National College of Ireland

Computing

4th Year 2023/24

Ryan Lanz

X21759971

X21759971@student.ncirl.ie

Organizer

Technical Report

Contents

Executive Summary

The project aims to address the organizational challenges faced by students in managing their coursework efficiently. The envisioned application consolidates project management, assignment tracking, and event scheduling into a centralized platform, eliminating the need for students to navigate multiple sources. Key functionalities include secure login, module management, date scheduling, user feedback, and file handling. The tech stack involves Ruby on Rails for design, Bootstrap for frontend, and SQLite for the backend database, ensuring a seamless integration of various features.

The structural breakdown uses functional and non-functional requirements, design and architecture, implementation, and testing. Comprehensive use case diagrams and detailed requirements show the user interactions, emphasizing user-friendly experiences.

Initial evaluations have shown success in meeting functional requirements, including registration, login, module management, and scheduling. Identified issues, such as users adding modules and events with empty fields, are acknowledged for future refinement. Non-functional requirements, encompassing reliability, usability, security, and performance, exhibit positive results that are positive.

Furthermore, the project is poised for further development. Planned features include file upload/download functionality for enhanced storage accessibility, a contact/feedback page, and a UI overhaul to refine the user experience. The acknowledgment of the project's early stage and ongoing testing instils confidence in the development process. The strategic sequencing of future developments aligns with the project's overarching goal of providing students with a comprehensive and user-friendly platform.

In conclusion, the project id in its infancy but it showing promise with how it has begun and will continue to grow to be how it has been envisioned.

1.0 Introduction

1.1. Background

The problem looking to be solved with this project is the creation of an application that can help students with management of projects and assignments and help organise and schedule key dates and other events in relation to college course. The goal is to have a centralized application of all their needs in one single place without the need to go to multiple different sources, websites, or applications. As students have a wide range of tasks, different languages being learned with strict timeframes as well, the organization of all these things it can be difficult to stay on top of everything and quite easy for things to get out of hand and leave them falling behind.

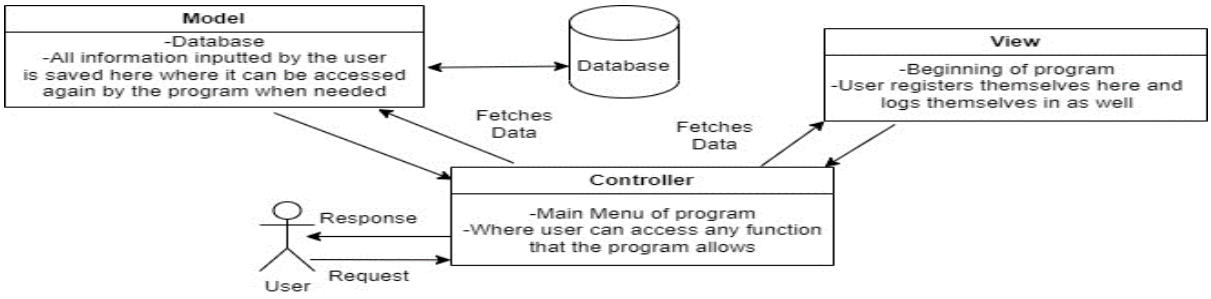
1.2. Aims

The main goals for the application would be to alleviate some of the stress of that and have the information in a single platform. With that in mind some of the functionalities the application should have include:

- Secure Login: A secure login that allows them to have their information and course information, assignments, due dates, exam dates relevant to them specifically.
- Management: The ability to add key dates that are in relation to their course, this helps to keep on top of everything when the important dates are known.

- Modules Access: Allows for adding, viewing, updating, and removing modules that the user can add course materials to be accessed with relevance to that module.
- Feedback: Send feedback to admins in relation to the application or send emails to a lecturer/HR/events team. Have the ability to send something without needing to leave the application.
- File Handling: Upload and Download files from a database that are related to your modules, this could be uploading all your course content so you have it all in one place and can download it for ease of access.

1.3. Technology



The project utilizes a wide range of technology in it's frontend and backend working asynchronously. The core technologies and components of the *backend* are:

Ruby on Rails: Rails is used here with a MVC (Model, View, Controller) framework which helps to streamline the backend development. Rails allows rapid development as it builds a lot of the code for you, hence removing a lot of the headache of backend coding is eliminated.

Database: The database here uses 2 separate databases, SQLite and PostgreSQL. SQLite is the default language for Rails and is used in the development environment. The projects deployment into production uses PostgreSQL as SQLite is not supported.

Gems & Libraries:

- Devise: This is used for user login and authentication. It adds secure login and registration functionalities to the program.
- SortableJS: Javascript library for drag and drop feature and sorting within separate lists and other containers. Used for moving tasks and events throughout the programs list and calendar features.
- Ranked Model: Ranked model allows for adding a rank and id to the tasks in the program from when they're created which automatically updates when they are moved.
- RequestJS: Javscript library for handling HTTP requests and responses. Makes AJAX requests and makes logs, modifies requests/responses as well as handles errors.

Frontend:

HTML/CSS & ERB: Embedded ruby renders the views in HTML to generate the frontend. Works alongside the backend to utilise the backend gems and code to make the program usable.

Bootstrap & Tailwind: Gems that help to create the UI and build components easily and quickly with a professional look.

1.4. Structure

The report follows this section with a breakdown of the system in different sections. This begins with the requirements of the project. This includes Functional, Non-Functional, and Data Requirements. The functional requirements also get further broken down with important use cases for the flow of the program alongside exceptions.

After that is the Design & Architecture, this gives an insight into the technologies used in order to build up the project and what exactly goes into it as well as how they function. It breaks down how they function in relation to the project and why they have been selected for the task. Afterwards is the implementation of them, followed by a guide of the interface showing the earliest prototype.

Furthermore, some rigid testing is done on the different variables and aspects of the program to make sure that they are functioning correctly and then these tests are evaluated to make sure they line up with the envisioned requirements. with the envisioned requirements.

2.0 System

2.1. Requirements

The program should be constructed in accordance with EU GDPR which is crucial in the construction of a program to avoid issues in the future. As data is stored on the user there are standards that need to be included.

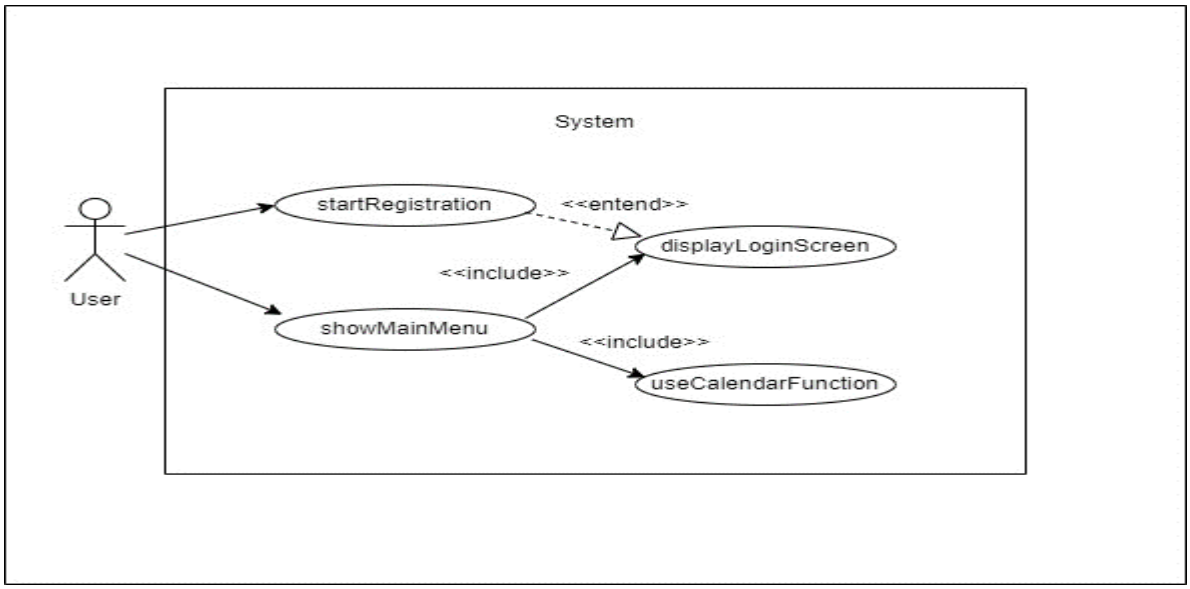
- User Data: Collection and storage of the user's personal information (name, email, username, password) needs to comply with GDPR by allowing for the user to have access to the data as well as asking consent for getting the users data. Information provided must also be securely saved where it cannot be accessed by personnel without authorization. This data stored will be retained only for how long it is necessary and if the user requests deletion it must be done.
- Event, Module, File Data: Data must be stored safely and securely belonging to the specific user, be properly labelled, and organized so as to be available to user upon request. This information will be detained for a certain period, stored securely, and must be deleted upon request by the user themselves.
- Database: All information stored is compliant with EU GDPR which includes data minimization and storage limitations which have clear retention policies that were explained above.

With these strict policies in place, they will adhere to current EU GDPR rules and be amended to comply with the rules at any change in rules, set by the GDPR committee in order to stay compliant.

2.1.1.Functional Requirements

FR-Number	Functional Requirement Title	Functional Requirement Description
FR-1	Modules	Module Management which should allow the user to manage their course modules. This includes adding, viewing, updating, and deleting said modules. Buttons should be provided for each of the corresponding actions and need to function as intended. A display must be in place to show the modules and information included about them including a name, ID, and lecturer.
FR-2	Scheduling	Date Management is required where users can schedule events and have key dates input in their calendar. A monthly calendar need be included which is interactive and allows users to add and edit dates, and view said dates when checking the calendar. List and task management must also be included. Users must be able to create, edit, update, and delete lists and tasks within those lists. Those tasks must also be movable within lists.
FR-3	Users	User Management needs to be included so that a user can amend their details that they input in the registration phase. They need to be advised that the update has been successful and can carry on without being logged out due to details being amended.

2.1.1.1 Use Case Diagram

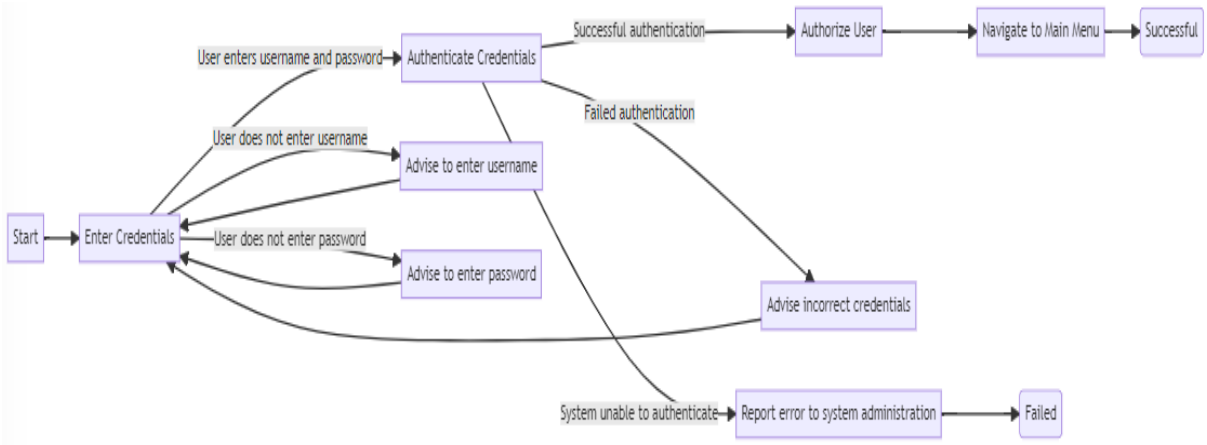


2.1.1.2 Requirement 1

Description

User Logging Into System

Use Case ID:	UC-1
Use Case Name:	displayLoginScreen
Created By:	Ryan Lanz
Date Created	24-06-23
Description:	User logging into system
Actors: Primary: Secondary:	User
Triggers:	User is at login screen
Pre-conditions:	User has launched the program. Program is connected to the database. Server is running. System is functioning correctly. User has already registered
Post-Conditions:	Successful: User is authorized and is logged in. Fail: User is advised to re-enter correct details and try again.
Normal Flow:	1) User enters username. 2) User enters password. 3) User presses login button. 4) System authenticates the credentials. 5) System authorizes user. 6) User is brought to main menu.
Alternate Flows:	1) User does not enter username. 1.1) System advises to enter a username. 2) User does not enter a password. 2.1) System advises to enter a password. 3) User enters incorrect credentials. 3.1) System advises details are incorrect and to try again. 3.2) User is returned to step 1.
Exceptions:	1.1 System advises that it is unable to complete the authentication process. 1.2 System logs the error. 1.3 The system reports the problem to the system administration. 1.4 System returns to the main menu.



Here is an Activity diagram of the use case. It shows the flow of all activities and actions that the login process has. It covers both the normal flow as well as alternate flows and exceptions.

Normal flow: 1. (Start: Login initiated with user entering credentials) 2. (Enter Credentials: User enters username and password) 3. (Authentication Credentials: System authenticates the credentials with the database) 4. (Successful Authentication: User is successfully authenticated) 5. (Authorize User: System authorizes user to access the main menu) 6. (Navigate to Main Menu: User is brough to the main menu). 7. (Success: Login process complete).

Alternate flow: 2.1) User does not enter correct username, advised to enter correct username, and is returned to step 2.

2.2) User does not enter correct password, advised to enter correct password, and is returned to step 2.

2.3) User does not enter correct username and password, advised to enter correct credentials, and is returned to step 2.

Exceptional flow: 3.1) System has an issue while authenticating the credentials, reports to the system admin, login process fails.

2.1.1.3 Requirement 2

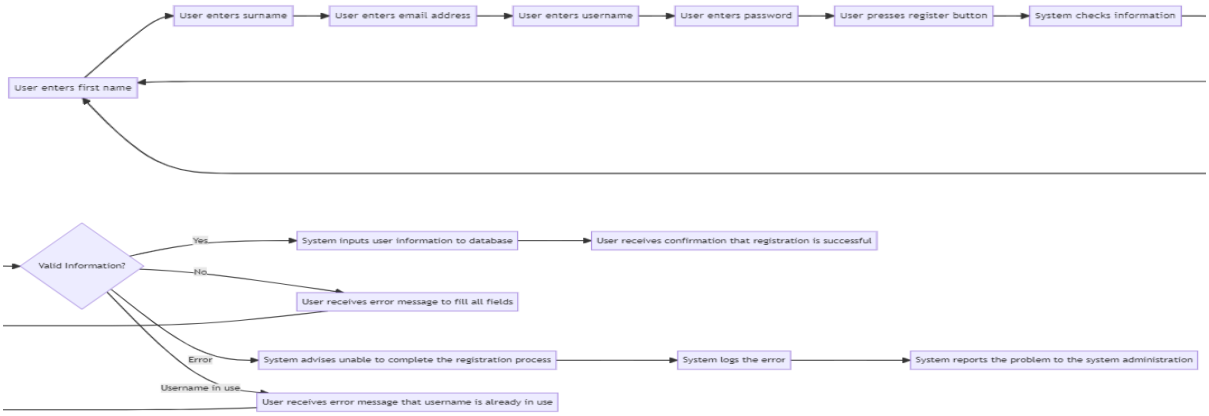
Description

User Registering On The System

Use Case ID:	UC-2
Use Case Name:	userRegister
Created By:	Ryan Lanz
Date Created	24-06-23
Description:	User registering on the system.
Actors: Primary: Secondary:	User
Triggers:	User enters details to register.
Pre-conditions:	User has launched the program and gone to register screen. The program is connected to the database. The system is functioning correctly. Server is running. User is not registered.
Post-Conditions:	User is registered successfully. User registration failed.
Normal Flow:	1). User enters first name. 2) User enters surname. 3) User enters email address. 4) User enters username. 5) User enters password. 6) User presses register button. 7) System checks information. 8) System inputs user information to database. 9) User receives confirmation that registration is successful.

Alternate Flows:	2.1) User does not fill all fields. 1.2) User presses register button. 1.3) System checks information. 1.4) User receives error message to fill all fields. 1.5) User returns to step 1. 3.1) User fills all fields. 2.2) User presses register button. 2.3) System checks information. 2.4) User receives error message that username is already in use. 2.5) User returns to step 1.
Exceptions:	4.1. The system advises that it is unable to complete the registration process. 4.2. The system logs the error. 4.3. The system reports the problem to the system administration.

The below diagram is an Activity diagram for Use Case 2. It had to be split in 2 as it was too long and the text was not viewable.



Normal flow: 1) User enters first name. 2) User enters surname. 3) User enters email address. 4) User enters username. 5) User enters password. 6) User presses register button. 7) System checks information. 8) System evaluates information to check if it is valid. 9) System stores information in database. 10) User receives registration confirmation.

Alternate flow: 2.1) User does not enter all fields. 2.2) User presses register button. 2.3) System checks information. 2.4) User receives error fill all fields. 2.5) User returns to Step 1.

3.1) User fills all fields. 2.2) User presses register button. 2.3) System checks information. 2.4) User receives error message that username is already in use. 2.5) User returns to step 1.

Exceptional flow: 4.1. The system advises that it is unable to complete the registration process. 4.2. The system logs the error. 4.3. The system reports the problem to the system administration.

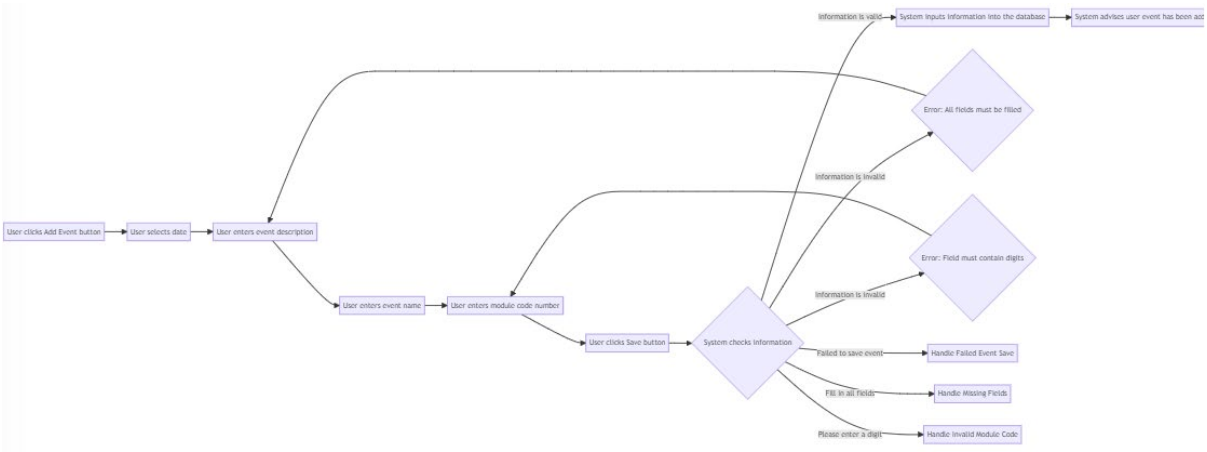
2.1.1.4 Requirement 3

Description

User Adding Event On Calendar

Description:	startCalendarFunction
Actors:	

Primary Secondary	User
Triggers:	User clicks Add button to initiate the add event function.
Pre-conditions:	User has logged in and has gone to the Calendar function. The program is connected to the database. The system is functioning correctly.
Post-Conditions:	Event has been added successfully. Event has not been added.
Normal Flow:	1) User clicks Add Event button. 2) User selects date. 3) User enters event description. 4) User enters event name. 5) User enters module code number. 6) User clicks Save button. 7) System checks information. 8) System inputs information into the database. 9) System advises user event has been added.
Alternate Flows:	3) User does not enter an event description 4) User enters event name. 5) User enters module code number. 6) User clicks Save button. 7) System checks information. 7.1) System advises user error, all fields must be filled. 7.2) User returns to step 3. 5) User enters letter for module code number. 6) User presses Save button. 7) System checks information. 7.1) System advises user error, field must contain digits. 7.2) User returns to step 5.
Exceptions:	1. Failed to save event. 2. Fill in all fields 3. Please enter a digit.



The activity diagram goes as follows:

Normal Flow: 1.) User clicks Add Event button: The user initiates the process of adding a new event to the calendar. 2) User selects date: The user selects a date for the new event. 3) User enters event description: The user enters a description for the event. 4)User enters event name: The user provides a name for the event. 5) User enters module code number: The user inputs a module code number for the event. 6) User clicks Save button: The user triggers the saving of the event. 7) System checks information: The system verifies the entered information. 8) System inputs information into the database: If the information is valid, the system stores the event details in the database. 9) System advises user event has been added: The system notifies the user that the event has been successfully added.

Alternate Flows: 1) User does not enter an event description: If the user skips entering the event description, the system prompts an error message indicating that all fields must be filled. The user is then directed back to the step of entering the event description. 2) User enters letter for module code number: If the user enters a non-digit character for the module code number, the system displays an error message stating that the field must contain digits. The user is then directed back to the step of entering the module code number.

Exceptions: 1) Failed to save event: If the system encounters an issue while saving the event, an error handler is triggered to handle the failed event save. 2) Fill in all fields: If the user fails to fill in all the required fields, an error handler is triggered to handle the missing fields.3) Please enter a digit: If the user enters a non-digit character for the module code number, an error handler is triggered to handle the invalid module code.

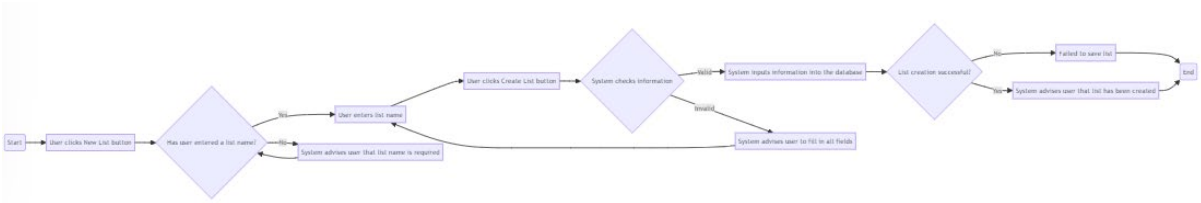
2.1.1.5 Requirement 4

Description

User creating a new list

Description:	createListFunction
Actors: Primary Secondary	User
Triggers:	User clicks New List button to initiate the list creation process
Pre-conditions:	User has logged in and has gone to the Lists section The program is connected to the database. The system is functioning correctly.
Post-Conditions:	List has been created successfully. List has not been added.
Normal Flow:	1) User clicks New List button. 2) User enters list name. 5) User enters module code number. 6) User clicks Create List button. 7) System checks information. 8) System inputs information into the database. 9) System advises user list has been created.
Alternate Flows:	2) User doesn't enter a list name. 2.1) System advises user that list name is required. 2.2) User returns to step 2.
Exceptions:	1. Failed to save list. 2. Fill in all fields

The activity diagram goes as follows:



Normal Flow: 1) User clicks New List button. 2) User enters list name. 5) User enters module code number. 6) User clicks Create List button. 7) System checks information. 8) System inputs information into the database. 9) System advises user list has been created.

Alternate Flows: 2) User doesn't enter a list name. 2.1) System advises user that list name is required. 2.2) User returns to step 2.

Exceptions: 1. Failed to save list. 2. Fill in all fields

2.1.1.6 Requirement 5

Description

User creating a new task

Description:	createListFunction
Actors: Primary Secondary	User
Triggers:	User clicks New Task button to initiate the task addition process.
Pre-conditions:	User has logged in and has gone to the Lists section. User has created the list to which the task is for The program is connected to the database. The system is functioning correctly.
Post-Conditions:	Task has been created successfully. Task has not been created.
Normal Flow:	1) User clicks New Tasj button. 2) User enters list name. 3) Enters task name. 4) User clicks Create List button. 5) System checks information. 6) System inputs information into the database. 7) System advises user task has been created.
Alternate Flows:	2) User doesn't enter a correct list name. 2.1) System advises user that list name is required. 2.2) User returns to step 2. 2) User doesn't enter task name. 2.1) System advises user that task name is required. 2.2) User returns to step 2.
Exceptions:	1. Failed to save Task. 2. Fill in all fields

The activity diagram goes as follows:

Normal Flow: 1) User clicks New Tasj button. 2) User enters list name. 3) Enters task name. 4) User clicks Create List button. 5) System checks information. 6) System inputs information into the database. 7) System advises user task has been created.

Alternate Flows: 2) User doesn't enter a correct list name. 2.1) System advises user that list name is required. 2.2) User returns to step 2.
2) User doesn't enter task name. 2.1) System advises user that task name is required. 2.2) User returns to step 2.

Exceptions: 1. Failed to save task. 2. Fill in all fields

2.1.1 Data Requirements

The program should be constructed in accordance with EU GDPR which is crucial in the construction of a program to avoid issues in the future. As data is stored on the user there are standards that need to be included.

- User Data: Collection and storage of the user's personal information (name, email, username, password) needs to comply with GDPR by allowing for the user to have access to the data as well as asking consent for getting the users data. Information provided must also be securely saved where it cannot be accessed by personnel without authorization. This data stored will be retained only for how long it is necessary and if the user requests deletion it must be done.
- Event, Lists, Tasks, Module, File Data: Data must be stored safely and securely belonging to the specific user, be properly labelled, and organized so as to be available to user upon request. This information will be detained for a certain period, stored securely, and must be deleted upon request by the user themselves.
- Database: All information stored is compliant with EU GDPR which includes data minimization and storage limitations which have clear retention policies that were explained above.

With these strict policies in place, they will adhere to current EU GDPR rules and be amended to comply with the rules at any change in rules, set by the GDPR committee in order to stay compliant.

2.1.2 Non-Functional Requirements

The Non-Functional Requirements define exactly how the system works behind the scenes. This is important for the user experience; the functional requirements are for the functions within the program but the non-functional requirements how those features as well as the program as a whole actually functions. These 4 requirements seem most valuable to me which

is why they are chosen and the criteria was selected in order to have the user experience be smooth and pleasant and work how it has been envisioned.

NFR-Number	Title	Non-Functional Requirement Description	Criteria
NFR-1	Reliability	Functions work as intended. Consistency is essential. Must always have a high level of uptime and not have any crashes. Exceptions can occur from user misusing a function, but they must be caught, and the system cannot incur fatal errors.	**Uptime:** 99.9% **Exceptions:** All Caught
NFR -2	Usability	U.I. needs to be user-friendly, easily accessible, and easy to navigate. Buttons and labels must be clear and input fields need clear instructions. Users should not spend more than >5s trying to find any given option. Should not select incorrect option more than once.	**Navigation Time:** <5s **Incorrect Options Selected:** <2 times
NFR -3	Security	Data input must be secure. Database connection needs to be secure, and data should be shielded from SQL injection. Sensitive user information needs to be encrypted. Authentication and authorization need to be implemented for login. If the incorrect information is input, then the login attempt fails, and user cannot gain access to the program.	**Encryption:** Password, SHA-256 **Database Authentication:** Values 100% correct
NFR -4	Performance	System must load almost instantly in all aspects. All functions should execute quickly when the user presses the associated button. Exception messages need to occur straight away. No loading or buffering at any stage of using the system.	**Login:** <3 **Register:** <3 **Add Functions (Calendar dates, modules):** <3 **Error Messages:** <1

2.2 Design & Architecture

2.2.1 Design Overview

The system follows the Rails framework using Ruby and Bootstrap for backend and frontend. Bootstrap in the frontend uses a combination of HTML, CSS, and JavaScript. It has plenty of pre-wrote scripts that you can add to your CSS page to and add sleek designs to your pages and components quickly. It utilizes gems for added features. As the project is in its early stages, a lot of the groundwork has been done using scaffolds. A scaffold in Ruby on Rails builds and programs basic components for

your program in a matter of seconds, combine this with the using scaffolds for the gems, if you had to code it for yourself depending on your experience it could take hours or days even.

2.2.2 System Architecture

Backend Framework:

- Ruby on Rails: This is the backbone of the application and follows a MVC (Model-View-Controller) design pattern. The three different components of it work in different ways to the development of the program.
    - The model consists of the different classes of Ruby coding, which interact with the database. Essentially, it is responsible for the CRUD database in my instance. If a user created a web page but with no backend like a model, they would be inputting data and it would simply do nothing.
    - The view is the how the data from the database and such is shown to the user. It uses templates to generate HTML with '.erb' (embedded Ruby code). If the model creates the data in the CRUD database, the view is used to make the pages with the means to input data into the data like a form.
    - The controller works alongside the model to take in the user input and request and 'control' the actions. The view presents a way to input the data, the controller takes the input from the view, and the model then communicates what the controllers' specific requests is to the database to update it.

Frontend Framework:

- Rails Views: Although views are used in the backend it would also count as frontend as it generates HTML and works alongside the other frontend technologies and languages.
- Bootstrap: This is a framework that uses HTML, CSS, and JavaScript in an easy to use and seamless way. Without the framework, you would need to code everything separately and integrate it to all work synchronously whereas with Bootstrap, you simply find what the design is you are looking for and then can add a line of code to your program from the Bootstrap website. And it is done that simply, you still have to tweak it to your needs but the base it there.

Database Design:

- SQLite: Ruby on Rails can use different databases as far as I knew but I used SQLite as it was set up with mine and works perfectly for any of my needs. With previous experience in MySQL, SQLite database and schema structure are suitable in all ways. The program uses 2 files for saving data. A 'db' or database file, and a 'schema.rb' or schema file. These need to be migrated and updated throughout the program in order for it to run correctly.

2.3 Implementation

Now that that the system architecture has been further broken down and the different ways the project works, lets go through the implementation of it and the key functions that define the program.

Backend Framework:

- *Model:* Below you can see the model classes for the modules and users. These are the only correctly and fully functioning sections of the projects so far and there is some important code to speak about.

Collegemodule.rb & Event.rb:

```ruby
class Collegemodule < ApplicationRecord
    #Adds link to signed in user
    belongs_to :user
end
```

```ruby
class Event < ApplicationRecord
    #Adds link to signed in user
    belongs_to :user
end
```

belongs_to :user makes them specific to signed in user so it's not viewable by everyone.

```ruby
class List < ApplicationRecord
    #Connect lists to tasks
    validates :name, presence: true
    has_many :tasks, dependent: :destroy
    belongs_to :user

    #Add ranked model
    include RankedModel
    ranks :row_order
end
```

List.rb:
has_many :tasks, dependent: :destroy creates a one-to-many relationship so that a list can have multiple tasks associated with it and that when a list is deleted, tasks are deleted with it.

ranks :row_order, with_same: :list_id implements the ranked-model gem to store the position of a list amongst other lists.

```ruby
class Task < ApplicationRecord
  validates :name, presence: true
  belongs_to :list
  belongs_to :user


    #Add ranked model but have rank within list
    include RankedModel
    ranks :row_order, with_same: :list_id
end
```

Task.rb:

Similar to list but *belongs_to :list* allocates the task to a specific list and user and *ranks :row_order, with_same :list_id* puts ranking order on tasks that are allocated to a list.

- *Controller:*

```ruby
class ApplicationController < ActionController::Base
    before_action :authenticate_user!
end
```

application_controller.rb:

before_action :authenticate_user! Ensures the user is authenticated before being able to access the program.

```ruby
class CalendarController < ApplicationController
  def month
    @date = Date.parse(params.fetch(:date, Date.today.to_s))
    @tasks = Task.where("start_date >= ? AND start_date <= ?", @date.beginning_of_month, @date.end_of_month)
    @events = Event.where(start_time: @date.all_month).group_by { |e| e.start_time.to_date }
    @lists = List.includes(:tasks).where(user: current_user) # Fetches list for current user only
  end
end
```

calendar_controller.rb

@date = Date.parse(params.fetch(:date, Date.today.to_s)) parses date from params from request and defualts to current date when no specified one.

@tasks = Task.where("start_date >= ? AND start_date <= ?", @date.beginning_of_month, @date.end_of_month) retrieves tasks in month in question aswell as stores tasks to be displayed.

@events = Event.where(start_time: @date.all_month).group_by { |e| e.start_time.to_date } retrrieves events from specified month and displays them in calendar.

@lists = List.includes(:tasks).where(user: current_user) fetches lists for the logged in user.

```ruby
# GET /lists or /lists.json
def index
  @lists = current_user.lists.rank(:row_order)
end

def sort
  @list = current_user.lists.find(params[:id])
  @list.update(row_order_position: params[:row_order_position])
  head :no_content
end

# GET /lists/1 or /lists/1.json
def show
  @list = current_user.lists.find(params[:id])
end

# GET /lists/new
def new
  @list = current_user.lists.build
end

# GET /lists/1/edit
def edit
  @list = current_user.lists.find(params[:id])
end
```

```ruby
# POST /lists or /lists.json
def create
  @list = current_user.lists.build(list_params)
  respond_to do |format|
    if @list.save
      format.html { redirect_to list_url(@list), notice: "List was successfully created." }
      format.json { render :show, status: :created, location: @list }
    else
      format.html { render :new, status: :unprocessable_entity }
      format.json { render json: @list.errors, status: :unprocessable_entity }
    end
  end
end

# PATCH/PUT /lists/1 or /lists/1.json
def update
  respond_to do |format|
    if @list.update(list_params)
      format.html { redirect_to list_url(@list), notice: "List was successfully updated." }
      format.json { render :show, status: :ok, location: @list }
    else
      format.html { render :edit, status: :unprocessable_entity }
      format.json { render json: @list.errors, status: :unprocessable_entity }
    end
  end
end
```

events_controller.rb, lists_controller.rb & tasks_controller.rb all contain similar methods with some exceptions. *Def index, def sort, def show* handles ranking, sorting, and displaying items. While *def new, def edit, def create, def update* contain the CRUD based actions.

```javascript
import { Controller } from "@hotwired/stimulus"
import Sortable from "sortablejs"

export default class extends Controller {
  connect() {
    this.initSortable()
  }

  initSortable() {
    new Sortable(this.element, {
      group: {
        name: this.element.dataset.group || 'default',
        pull: true,
        put: (to) => {
          // Ensure the task can only be dropped within another task list
          return to.el.classList.contains('tasks')
        }
      },
      animation: 150,
      onEnd: (event) => {
        let url = this.element.dataset.sortableUrl
        let csrfToken = document.querySelector("[name='csrf-token']").content
```

```
        fetch(url, {
          method: 'PATCH',
          headers: {
            'Content-Type': 'application/json',
            'X-CSRF-Token': csrfToken
          },
          body: JSON.stringify({
            id: event.item.dataset.id,
            position: event.newIndex,
            list_id: event.to.closest('.tasks').dataset.listId,
            date: event.to.dataset.date
          })
        })
        .then(response => response.json())
        .then(data => {
          console.log('Updated successfully:', data)
        })
        .catch(error => console.error('Error updating:', error))
      }
    })
  }
}
```

Sortable_controller.js:

Group configures drag-and-drop features between the groups alongside *pull* and *put*. *Put* rstricts the function to only allow dropping elements in the task class. Once the item is then dropped, *onEnd* is then executed to update the items rank behind the scenes.

```
<%= link_to "<", calendar_month_path(date: @date - 1.month) %>
<%= link_to "Today", calendar_month_path %>
<%= link_to ">", calendar_month_path(date: @date + 1.month) %>
<%= @date.strftime('%B %Y') %>
<%= render 'calendar/month', date: @date, tasks: @tasks, lists: @lists %>
```

month.html.erb:

Contains mostly navigational liknks and renders partials. *@date.strftime('%B %Y')* displays the current month and year.

```
<div class="container">
  <div class="row">
    <!-- Lists Column -->
    <div class="col-md-4 lists">
      <h2>Lists</h2>
      <div id="lists" class="sortable">
        <% if @lists.present? %>
          <% @lists.each do |list| %>
            <div class="list sticky-note" id="list-<%= list.id %>" data-list-id="<%= list.id %>">
              <h3><%= list.name %></h3>
              <div class="tasks sortable" id="tasks-list-<%= list.id %>" data-list-id="<%= list.id %>">
                <% list.tasks.each do |task| %>
                  <div class="task" id="task-<%= task.id %>" data-task-id="<%= task.id %>">
                    <%= task.name %>
                  </div>
                <% end %>
              </div>
            </div>
          <% end %>
        <% else %>
          <p>No lists found.</p>
        <% end %>
      </div>
```

_month.html.erb:

This is the lists column for displayed the list for the user, *@lists* renders the list as a sticky note with the tasks inside.

The calendar column builds the calendar from scratch, there is no gem invovled. This was done for flexibility to integrate with the lists and tasks as it proved difficult implementing it with a gem like Simple Calendar.

grid-cols-7 uses a 7 day shaped grid system for the calendar.

Date::ABBR_DAYNAMES.rotate rotates day name array from Monday onwards.

```
<% @date.all_month.each do |day| %>
  <div class="border min-h-24 <%= "bg-blue-200" if day == Date.today %>">
    <div class="date"><%= day.strftime('%d') %></div>
```

This helps to create a separate ‘cell’ for each day so they can be easily distinguished and highlights the current day.

```
<% @date.all_month.each do |day| %>
  <div class="border min-h-24 <%= "bg-blue-200" if day == Date.today %>">
    <div class="date"><%= day.strftime('%d') %></div>
    <div id="day-<%= day %>" class="task-dropzone" data-date="<%= day %>">
      <% @tasks.where("DATE(start_date) = ?", day).each do |task| %>
        <div class="task" id="task-<%= task.id %>" data-task-id="<%= task.id %>">
          <%= task.name %>
        </div>
```

This is a ‘dropzone’ which is for the tasks to be dropable in the cells and enables them to be movable throughout the dates and dropped and then displayed accordingly.

```
<script>
  document.addEventListener('DOMContentLoaded', function() {
    // Initialize Sortable for task lists
    const lists = document.querySelectorAll('.tasks');
```

'document.addEventListener('DOMContentLoaded', function() makes certain that the script runs after HTML has loaded to prevent errors from element manipluation.

'const lists = document.querySelectorAll('.tasks'); gets all tasks elements and initializes SortableJS.

```
group: {
  name: 'tasks',
  pull: true,
  put: true
},
animation: 150,
```

name: 'tasks' defines tasks as a group so they can be moved between lists.

```
onEnd: function(evt) {
  const taskId = evt.item.id.split('-')[1]; // Extract task ID
  const newListId = evt.to.closest('.tasks').id.split('-')[2]; // Extract list ID
  const newIndex = evt.newIndex; // Get new index
  console.log(`Moved task ${taskId} to list ${newListId} at position ${newIndex}`);
```

This section is utilised when the drag/drop feature is completed. It essentially extracts the elements attriubutes (like the ID), retrives the ID of the list it's being put onto and then gets the new index of the tasks in the new list.

```
if (newListId) {
  fetch(`/tasks/${taskId}/sort`, {
    method: 'PATCH',
    headers: {
      'Content-Type': 'application/json',
      'X-CSRF-Token': document.querySelector('meta[name="csrf-token"]').content
    },
    body: JSON.stringify({ list_id: newListId, row_order_position: newIndex })
```

'/tasks/${taskID}/sort is the target endpoint to update order of tasks and *PATCH* updates the existing resources. The JSON object is used by the server to update the datbase with *list_id* and *row_order_position*.

```
            console.log( Moved task ${taskId} to date ${newDate} );

            fetch(`/tasks/${taskId}/update_date`, {
              method: 'PATCH',
              headers: {
                'Content-Type': 'application/json',
                'X-CSRF-Token': document.querySelector('meta[name="csrf-token"]').content
              },
              body: JSON.stringify({ date: newDate })
            })
            .then(response => {
              if (!response.ok) {
                console.error(`Failed to update task ${taskId} to date ${newDate}`);
                throw new Error('Network response was not ok.');
              }
              return response.json();
            })
            .then(data => console.log('Task date updated successfully:', data))
            .catch(error => console.error('Error:', error));
          }
        });
      });
    });
  </script>
```

Although there are some similarities between the above section of *_month.html.erb* and *sortable_controller.js* they both have their own functionalities and serve their own purposes. *sortable_controller.js* is a stimulus controller that handles different functionalities throughout the program, it initializes automatically to handle sortable elements throughout so not to be duplicated often. The *_month.html.erb* is a script embedded into the view file initalizing sortable elements on the particular page. There was major issues during creation and I had originally just used sortable on the Lists page but I came across problems when implementing it on the calendar and lists/tasks on the calendar page. This lead to it being initalized for lists and tasks on the calendar page aswell to get it working again.

Schema.rb: Below are the schemas and tables for the database of the project.

```
ActiveRecord::Schema[7.0].define(version: 2024_07_28_144821) do
  create_table "collegemodules", force: :cascade do |t|
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
    t.integer "user_id", null: false
    t.string "module_name"
    t.string "module_id"
    t.string "module_lecturer"
    t.index ["user_id"], name: "index_collegemodules_on_user_id"
  end

  create_table "events", force: :cascade do |t|
    t.string "title"
    t.text "description"
    t.datetime "start_time"
    t.datetime "end_time"
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
    t.integer "user_id", default: 1, null: false
    t.index ["user_id"], name: "index_events_on_user_id"
  end
```

```
create_table "lists", force: :cascade do |t|
  t.string "name"
  t.datetime "created_at", null: false
  t.datetime "updated_at", null: false
  t.integer "row_order"
  t.integer "user_id"
end

create_table "tasks", force: :cascade do |t|
  t.string "name"
  t.integer "list_id", null: false
  t.datetime "created_at", null: false
  t.datetime "updated_at", null: false
  t.integer "row_order"
  t.integer "user_id"
  t.date "start_date"
  t.index ["list_id"], name: "index_tasks_on_list_id"
  t.index ["user_id"], name: "index_tasks_on_user_id"
end
```

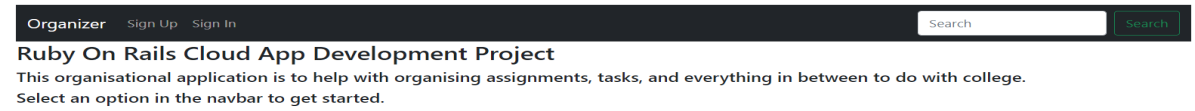
```
  create_table "users", force: :cascade do |t|
    t.string "email", default: "", null: false
    t.string "encrypted_password", default: "", null: false
    t.string "reset_password_token"
    t.datetime "reset_password_sent_at"
    t.datetime "remember_created_at"
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
    t.index ["email"], name: "index_users_on_email", unique: true
    t.index ["reset_password_token"], name: "index_users_on_reset_password_token", unique: true
  end

  add_foreign_key "collegemodules", "users"
  add_foreign_key "events", "users"
  add_foreign_key "tasks", "lists"
  add_foreign_key "tasks", "users"
end
```

2.4 Graphical User Interface (GUI)

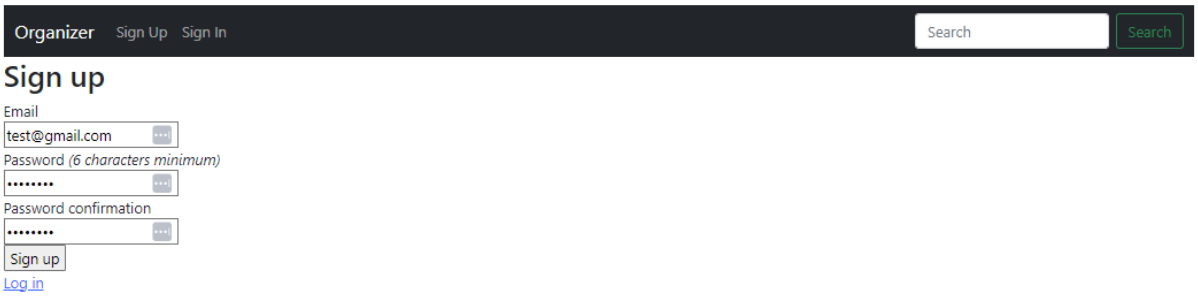
System Design:

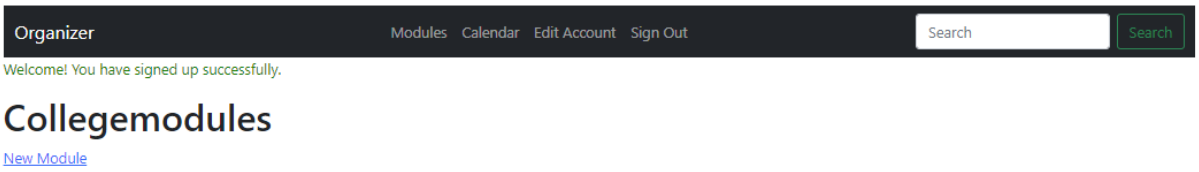
Below is a user manual for each page of the program. Each part contains a screenshot to help visualize and a brief explanation related to the corresponding screenshot. This helps to understand the program and understand why certain features were created the way they are as well as further breakdown the program.

Start Screen: Home page with options in the navbar to proceed.

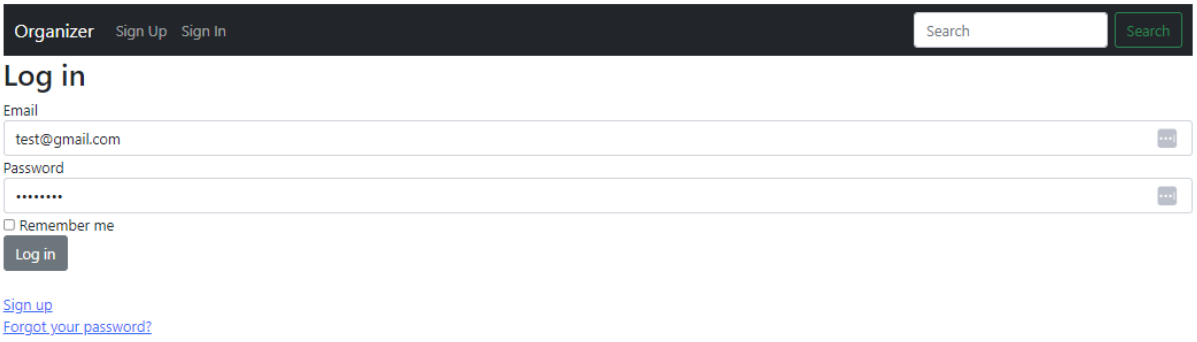


Registration: Sign up page for user to create an account, notifies them when successful or unsuccessful.

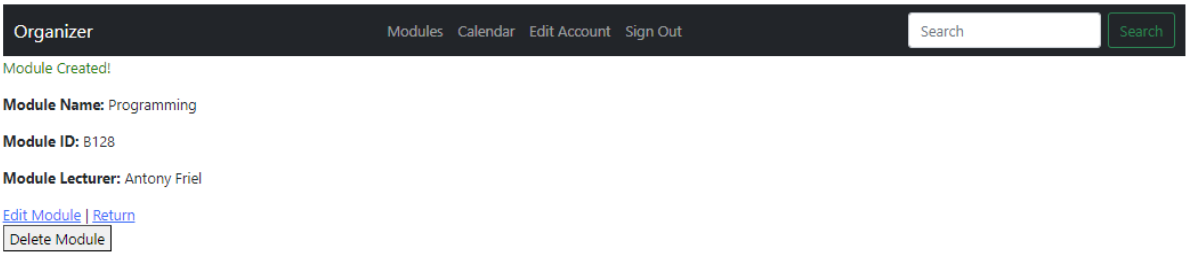


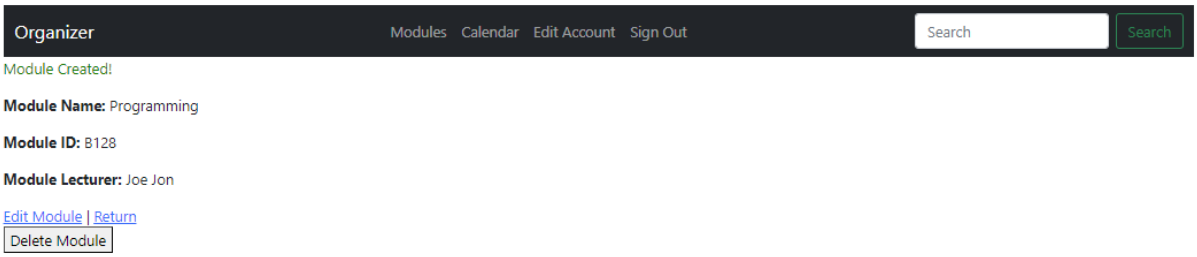


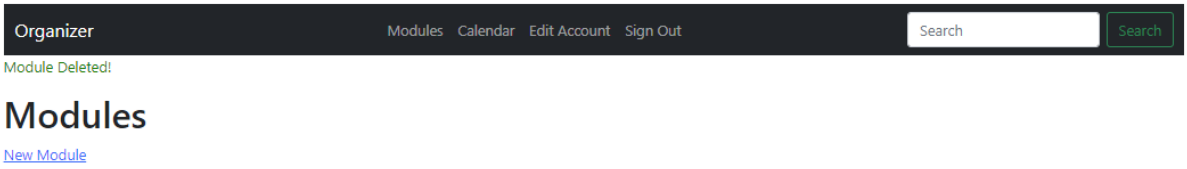
Login: Login page for users who have successfully created account, signed in automatically after registration.



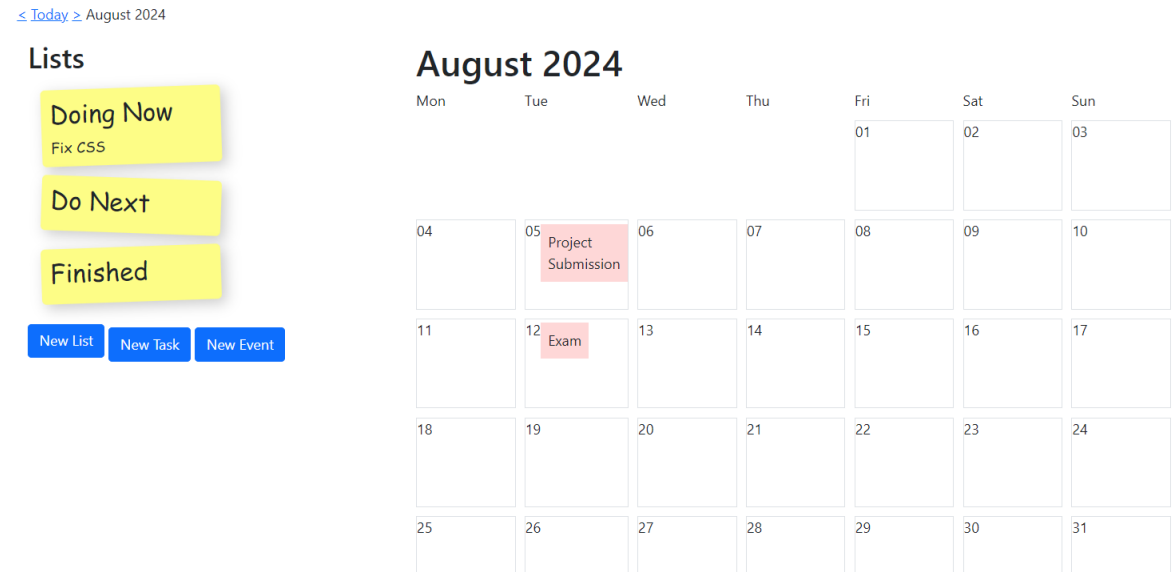
Modules: College modules page, uses the CRUD database for the data and is linked to the user.



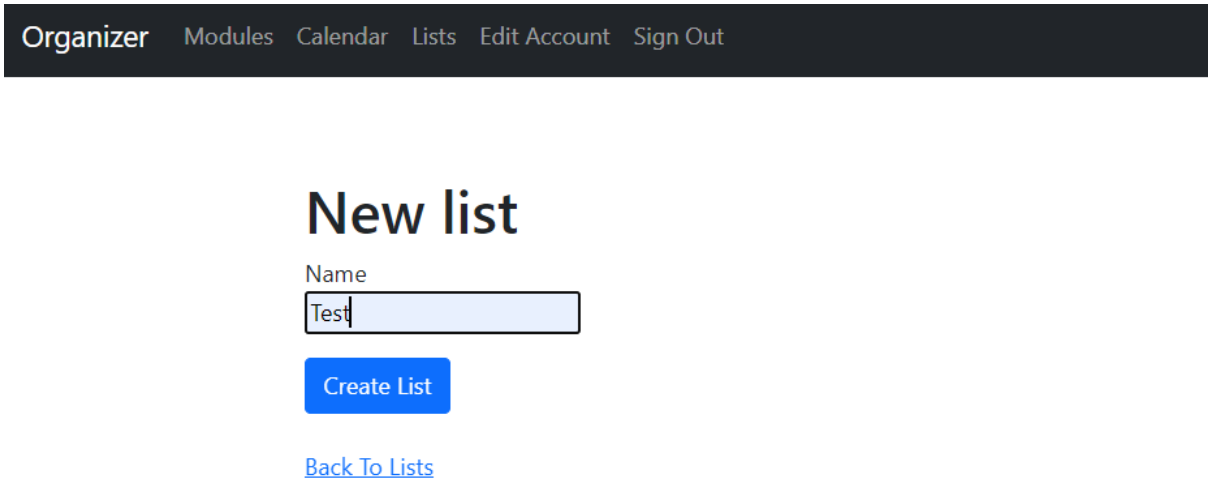




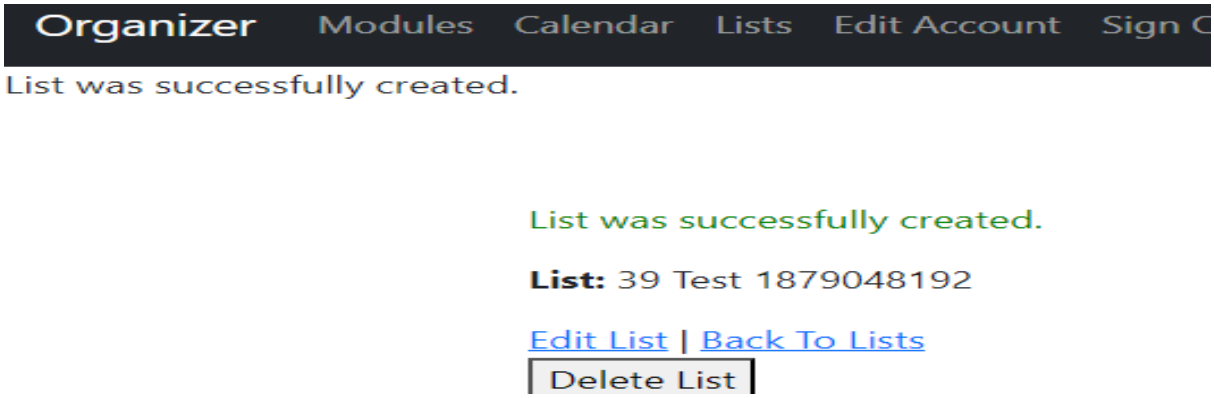
Calendar: Interactive calendar uses CRUD database as well. Contains Lists and Tasks which are movable through lists. Events are movable throughout the calendar aswell.



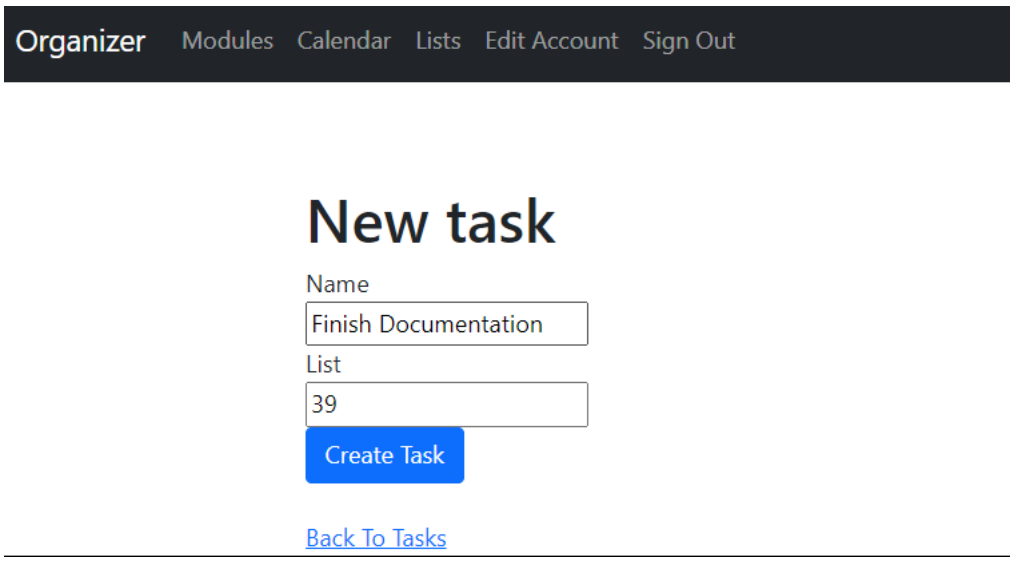
Create new list:



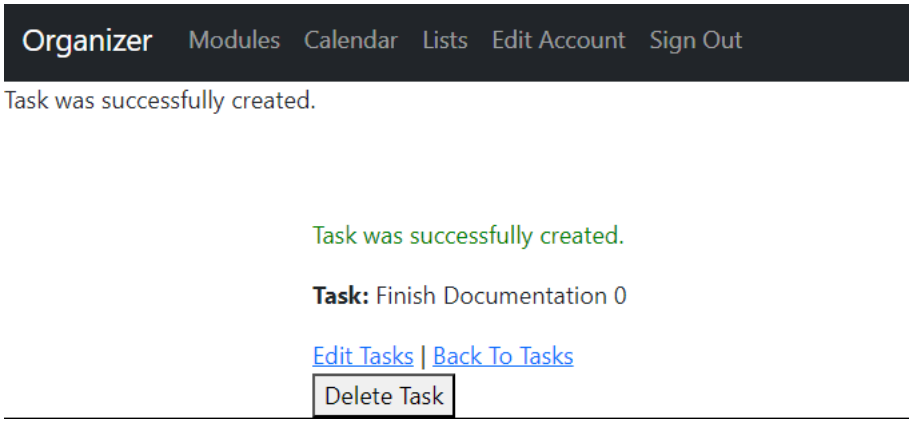
List created:



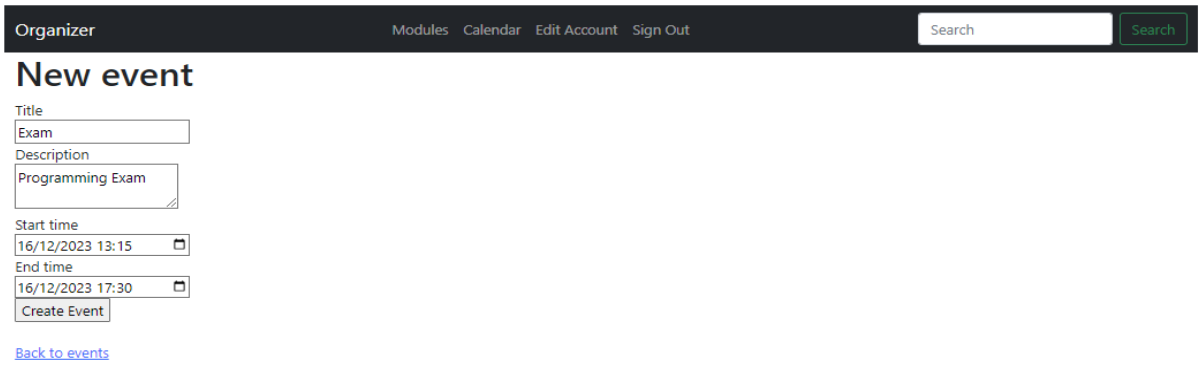
Create new task (list name has to be the list ID):



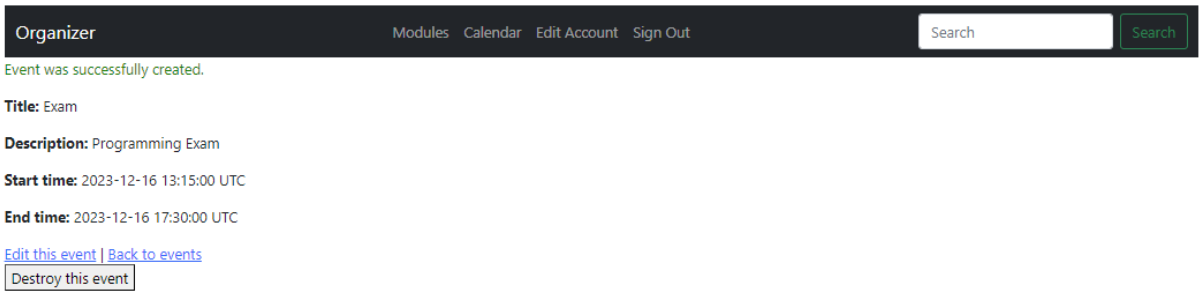
Task created:



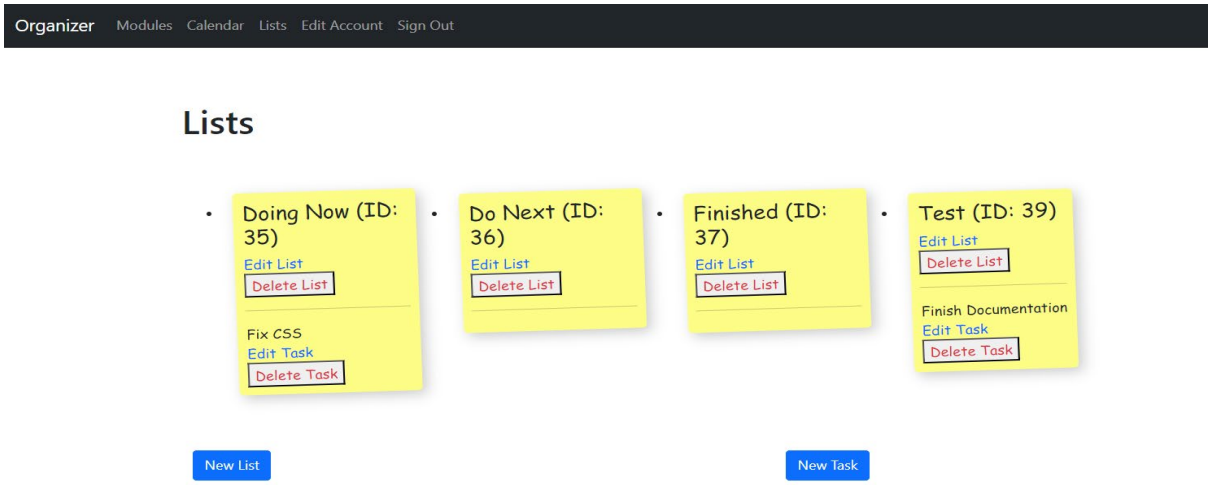
New event:



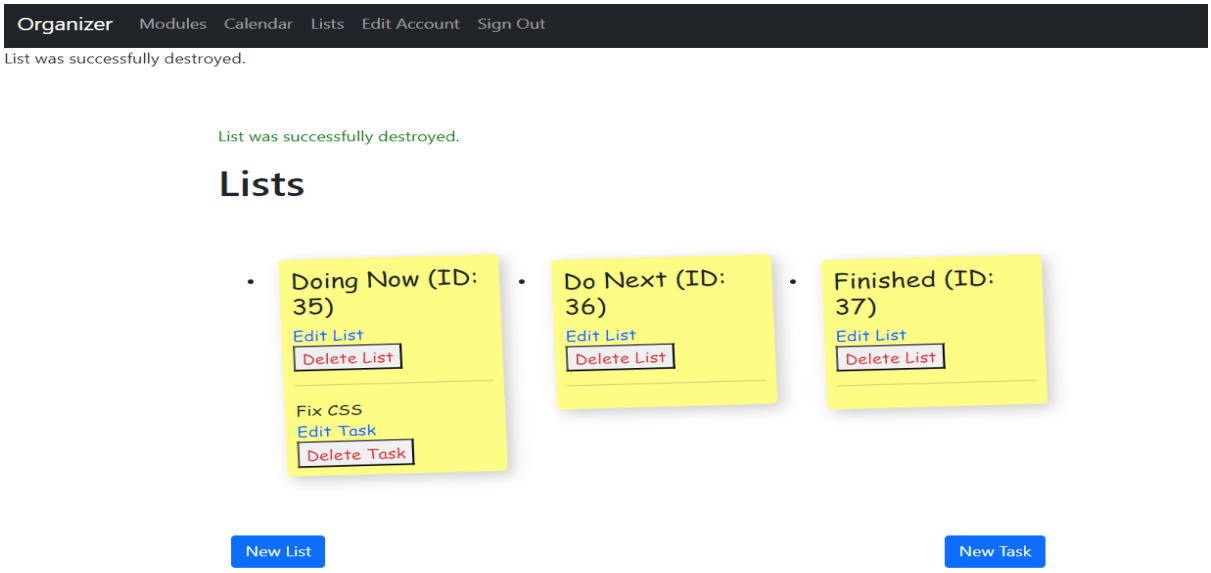
Event created:



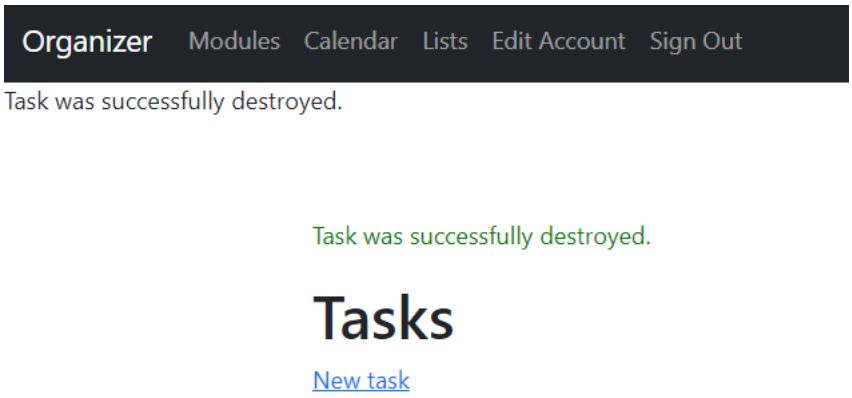
Lists page view:



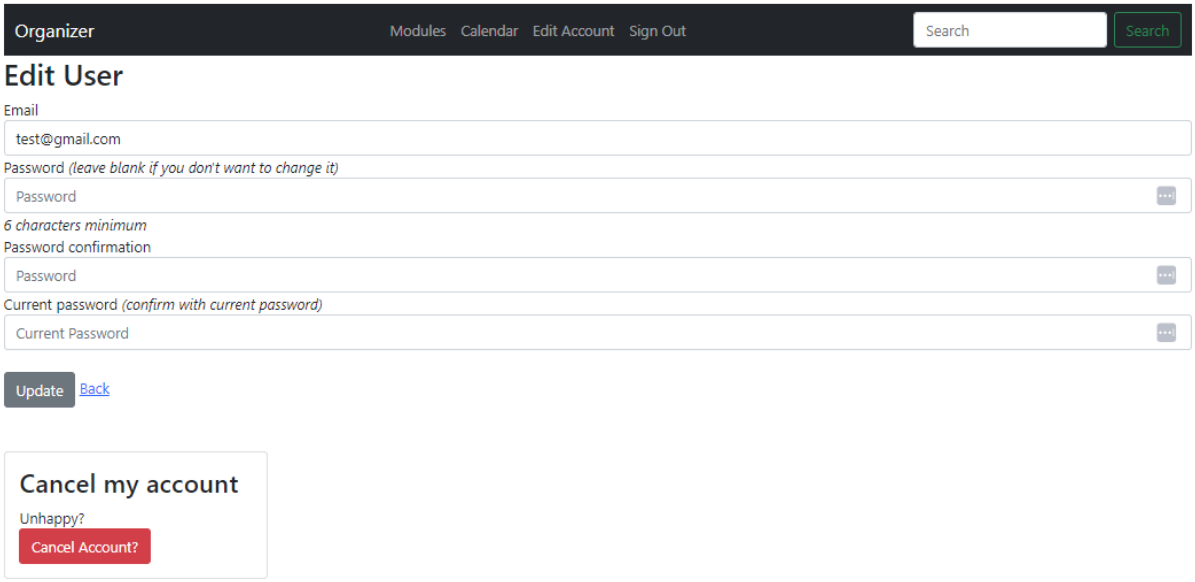
Delete list button:



Delete task button:



Edit Profile: Account settings to amend previously input data, i.e. password.



2.5 Testing

Fixtures For Tests:

Fixtures are done in the testing phase to give data to the system to input when testing the different functions, systems etc. This caused me quite a headache because the users were not originally linked to the modules when it was created.

'users.yml', 'events.yml', 'collegemodules.yml, & 'tasks.yml'

```
one:
  email: "user@test.com"
  encrypted_password: <%= Devise::Encryptor.digest(User, 'password') %>
  reset_password_token:
  reset_password_sent_at:
  remember_created_at:

two:
  email: "user2@test.com"
  encrypted_password: <%= Devise::Encryptor.digest(User, 'password') %>
  reset_password_token:
  reset_password_sent_at:
  remember_created_at:
```

```yaml
one:
  name: "Do Today"
  user: one

two:
  name: "Doing Currently"
  user: two
```

```yaml
one:
  module_name: "Programming"
  module_id: "1234"
  module_lecturer: "Ryan Lanz"
  user: one

two:
  module_name: "Security"
  module_id: "2345"
  module_lecturer: "Lanz Ryan"
  user: two
```

Individual Controller Tests:

```
# Running:

TasksControllerTest#test_should_update_task = 0.59 s = .
TasksControllerTest#test_should_destroy_task = 0.03 s = .
TasksControllerTest#test_should_create_task = 0.04 s = .
TasksControllerTest#test_should_show_task = 0.29 s = .
TasksControllerTest#test_should_get_edit = 0.02 s = .
TasksControllerTest#test_should_get_new = 0.02 s = .
TasksControllerTest#test_should_get_index = 0.02 s = .
ListsControllerTest#test_should_update_list = 0.02 s = .
ListsControllerTest#test_should_destroy_list = 0.02 s = .
ListsControllerTest#test_should_create_list = 0.02 s = .
ListsControllerTest#test_should_show_list = 0.02 s = .
ListsControllerTest#test_should_get_edit = 0.02 s = .
ListsControllerTest#test_should_get_new = 0.02 s = .
ListsControllerTest#test_should_get_index = 0.02 s = .
CollegemodulesControllerTest#test_Should_update_modules = 0.04 s = .
CollegemodulesControllerTest#test_Should_delete_module = 0.02 s = .
CollegemodulesControllerTest#test_Should_create_new_module = 0.02 s = .
CollegemodulesControllerTest#test_Should_show_modules = 0.02 s = .
CollegemodulesControllerTest#test_Should_retrieve_index_page = 0.02 s =
CollegemodulesControllerTest#test_Should_show_edit_page = 0.02 s = .
CollegemodulesControllerTest#test_Should_retrive_new_page = 0.02 s = .

Finished in 1.298620s, 16.1710 runs/s, 20.7913 assertions/s.
21 runs, 27 assertions, 0 failures, 0 errors, 0 skips
```

Individual Model Tests:

```
C:\Users\ryanl\calendar>rails test test/models -v
Running 13 tests in a single process (parallelization threshold is 50)
Run options: -v --seed 43282

# Running:

TaskTest#test_Should_work_with_name,_list,_user,_and_start_date = 0.12 s = .
TaskTest#test_Shouldn't_work_without_list = 0.00 s = .
TaskTest#test_Shouldn't_work_without_user = 0.00 s = .
TaskTest#test_Shouldn't_work_without_start_date = 0.00 s = .
TaskTest#test_Shouldn't_work_without_name = 0.00 s = .
UserTest#test_shouldn't_work_with_no_password = 0.00 s = .
UserTest#test_should_work_with_email_&_password = 0.00 s = .
UserTest#test_shouldn't_work_with_no_email = 0.00 s = .
ListTest#test_Shouldn't_work_without_name = 0.00 s = .
ListTest#test_Should_work_with_name_and_user = 0.00 s = .
ListTest#test_Shouldn't_work_withou_user = 0.00 s = .
CollegemoduleTest#test_Should_be_valid_with_a_user = 0.01 s = .
CollegemoduleTest#test_Shouldn't_be_valid_without_a_user = 0.00 s = .

Finished in 0.158935s, 81.7945 runs/s, 81.7945 assertions/s.
13 runs, 13 assertions, 0 failures, 0 errors, 0 skips
```

Individual Security tests:

```
# Running:

ListsSecurityTest#test_Block_access_to_other_user's_list = 0.62 s = .
ListsSecurityTest#test_Shows_index_when_signed_in = 0.04 s = .
ListsSecurityTest#test_Allow_access_to_signed_in_user's_list = 0.03 s = .
ListsSecurityTest#test_Redirect_to_login_page_if_not_logged = 0.01 s = .
XssProtectionTest#test_should_escape_HTML_in_login_form_input = 0.04 s = F


Failure:
XssProtectionTest#test_should_escape_HTML_in_login_form_input [test/security/xss_test.rb:19]:
Expected response to be a <2XX: success>, but was a <422: Unprocessable Entity>


rails test test/security/xss_test.rb:14

TasksSecurityTest#test_Redirect_to_login_page_if_not_logged_in = 0.01 s = .
TasksSecurityTest#test_Shows_index_when_signed_in = 0.03 s = .
TasksSecurityTest#test_Allow_access_to_signed_in_user's_task = 0.02 s = .
TasksSecurityTest#test_Shouldn't_allow_access_to_other_users_tasks = 0.01 s = .
CollegemodulesSecurityTest#test_Block_access_to_other_users_modules = 0.03 s = .
CollegemodulesSecurityTest#test_Shows_index_when_signed_in = 0.02 s = .
CollegemodulesSecurityTest#test_Allow_access_to_signed_in_users_modules = 0.02 s = .
CollegemodulesSecurityTest#test_Redirect_to_login_page_if_not_logged = 0.01 s = .
AuthenticationSecurityTest#test_login_should_be_protected_from_SQL_injection = 0.02 s = .

Finished in 0.927479s, 15.0947 runs/s, 16.1729 assertions/s.
14 runs, 15 assertions, 1 failures, 0 errors, 0 skips
```

CI/CD:

Project is deployed through Heroku successfully. Initially done through AWS but issues started arising through being deployed through a production environment instead of development environment.

Acceptance Test Functional Requirements Testing Results:

FR-Number	Functional Requirement Title	Functional Requirement Description	Testing Results
FR-1	Modules	Module Management which should allow the user to manage their course modules. This includes adding, viewing, updating, and deleting said modules. Buttons should be provided for each of the corresponding actions and need to function as intended. A display must be in place to show the modules and information included about them including a name, ID, and lecturer.	Test 1: Success as the user can add a module, including name, ID, and lecturer. Test 2: Success as the user can edit modules. Test 3: Success as the user can view modules. Test 4: Success as the user can delete modules. Test 5: Failed as the user was able to add a module with no fields filled in.
FR-2	Scheduling	-Date Management is required where users can schedule events and have key dates input in their calendar. A monthly calendar needs to be included which is interactive and allows users to add and edit dates. Specific information needs to be included where the user can add time, name, description, and view said dates when checking the calendar.	Test 1: Success as the user can add events with time and description. Test 2: Success as the user can edit events. Test 3: Success as the user can view events in the calendar. Test 4: Success as the user can delete events. Test 5: Failed as the user was able to add an event with no fields filled in. Test 6: Success as the user can create lists with a name and user.

			Test 7: Success as tasks can be moved within lists. Test 8: Success as tasks cannot be created without a user. Test 9: Success as tasks cannot be created without a name. Test 10: Success as tasks cannot be created without a start date. Test 11: Success as tasks cannot be created without a list. Test 12: Success as lists can be shown. Test 13: Success as lists can be accessed via the index. Test 14: Success as new list pages can be retrieved. Test 15: Success as lists can be updated. Test 16: Success as lists can be deleted
FR-3	Users	User inputs details from registration and the system authenticates them, should be done with a username and password. Needs to advise user if there are incorrect details entered with exception handling. Successful authentication authorizes the user to access the programs' features. User can Sign out from program successfully.	Test 1: Success as the user can update details without being logged out. Test 2: Success as the user receives confirmation upon successful updates. Test 3: Success as users can log in with correct credentials. Test 4: Success as users are blocked and shown an

			error for incorrect usernames. Test 5: Success as users are blocked and shown an error for incorrect passwords. Test 6: Success as users can successfully sign out.

Acceptance Test Non-Functional Requirements Testing Results:

NFR-Number	Title	Non-Functional Requirement Description	Criteria	Testing
NFR-1	Reliability	Functions work as intended. Consistency is essential. Must always have high level of uptime and not have any crashes. Exceptions can occur from user misusing a function, but they must be caught and the system cannot incur fatal errors.	**Uptime:** 99.9% **Exceptions:** All Caught	Success: Thorough testing done of using the application. Exceptions and errors were created for every function where necessary, and user will incur no fatal issues. User is always informed when they have used something incorrectly.
NFR -2	Usability	U.I. needs to be user-friendly, easily accessible, and easy to navigate. Buttons and labels must be clear and input fields need clear instructions. Users should not spend more than >5s trying to find any given option. Should not select incorrect option more than once.	**Navigation Time:** <5s **Incorrect Options Selected:** <2 times	Success: Employed user testers for this phase to check times. Users spent +/-4 seconds navigating menu to view options available. No testers clicked wrong options.

NFR -3	Security	Ensure the application is secure against unauthorized access and vulnerabilities such as SQL injection and XSS attacks. Only authenticated users should access their data, and proper error handling should be implemented for invalid input attempts.	**Encryption:** Password, BCrypt, devise **Database Authentication:** Values 100% correct	Test 1: Failed as the XSS protection test returned a 422 error instead of a 2XX success response. Test 2: Success as SQL injection attempts in the login form are blocked. Test 3: Success as only signed-in users can access their tasks. Test 4: Success as unauthorized users are redirected to the login page. Test 5: Success as users cannot access other users' tasks. Test 6: Success as only signed-in users can access their lists. Test 7: Success as unauthorized users are redirected to the login page when trying to access lists. Test 8: Success as users cannot access other users' lists. Test 9: Success as only signed-in users can access their modules. Test 10: Success as unauthorized users are redirected to the login page when trying to access modules.

				Test 11: Success as users cannot access other users' modules.

2.6 Evaluation

With the project having been evaluated now against functional and non-functional requirements. Steady progress has been made with functional features, and although not at commercial deployment level yet, promise is shown with the foundation laid. As this is useful for me, I'll be continuing to build it and increase the complexity, scope, and adding features. Let's have a run through below.

Functional Requirements:

The functions of the program have been implemented at it's core but have room to improve. The user registration and login are there, with devise creating a secure and easy user management system. The CRUD database for module, list, and task management was implemented successfully along with it being tied to the user that is signed in for security. Everything has been rigorously tested with all functions working as intended. Although the features are there, improvements need to be made with the modules section most. It doesn't serve much purpose at the minute, the ability to add/edit/update/delete modules exists but the addition of a file upload service and any other corresponding features to service it isn't there. This needs to be integrated into the scheduling aswell to link it with say, class times for modules and what needs to be submitted.

Non-Functional Requirements:

All non-functional requirements working correct and efficiently throughout the tests. With regard to reliability, usability, security, as well as performance the tests perform under or on the testing acceptance times. Further tests need to be run on future functions and system updates but everything all good for now.

3 Conclusions

With significant strides made to the project and a solid foundation for future work to be done, there is plenty of promise for the future of it. I'm happy with the work done, especially the implementing of sortable to lists/tasks/calendar as it was extremely difficult to get it functional the way it was set up in the functional requirements but from perseverance and research it was possible.

4 Further Development or Research

With the future of the project there are more features to be added:

- File Upload & Downloading for storage and accessibility. This is key to the project as the goals include having your work in a single space.
- Contact/Feedback page with potential for simple AI implementation of sorts. Definite addition of a feedback with mailer properties for submitted issues.
- Add features to module management with it being intertwined more in the calendar and with the other features.

5 References

Bibliography

Dedecker, J. (n.d.). *No Route*. Retrieved from Stackoverflow: https://stackoverflow.com/questions/6557311/no-route-matches-users-sign-out-devise-rails-3

Devise. (n.d.). *Devise Gem*. Retrieved from rubydoc.info: https://rubydoc.info/github/heartcombo/devise

Github. (n.d.). *SortableJS*. Retrieved from Sortablejs.github.io: https://sortablejs.github.io/Sortable/

6 Appendices

6.1 Project Proposal

National College of Ireland

Project Proposal

Computing Project

28/10/2023

Computing

Software Development

2023-2024

Ryan Lanz

x21759971

x21759971@student.ncirl.ie

Contents

6.1.1 Objectives

The aim is to develop a convenient application that assists students and professionals in organizing their tasks in a concise, professional, and effective manner. The application will address the common challenge of individuals struggling to complete tasks without clear guidelines and difficulties in tracking progress and archiving information. The vision for the application is to create a unified platform, combining the functionalities of Moodle and Microsoft Teams, to provide users with a comprehensive solution.

The expected impact and outcomes of the project include improved timekeeping, streamlined archiving of documents, enhanced accessibility, reduced scheduling conflicts, and overall increased benefits and efficiency for all users.

There are several requirements for this project that need be fulfilled including user authentication, task management and scheduling with different features, feedback methods, all the while being highly usable, secure and perform well.

6.1.2 Background

The motivation behind this project stems from the daily use of Teams and Moodle, recognizing their usefulness while desiring a different user interface and integrated functionality. The recent upgrades to the Moodle user interface have improved its usability, but there is still room for enhancement in terms of user experience and layout.

The project is driven by the belief that it will significantly enhance productivity, boost morale, improve managerial skills, and facilitate timely task completion. In an era where remote work has become prevalent, it is easier to become distracted and lose track of time. By consolidating all tasks in a single platform, the application aims to mitigate these challenges.

I created a project on this previously as a basic program using java, GUI, and MySQL and would like to utilise Ruby on Rails and CSS to create a functioning website as it is much better to create with and can be used anytime through a web browser on a computer as well as a phone.

I have to create a cloud application with Ruby on Rails for another module before Christmas with CRUD capabilities and plan to create a base of this project for that and then build upon it further. The requirements could all be fulfilled solely with that framework but I might implement something like React or Angular if it is going well and can be picked up easily enough.

6.1.3 State of the Art

There are loads of applications that are similar, ones like Todoist (*I found I didn't like the overall UI of it and found myself not wanting to add any tasks to it and didn't like how the tasks were added. I disliked this the most and don't understand why it's as highly regarded.*), Trello (*Trello I really liked the UI but thought it was lacking some features. I would love to create an interactive calendar with similarities to this*), Toodledo (*This app I didn't like the UI of it at all either, it didn't feel like the application it's meant to be because of the way it's laid out. However, I really liked all the features available in it.*), OmniFocus (*I couldn't use this really, not for any reason other than I feel this is for people who have created a productivity and management system for themselves already over time and need to implement it and use it an application to accommodate it.*), and I have tried multiple of them but they have never worked for me. I have found myself not drawn in by them and end up just deleting them in the end as I don't utilise them. I would like to compile the things I liked from these apps and create a compilation of those features and designs.

For my application I would like to take elements I like from Toodledo and implement some UI features from Trello as well as adding my own features to it as well. That would include features from Moodle/Teams like file storage of documents etc that correspond to different modules, as well as assignment and exam grading sections to keep track of that. If I combine all these elements successfully, I feel it would be a unique project compared to the others and be extremely helpful.

6.1.4 Technical Approach

The methodology chosen for this project shall be PRINCE2. PRINCE2 allows for a structed and controlled way of project management and sections the whole project into different divisions or stages while clear goals and responsibilities are emphasized. This allows for every role and responsibility of every section within the project to be defined clearly with a goal, milestone, and timeframe. There are regular checks and reviews to gain full transparency and keep realistic goals of project completion as well it being good for organization and communication. Internal stakeholders include developers, QA, and team members involved on the project.

I have program down the project into 6 different stages in order to follow complete it in a timely manner and not fall behind and try to avoid delays.

Time:

Stage 1: Requirements Gathering & Proposal

Start date: 18/10 **End Date:** 28/10 **Tolerance:**

Stage 2: Set Up Development Environment & Commence Coding

Start date: 31/10 **End Date:** 10/12 **Tolerance:**

Stage 3: Mid-Point Implementation (Finish v1 of project with CRUD capabilities)

Start date: 11/12 **End Date:** 15/12 **Tolerance:** +/- 1 Week

2024

Stage 4: Commence Coding For v2 (Full functioning UI and all requirements fulfilled)

Start date: 20/01 **End Date:** 01/04 **Tolerance:** +/- 2 Weeks

Stage 5: Acceptance Testing

Start date: 02/04 **End Date:** 15/04 **Tolerance:** +/- 2 Weeks

Stage 6: Commence Coding For v3 & Finish Documentation

Start date: 16/04/06 **End Date:** 05/05 **Tolerance:** 5 Days

Final Submission: 15/05

6.1.5 Technical Details

Frontend Development and User Interface: The user interface and frontend experience will be constructed using Ruby on Rails (RoR), HTML, and CSS. The versatility of RoR allows for seamless integration with HTML and CSS,

improving the UI and making the interactivity with the application easy to use. With CSS being utilised it allows for easy editing and real-time incorporation of ideas and the program you imagine making is created exactly how you want to make it. With all these languages integrated it will mean the application will be user-friendly and useful. Backend Development: Ruby on Rails will be used for the backend development of this project as well. We have began developing with it and it is easy to use and has a lot of support with various frameworks, libraries with plenty of services like APIs and such. It is brilliant for backend as you can create websites extremely quickly compared to other languages like Java or Python and they don't lack and professionalism despite the quick creation.

Backend Development: Ruby on Rails will also power the backend development. Its user-friendly syntax, extensive library support, and numerous frameworks simplify the creation of complex backend functionalities. RoR's efficiency stands out, enabling rapid development without compromising professionalism. The availability of diverse services like APIs further enriches the backend capabilities, ensuring a robust and dynamic application. RoR's Model-View-Controller (MVC) architecture guarantees clean and organized code, facilitating easier debugging and future enhancements.

Database management: PostgreSQL will be employed for the management of the database. The utilization of this kind of database management is useful and widely used for applications that rely on CRUD operations. This enables quick and efficient data retrieval and also supports any kinds of queries the user might have in relation to this program. Additionally, it offers support for JSON, which can be implemented in this project as well.

Hosting: Heroku seems to be the most reasonable choice for this project as there is a free tier which can work for small projects while still performing well and suits the small scale of the project. Alongside this that, it focuses on simplicity and a straightforward deployment process which is helpful for new users and a project of this scale.

6.1.6 Project Plan

Business Requirements:

- Feedback focused: The program aims to evolve and scale with user-submitted feedback shaping how the program will look and act as no-one knows something as much as the person who uses it most. This is a good look for business as well as customer care and support show good company morals and willingness to listen to the customer based shows they are valued.
- Performance based: Program needs to perform for users smoothly with no errors, appropriate error and exception handling needs to be in place with any potential overlooked issues reported automatically by the system where possible, and if system does not pick it up allow for customer feedback to report it.

Functional Requirements:

- User Registration, Authentication, Authorization: Users must be able to create an account on their own accord and be able to login with the details they have provided. They must be authorized to use any functions appropriate to them without assistance.
- Management: User must be able to use management facilities of all functions, scheduling for adding and editing key dates in their personal calendar, module management for adding, viewing, updating, and deleting modules appropriate to them that are in their course, account management where they have the ability to amend details about themselves.
- File Storage: User must be able to upload & download files that correspond to the module they were uploaded to. They must be able to add and remove them as such.
- Grading System: User must be able to utilise a grading system for assignments and exam results. This should show a tally of current progress.

- Feedback: Must be able to submit feedback that can be used to benefit the program in the future. Whether it be issues with the program to fix, advice on what can be improved in the future, or just general praise or complaints users might have.

Non-Functional Requirements:

- Usability: Program is easy to use, users should spend much time navigating. <5s navigating.
- Reliability: Program is consistent and works as intended, any issues need to be reported. Uptime: 99%
- Performance: All functions of application load instantly. Allow for slight delay in startup of program which could depend on specs of the user's device. Startup <5s & Functions load time <1s.
- Security: Security measure implemented and compliance with GDPR is mandatory. Encryption of user data must be used and comply with GDPR rules with how it is stored.

Time Requirements

Phase	Milestones	Timeframe
Planning & Learning	-Requirements Gathering -Project Proposal & Plan Complete	18/10-28/10
Implementation	-Roles and Responsibilities Defined -Development Underway	31/10-10/12
Mid-Point Review	-V1 Completed -Documentation Compiled & Video Showcase Completed	20/12/2023
Final Implementation	-V2 Development Underway	20/01/2024-01/04
Acceptance Testing	-Acceptance Testing Underway	02/04 to 15/04
Close Project	-Any Acceptance Testing Issues Fixed -Finalize Documentation -Check All Deliverables	16/04 – 05/05
Finishing Review	-Documentation Prepared -Video Presentation Completed	11/05 to 15/06
Finish	-Submit Full Project	15/05/2024

6.1.7 Validation

The validation and verification of different aspects of the project will be done throughout the course of the year. There will be 3 main points of validation:

Mid-Point Review: The mid-point review will prove crucial to assess the progress of the project. It is a milestone for progress and the success of the project so far can be judged from here to an extent. It will be used for knowing what functions have been implemented successfully and if there are any issues with them, and issues that could arise later on in the development process. If there have been

any deviations from the original plan, they can be focused on in-depth in the next development phase and ironed out quickly.

Acceptance Testing: The acceptance testing is a crucial stage of any project regardless of who is undertaking it. The acceptance testing stage is where all the work that has been put into the project is thoroughly tested in comparison to the requirements that were set out and the beginning of the project. This is a stage that makes or breaks the application and tests if it has been successfully implemented and is what was originally proposed. All the requirements set out previously (business, functional, non-functional) will be tested thoroughly and any errors and can hopefully be rectified in the final review.

Final Review: The final review will be the last stage of the project before it has been submitted. Any issues from acceptance testing, if needed to be fixed, will have a final attempt at rectification before the project is submitted. Not only will the code be validated but the documentation needs to be verified and validated thoroughly as well that is represents the end product correctly.

6.2 Reflective Journals

_____

Signature