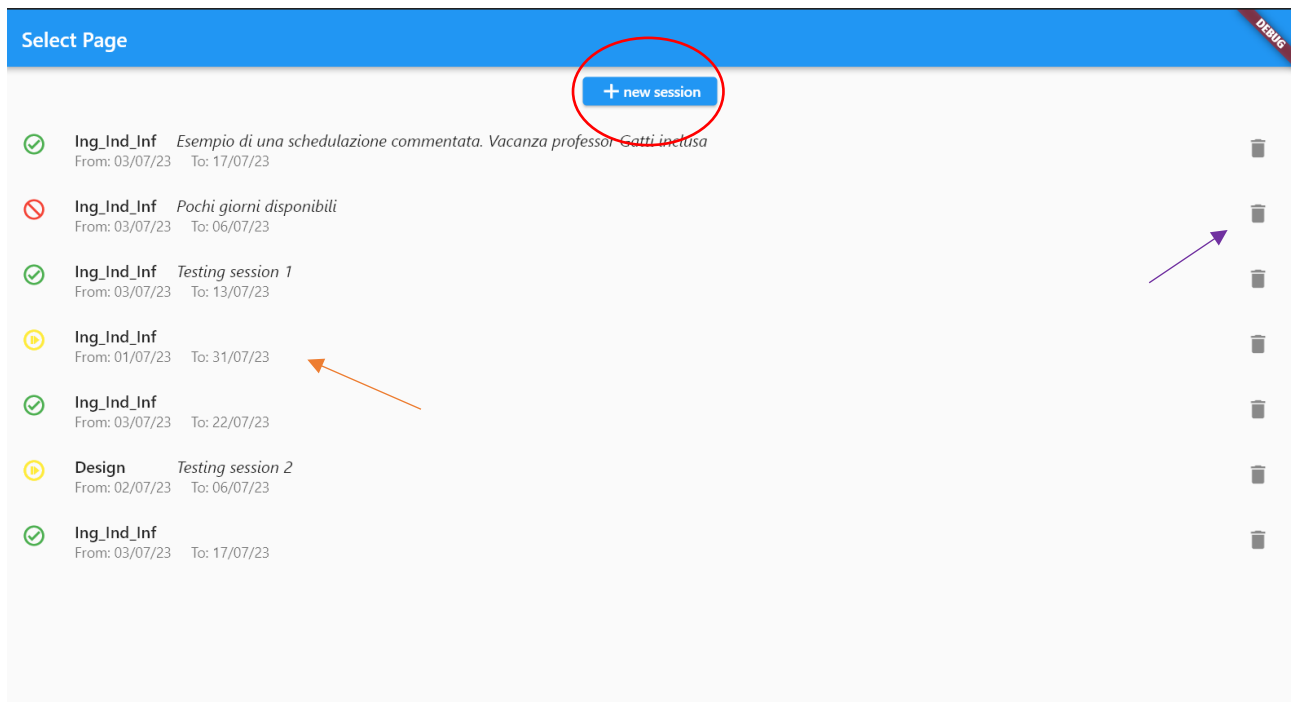
Once clicked it will appear this window in which is possible to enter the server url where the back-end is hosted.



Once written, it is necessary to press 'Add' button.

Session of work management

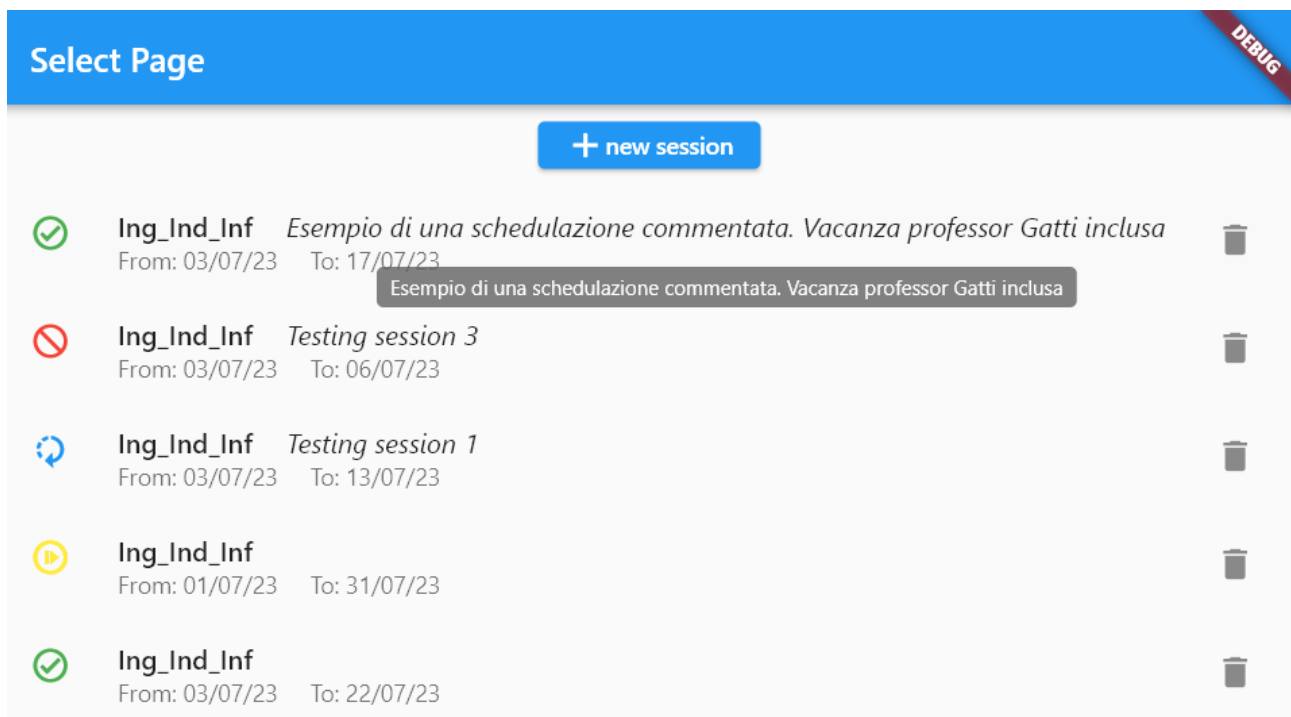
After the click on the Login button this page is shown



Now the User is in the page where all the sessions already created are listed.

For each session he can see the name of the school, a comment and the start and end date. These values identify the problem session in the session page.

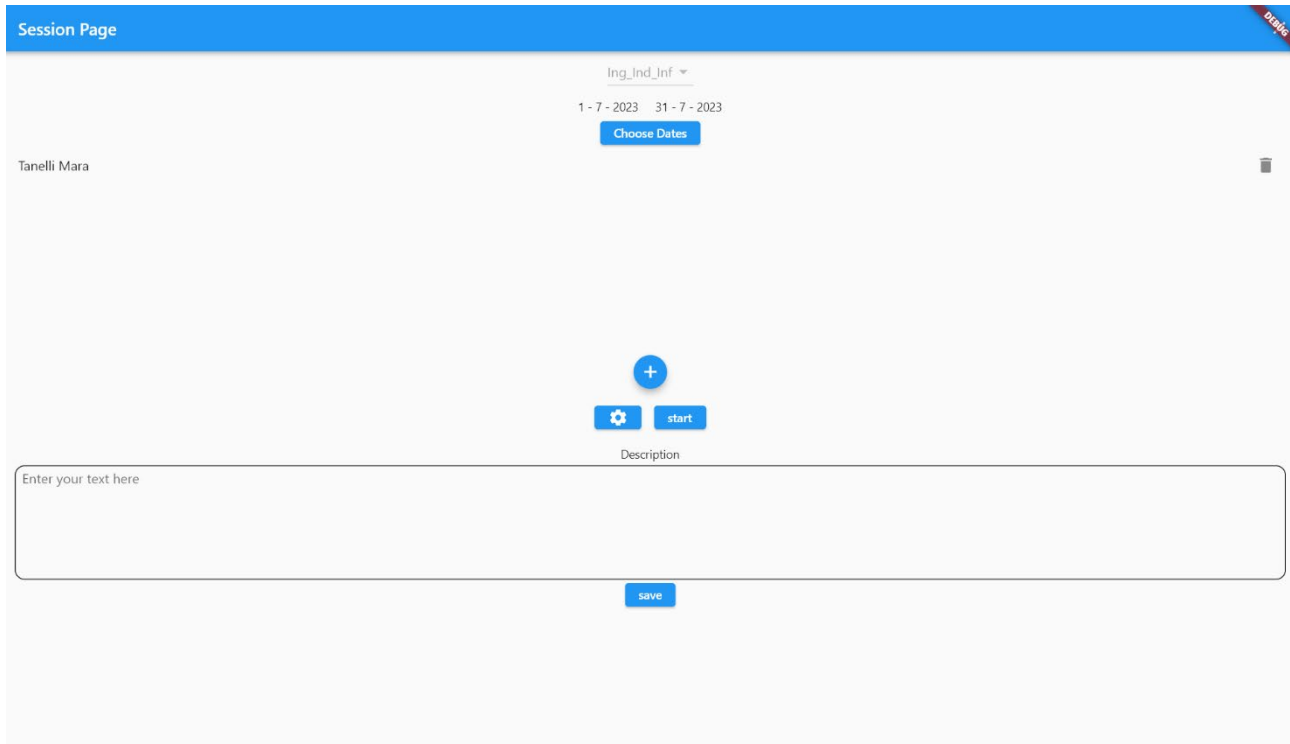
He can open an existing one (indicated by the orange arrow) clicking on it, delete it clicking on the button on the right related to the desired session (basket button, indicated with the purple arrow). Otherwise, the user can create a new session with the specific button on top of the page (red circle). Moreover, on the left of the session the user can see if the session is working, so if its optimization is started. The yellow icon means that the session is not started, the optimization of the session is successfully completed if the icon is green while the red symbol means that the schedule is finished but it has not performed a result. The blu icon as in the following picture means that the optimization is in progress.



School and exam session dates and selection

When a session is selected or it is created a new page appears, it is the 'Session Page'.

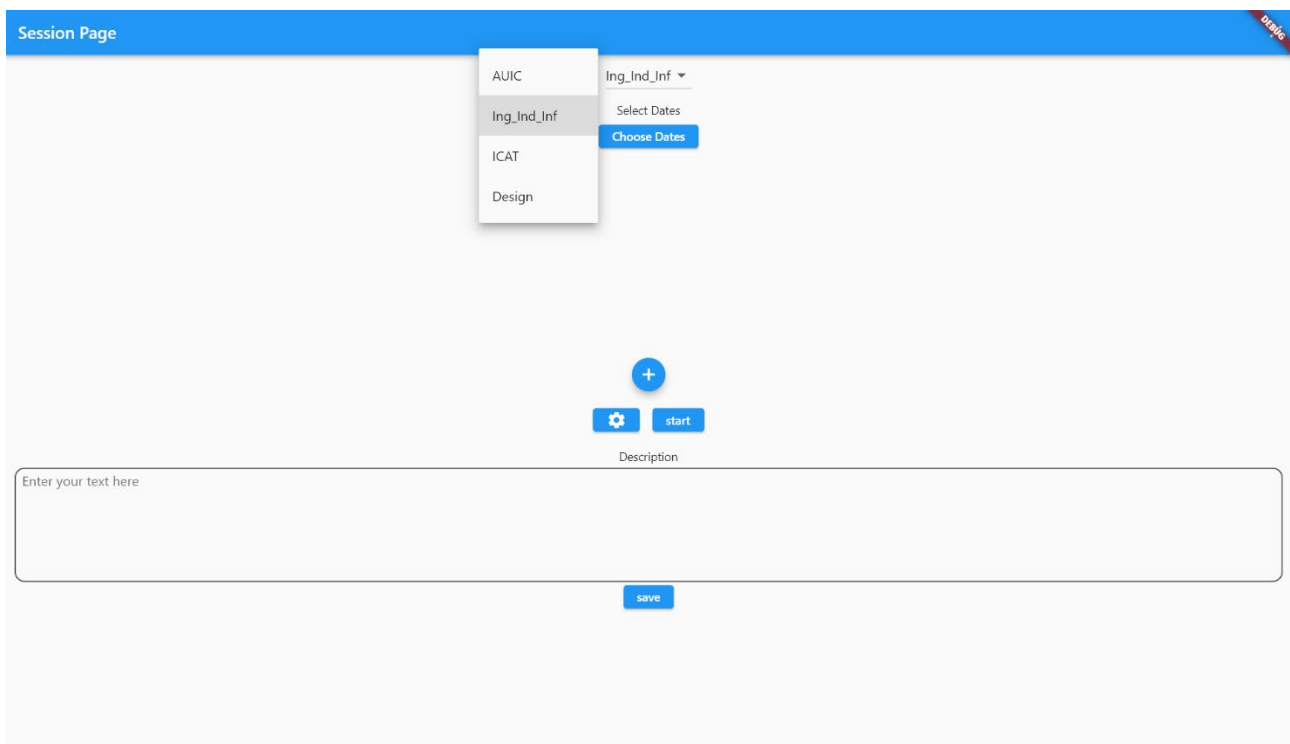
This page appears if the problem session is new, is not scheduled (yellow icon near the session), or it is scheduled but with no solution (red icon near the session).



The screenshot shows the 'Session Page' interface. At the top, there is a blue header bar with the text 'Session Page' and a red 'draft' label in the top right corner. Below the header, the page displays the session name 'Ing_Ind_Inf' with a dropdown arrow, the dates '1 - 7 - 2023' and '31 - 7 - 2023', and a 'Choose Dates' button. The user's name 'Tanelli Mara' is visible on the left. In the center, there is a blue circle with a white plus sign, a settings gear icon, and a 'start' button. Below these is a large text input field with the placeholder 'Enter your text here' and a 'save' button. The page has a light gray background and a clean, modern design.

While if the session is successfully scheduled, will appear a calendar page with the calendar of the scheduled exams with the possibility to download the Excel file of the calendar (clicking on the specific button indicated with the red circle). It is explained later in a dedicated paragraph.

Now, if the session is new, it is possible to choose the school in the list on the top of the page clicking on it.



This screenshot shows the 'Session Page' interface with the school selection dropdown menu open. The dropdown menu lists four options: 'AUIC', 'Ing_Ind_Inf' (which is highlighted), 'ICAT', and 'Design'. The 'Choose Dates' button is still visible. The rest of the interface, including the user name 'Tanelli Mara', the settings gear icon, the 'start' button, the text input field, and the 'save' button, remains the same as in the previous screenshot.

But once it is chosen and the problem session is created clicking on save button in the top part of the page it is not possible to change it.

Problem Session Savings

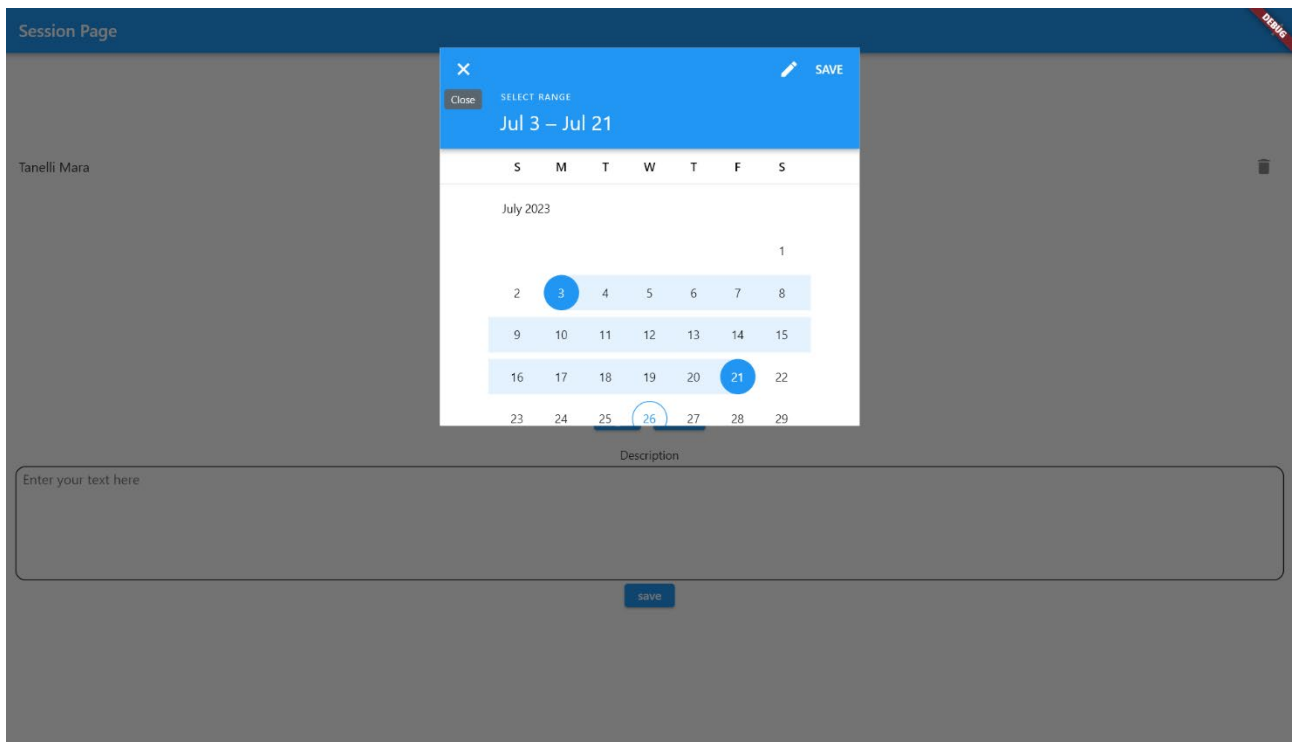
So, to save the problem session the user can click on the save button.

The screenshot shows the 'Session Page' interface. At the top, there is a blue header bar with the text 'Session Page' and a 'Back' button. Below the header, there is a dropdown menu for 'Ing_Ind_Inf' with the selected value '1 - 7 - 2023' and a date range '31 - 7 - 2023'. A 'Choose Dates' button is located below the date range. The main content area has a light gray background. On the left, the name 'Tanelli Mara' is displayed. In the center, there is a blue circular button with a white plus sign, a gear icon, and a 'start' button. Below these is a 'Description' label and a large text input field with the placeholder text 'Enter your text here'. A 'save' button is located at the bottom of the text input field, and a red arrow points to it.

Selecting start and end date

Other essential parameters that the user must insert are the start and date. He can do this clicking on 'Choose dates' and selectin the start and end dates.

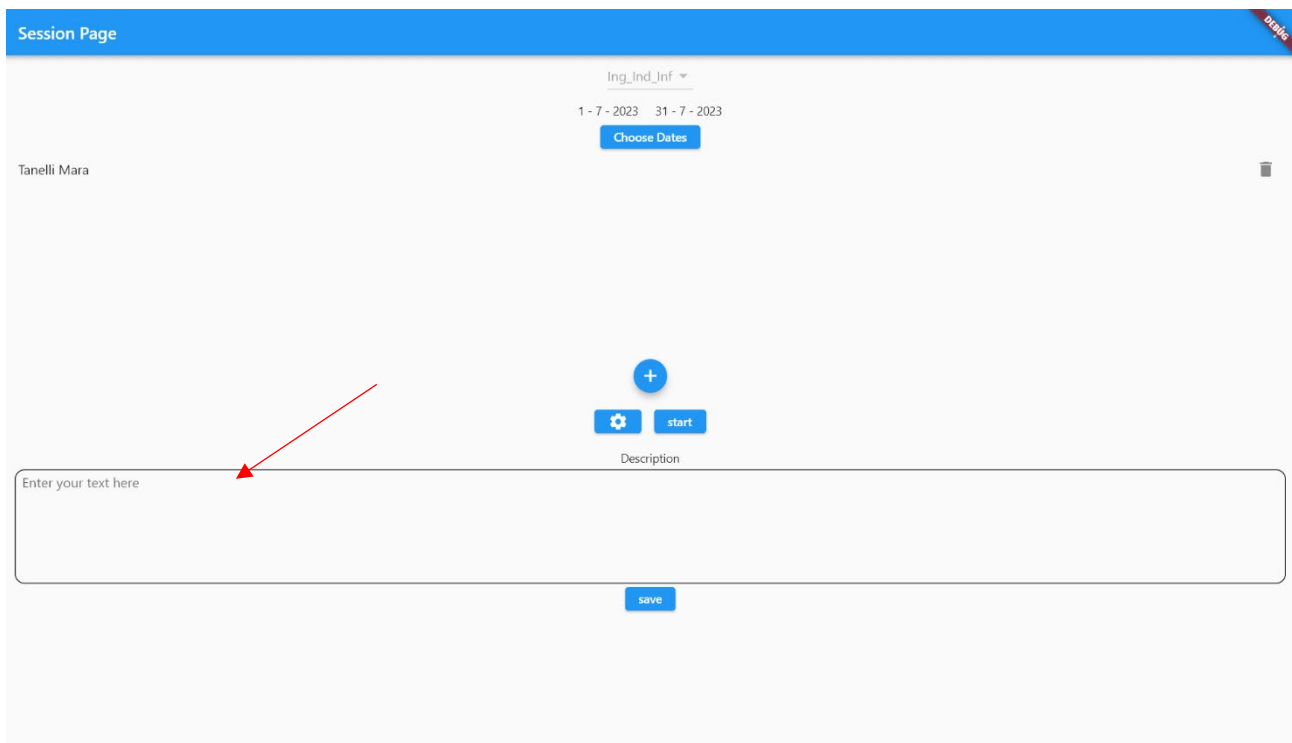
The screenshot shows the 'Session Page' interface. At the top, there is a blue header bar with the text 'Session Page' and a 'Back' button. Below the header, there is a dropdown menu for 'Ing_Ind_Inf' with the selected value '1 - 7 - 2023' and a date range '31 - 7 - 2023'. A 'Choose Dates' button is located below the date range, and a red arrow points to it. The main content area has a light gray background. On the left, the name 'Tanelli Mara' is displayed. In the center, there is a blue circular button with a white plus sign, a gear icon, and a 'start' button. Below these is a 'Description' label and a large text input field with the placeholder text 'Enter your text here'. A 'save' button is located at the bottom of the text input field.



Once the user press start the selected dates are visible above the button to choose them.

Comment adding

If the user what to add or change some comments regarding the problem session he can write them in the dedicated box.



Unavailability management

For what concern the unavailability management in the 'Session Page' are listed all the unavailability already present.

Session Page

Ing_Ind_Inf

1 - 7 - 2023 31 - 7 - 2023

Choose Dates

Tanelli Mara

+

start

Description

Enter your text here

save

If the user what to delete one of them, he can press the basket button (black arrow) related to the unavailability on the right, while if he wants to add a new unavailability he can press the '+' button (red arrow).

Session Page

Ing_Ind_Inf

1 - 7 - 2023 31 - 7 - 2023

Choose Dates

Tanelli Mara

+

start

Description

Enter your text here

save

Clicking the '+' button the Unavailability Page will appears.

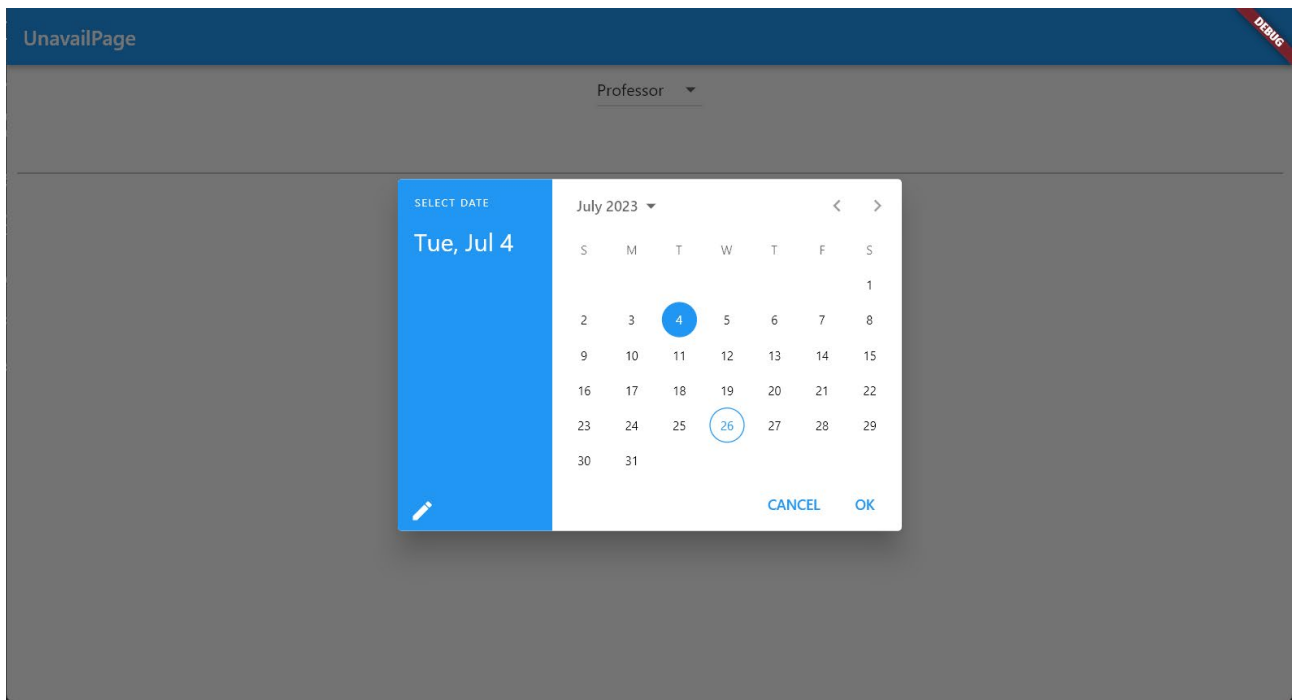
The screenshot shows a web application titled "UnavailPage" with a blue header. A dropdown menu is open, showing "Professor" and "Politecnico" options. Below the menu are three buttons: "Single Day", "Date Range", and "Open Recurrent Date Picker". A "DateTime List:" label is positioned above a large, empty white area. A "Save" button is located at the bottom center. A red "drag" label is visible in the top right corner of the header.

On the top it possible between Politecnico and Professors for what concerns the type of unavailability.

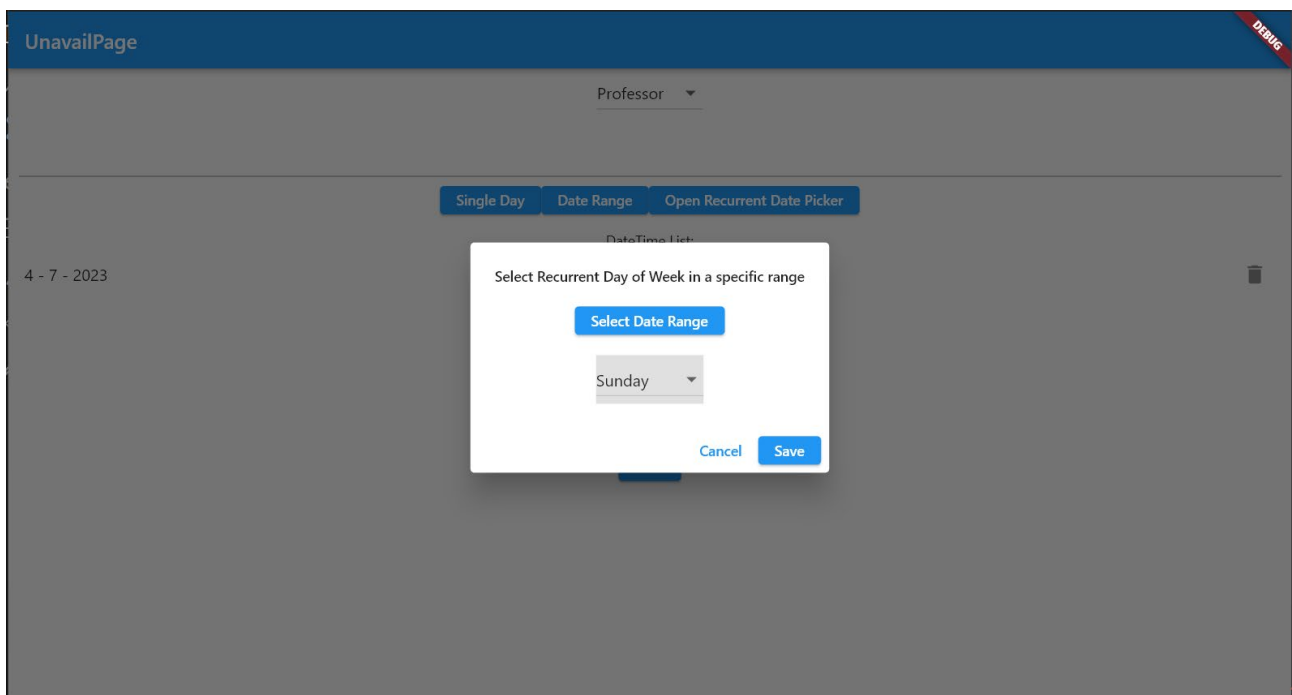
This screenshot shows the same "UnavailPage" interface, but the dropdown menu is now closed, displaying only the "Professor" option. The layout of the buttons and the "DateTime List:" area remains the same. The red "drag" label is still present in the top right corner.

Then, is possible to enter the name of the professor or the name of a specific classroom (or generic university) and select the dates of unavailability.

This is for the single day, the date range is similar to the selection of the start and end date.



This is for the recurrent day. In which it is possible to select the day and the period of the repetition.



After the unavailability are inserted the page will appear in this way

UnavailPage

Professor

Single Day

Date Range

Open Recurrent Date Picker

DateTime List:

4 - 7 - 2023

Save

Once the unavailability is inserted it is necessary to press 'Save' on the bottom part of the page

Settings management

Other parameters that are important to enter are the settings, the user can access to the settings page clicking on the button indicated in the figure below.

Session Page

Ing_Ind_Inf

1 - 7 - 2023 31 - 7 - 2023

Choose Dates

Tanelli Mara

+

⚙️

start

Description

Enter your text here

save

The settings page will appear like this.

Settings Page

number of calls

3

select current semester

2

minDistanceExams

2

Default

6

Add Exception

No Exceptions

Save

And the user can insert the value that he needs for the number of calls, the number of semester, the minimum distance between exams and the minimum distance between call (Default), but if a professor ask to have more time than the default one between his calls, the user can click on the button 'Add Exception' and this windows will appear.

Settings Page

number of calls

3

select current semester

2

minDistanceExams

2

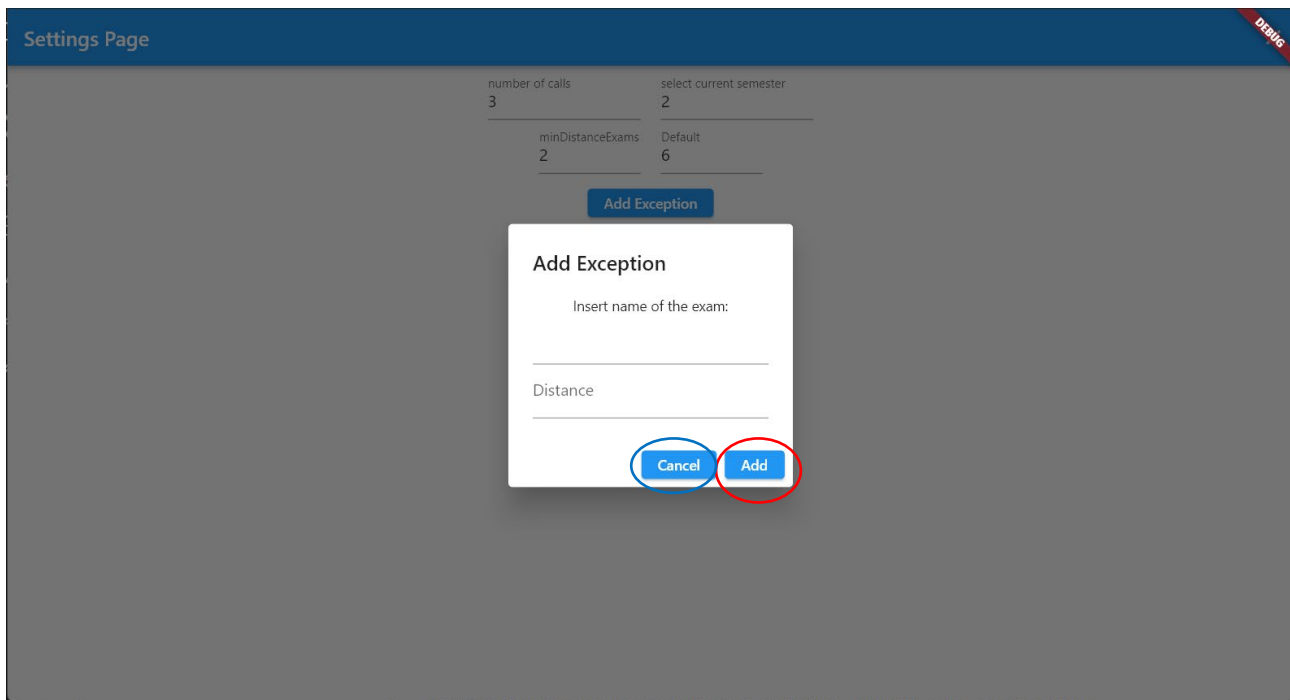
Default

6

Add Exception

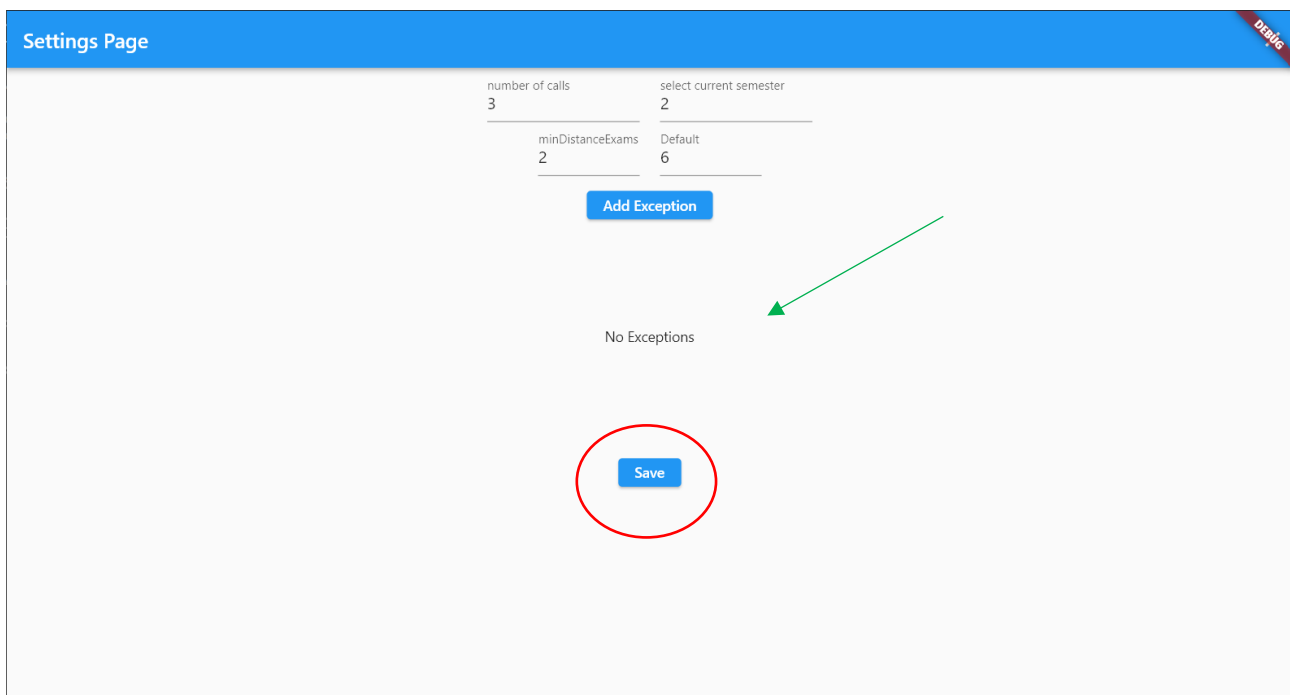
No Exceptions

Save



Here the user can put the exam name and the custom distance, after this he must click add to add it or cancel to don't add. The added exception will be shown in the centre of the Settings Page (green arrow in the figure below).

To save settings there is 'Save' button in the Settings Page (red circle in the figure below).



Calendar scheduling starting

To start the process of finding the optimal schedule of the exam the user has to press the 'start' button in the Session Page.

Session Page

Ing_Ind_Inf

1 - 7 - 202331 - 7 - 2023

Choose Dates

Tanelli Mara

+

start

Description

Enter your text here

save

But before doing this the user must have selected the school, the start and end date and insert the settings, the unavailability is optional. If the necessary data are not present will appear a box on the bottom part of the Session Page with the error when the 'start' button is pressed.

Session Page

Debug

Ing_Ind_Inf

Select Dates

Choose Dates

+

start

Description

Enter your text here

save

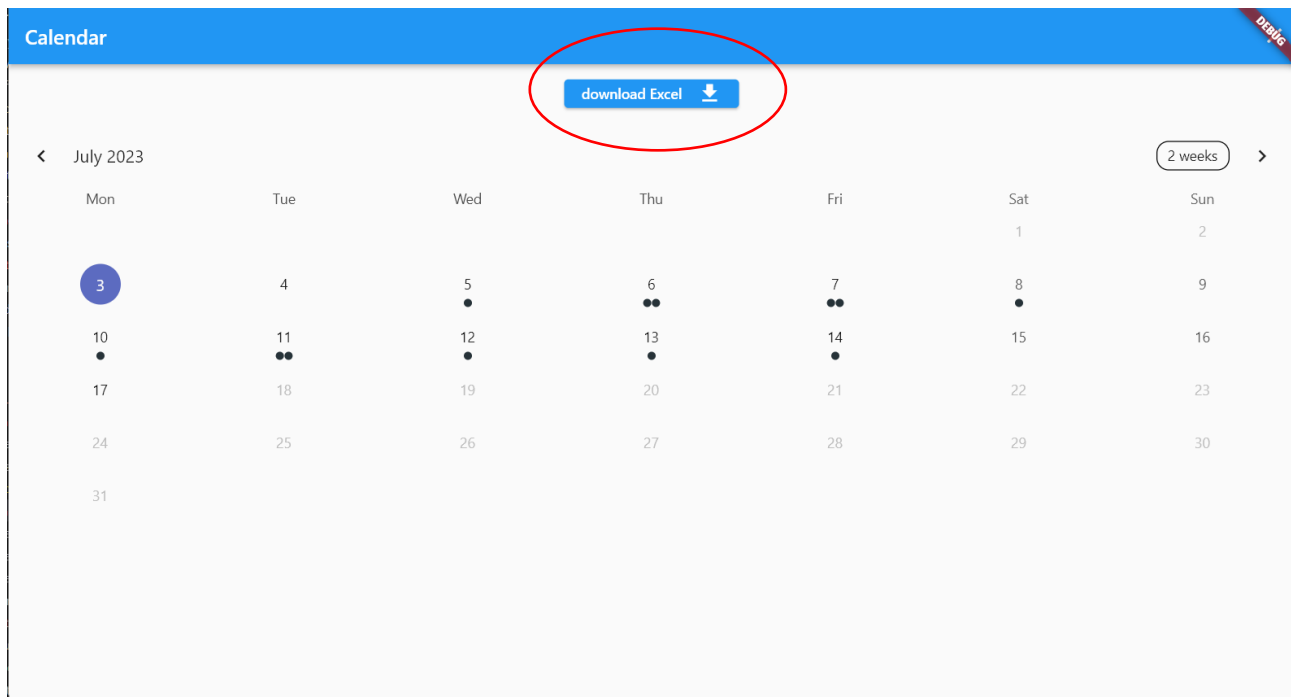
School is not defined

Optimization status checking

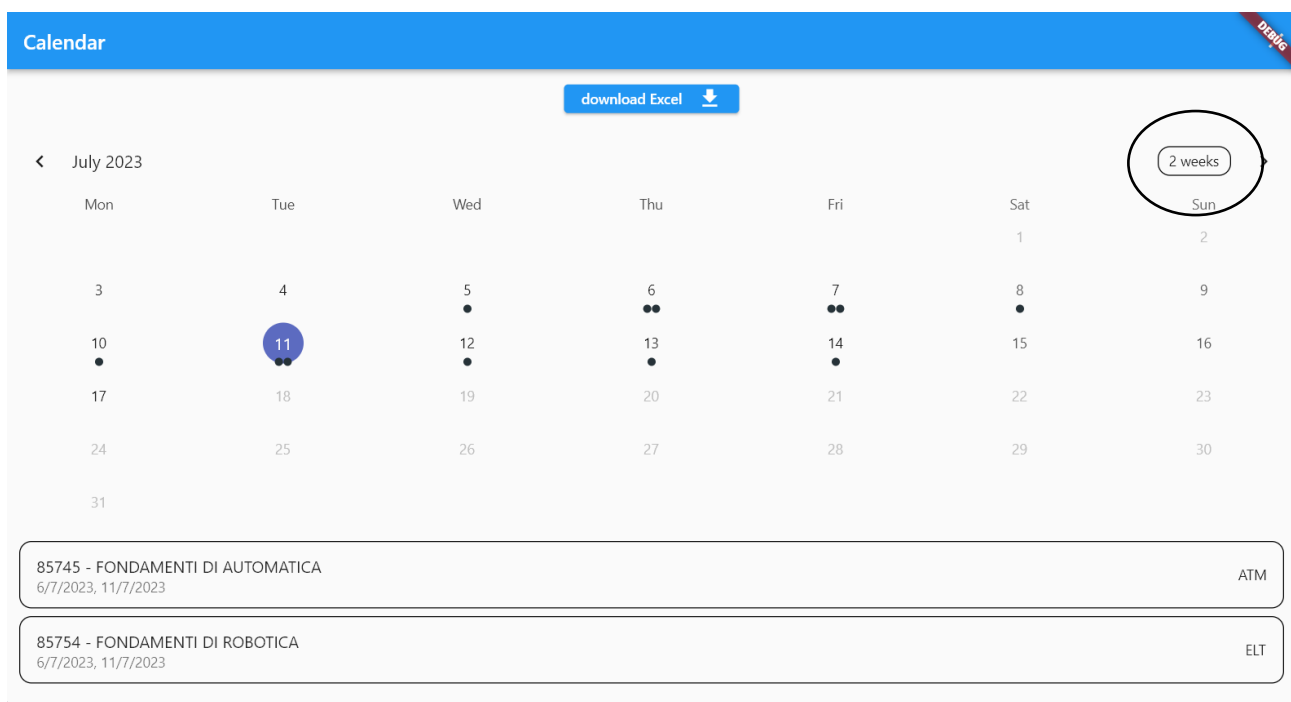
To check the status of the process, in the main page, the user can press on the dedicated button.

Calendar Download

When the optimal scheduling process is finished, the user will notice a green icon near to the problem session in the 'Select Page', the user can click in the problem session and as previously explained it will appear the calendar page. Here there is the possibility to download the excel file of the calendar scheduled clicking on the dedicated button.



If the user moves the mouse over a day with an exam, this will appear.



And clicking on the button in the black circle the showing settings change (month, week, two weeks).

Notes

These are all the action that a user can do, it is important to say that a user can work on a session even if the scheduling process of another one is in progress, but he cannot modify or delete the session in progress. Lastly is important that the database that contain the exam is well formatted. Here there is a description of how it should be.

The field must be the following in this order: School, Course Code, M/E, Course Name, Semester, Year, SEM, Location, Exam Head, Professor, Section, Enrolled number, CFU, Passed %, Average Mark.

Important remark the fields 'Professor' and 'Section' can contains more elements to associate the professor with its section the order must be the same. Moreover to separate professors and sections they must be separated with '-'. To understand the value of the other fields the user can refer to the Design Document, in particular to the 'Database' paragraph of the 'Component Breakdown' chapter.

4 - INSTALLATION GUIDE

To run the software in debug mode it is important to install different libraries or packages, they are:

For the back-end

- Python to run the backend
- Mip (Mixed Integer Programming) library to create and eventually solve the optimization problem
- GUROBI, to solve the optimization problem, it is not necessary because we can use only the mip solver but GUROBI provide better performances.
- Flask, it is used to create the server for the backend.
- firebase_admin, to communicate with the firebase database.
- pandas and openpyxl that in our case are used to work with the Database Exam.
- Other libraries as json or os are built-in so they don't need to be installed separately.

For the front-end

- All the libraries and dependendecies are described in the standard pubspec.yaml files:
- dependencies:
 - file_saver: any
 - provider: ^6.0.0
 - flutter_datetime_picker: ^1.5.1
 - http: ^1.1.0
 - go_router: any
 - table_calendar: ^3.0.9
 - file_picker: ^5.3.2

Once everything is installed it is possible to run the back-end on the the server pc and the front-end on the client pc changing the configuration parameter depending on the ip address of the server.

To test the software not in the debug mode, it is created also the .exe file for the back-end and the build for the webapp hosted on the free hosting platform Netlify. In this case it is sufficient to execute the back-end and access to the webapp with a browser.

In both cases for simplicity the tests are done on the same pc that runs the webapp and the server.