A Cloud-Based Training Management System

A Requirement for the

Advanced Software Engineering Course (SDEV 4304)

Submitted by:

Ghadeer Alhayek (220191207)

Ayah Elshawa (220191521)

Raghad Alkhatib (220190759)

Malak Tafesh (220191354)

Submitted to:

Eng. Dr. Rebhi Baraka

Abstract

We developed a cloud-based training management web system that is designed to streamline and automate training processes in organizations. TMS offers a user-friendly interface that allows managers, advisors, and trainees to efficiently manage and track training activities. It facilitates creating training programs and meetings, tracking trainee's attendance and progress, and providing interactive learning materials. With features such as registration approval, resolving meeting conflicts, performance assessments, and reporting tools, TMS ensures that training initiatives are effectively implemented and monitored. Its cloud-based architecture allows for easy scalability and accessibility, making it ideal for organizations of all sizes.

1. Introduction

The cloud-based training management system is a web application designed to streamline and automate the management of training programs. It provides a centralized platform for trainees, managers, and advisors to collaborate and track the progress of training activities. TMS offers features such as trainee registration, course enrollment, progress tracking (through attendance forms), and communication tools (using emails).

We developed TMS by following the agile methodology, which emphasizes iterative development, frequent feedback, and collaboration. This approach allowed us to continuously improve and adapt to our changing comprehension of the requirements throughout the development lifecycle. The use of agile practices facilitated quick delivery of new features, enhanced responsiveness to the requirements, and efficiently managed our project timelines.

Overall, the cloud-based training management system provides a comprehensive solution for managing trainees, advisors and training programs efficiently and effectively. Its cloud-based architecture, and our approach with the agile development methodology, ensured flexibility, scalability, and responsiveness in meeting all of TMSs' requirements.

2. TMS's Requirements

TMS consists of 3 components, the manager, advisor and trainee. All these components fulfil the following requirement

1. Trainees:
    - They must register by providing necessary information, uploading personal files and documents upon registration.
    - They can apply for training programs, fill in attendance forms, and request meetings with their advisors.

2. Advisors:
    - They must provide necessary information for registration.
    - They are classified based on their discipline, manage their own trainees and keep up with their attendance

- They can approve or reject trainee meeting request.

3. Managers:
    - They review training and registration requests by both advisors and trainees and decide whether to accept them or not.
    - They generate unique trainee and advisor IDs, handle everyone's account management, manage training programs, and billing issues.
    - They can update everyone's data in the cloud.
    - They can send emails to advisors and trainees.

4. The System (TMS):
    - TMS is responsible for generating highly secure and random passwords for users upon manager's approval of their registration request.
    - It's also responsible for resolving meeting schedule conflicts.

- **TMS's UML Diagram**

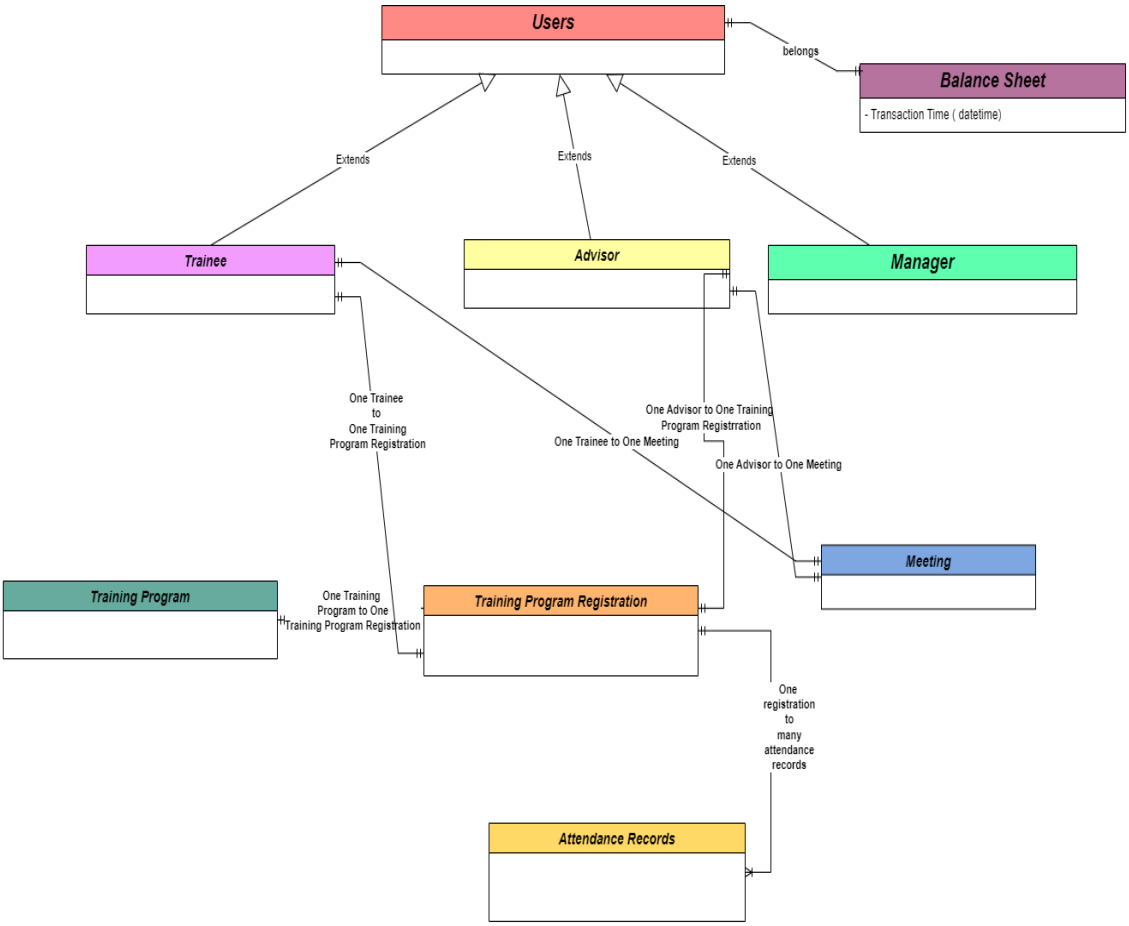


Figure 1- TMS's Componenr UML Diagram

- **TMS's Use Case Scenario**

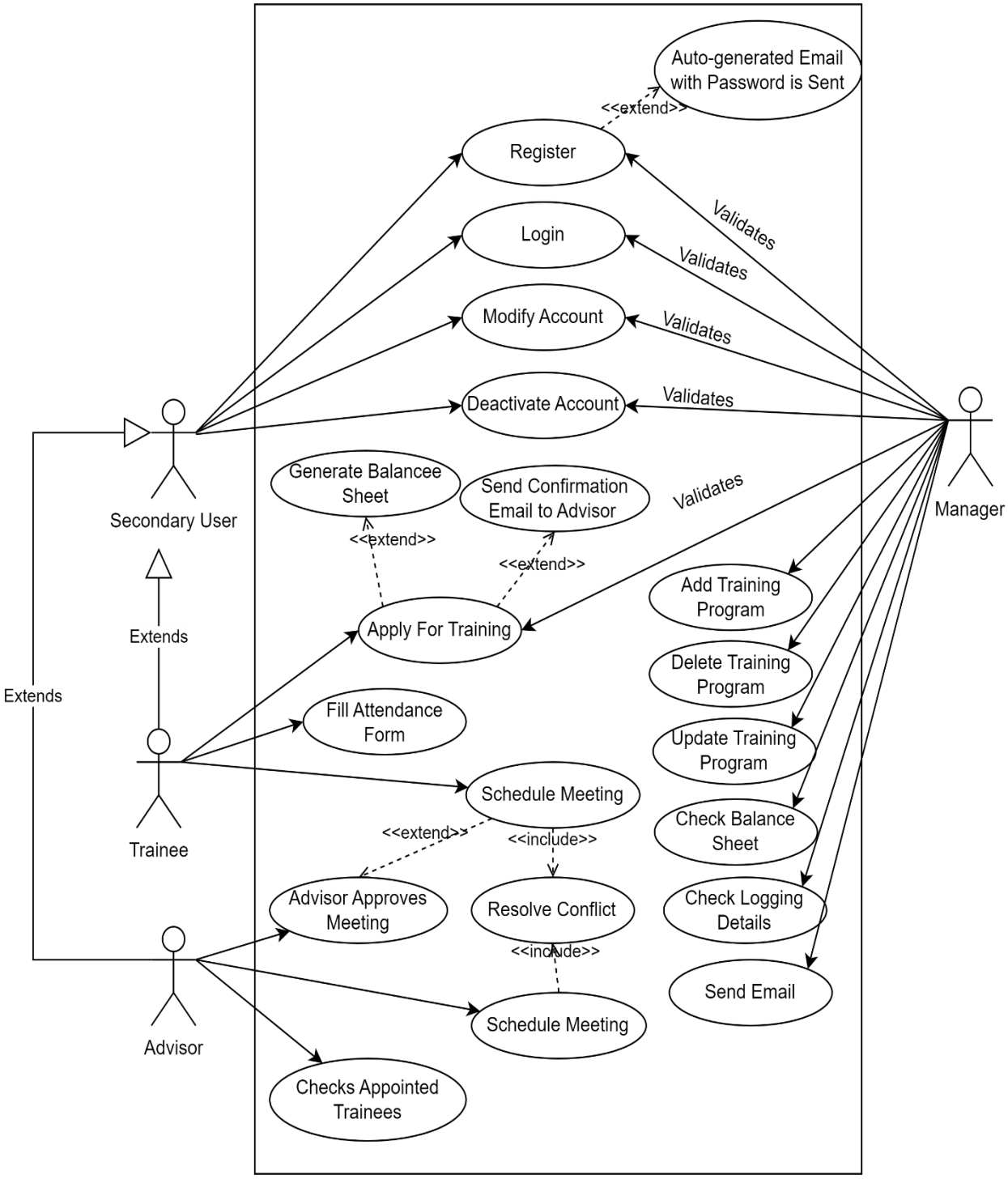


Figure 2 - TMS's Use Case Scenario

3. TMS's Architecture and Design

Based on the requirements of TMS, we decided that to build a three-tier architecture system. It consists of, the presentation tier, application tier, and data tier.

1. Presentation Tier (Frontend): also known as the frontend, is responsible for handling the user interface and displaying information to the users. It consists of an HTML frontend that enables users to interact with TMS.

2. Application Tier (Backend): also known as the backend, encompasses the business logic and application processing. It includes a Flask backend that processes requests received from the frontend, interacts with the MySQL database, and returns responses to the users.

3. Data Tier (Database): or the database tier, is responsible for storing and managing the data required by TMS. In our case, we hosted a MySQL database on Amazon RDS to store and retrieve data related to trainees, managers, advisors, and the overall system.

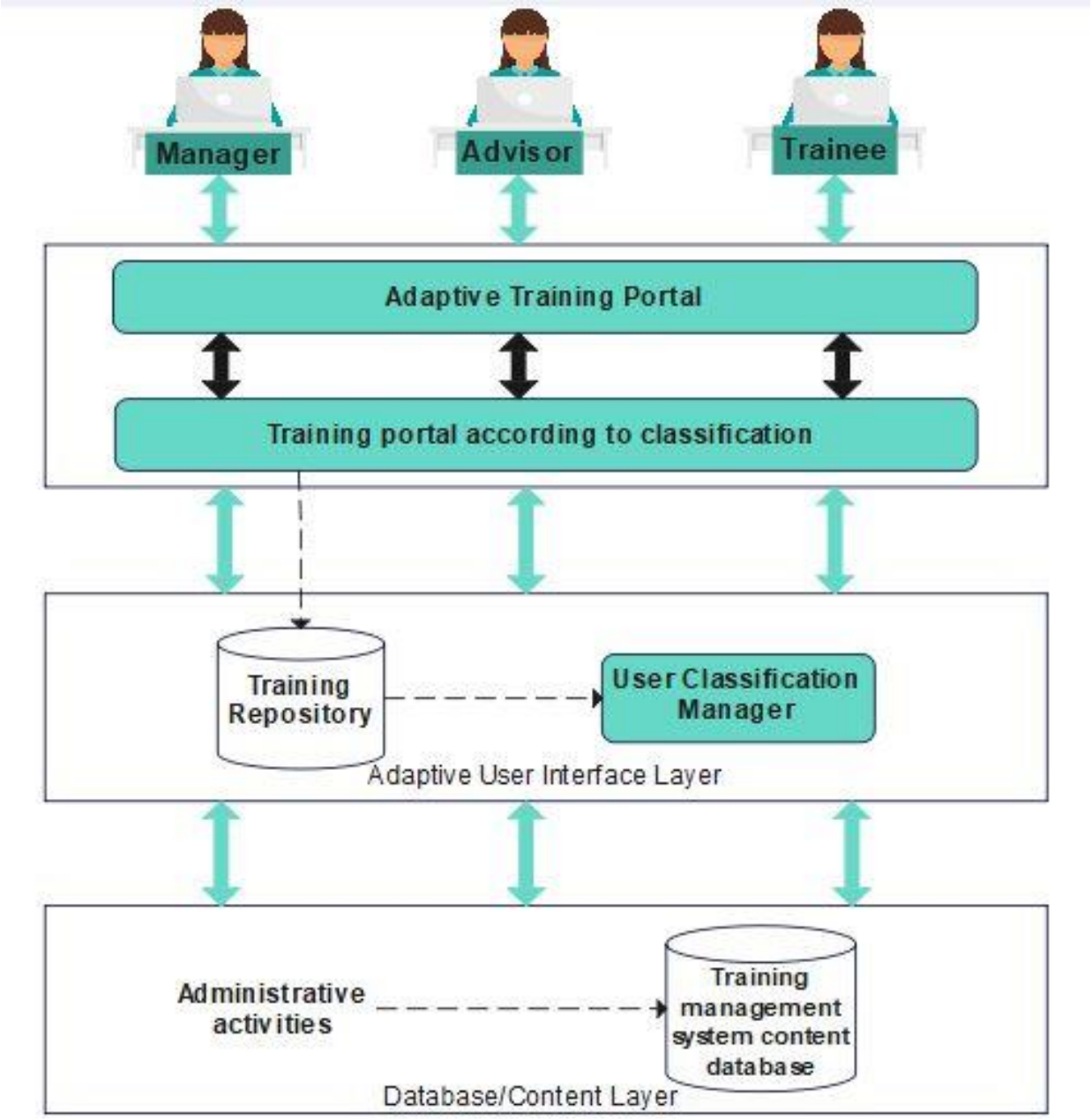


Figure 3 - Abstract View of TMS's Architecture

- **Functional components of TMS:**

  - The provides user registration and login functionality to allow trainees, managers, and advisors to access their respective accounts.

  - Trainees can upload their CVs upon registration, view their own progress, and communicate with their advisors via scheduling meetings.

- Managers have the ability to view and manage trainee profiles, assign advisors, and monitor overall training progress.
- TMS is integrated with external services for email notifications (Amazon SES) and user authentication (Amazon IAM).
- Advisors are able to review their trainees, provide feedback, and approve or schedule meetings with their trainees.
- TMS should support email functionality for important updates and actions, and resolve any scheduling conflict in meetings.

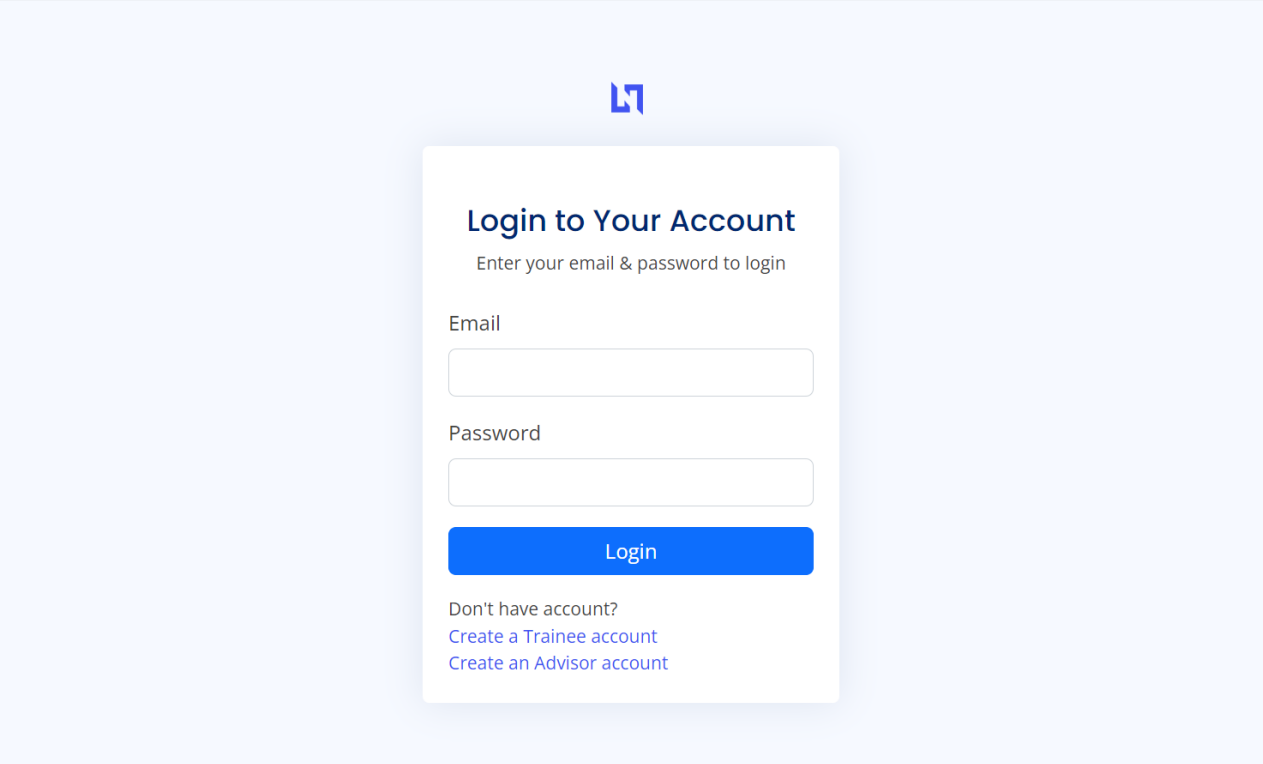
- **Data Management Requirements:**
  - TMS stores and manages trainee, manager, and advisor profiles, including personal information, contact details, and role-specific data..

- **Security and Access Control Requirements:**
  - TMS implements secure user authentication and authorization mechanisms to ensure only authorized individuals can access specific features and data.
  - Trainee, manager, and advisor accounts are protected using strong password encryption.
  - Role-based access control are enforced to restrict access to sensitive information and functionalities based on user roles.

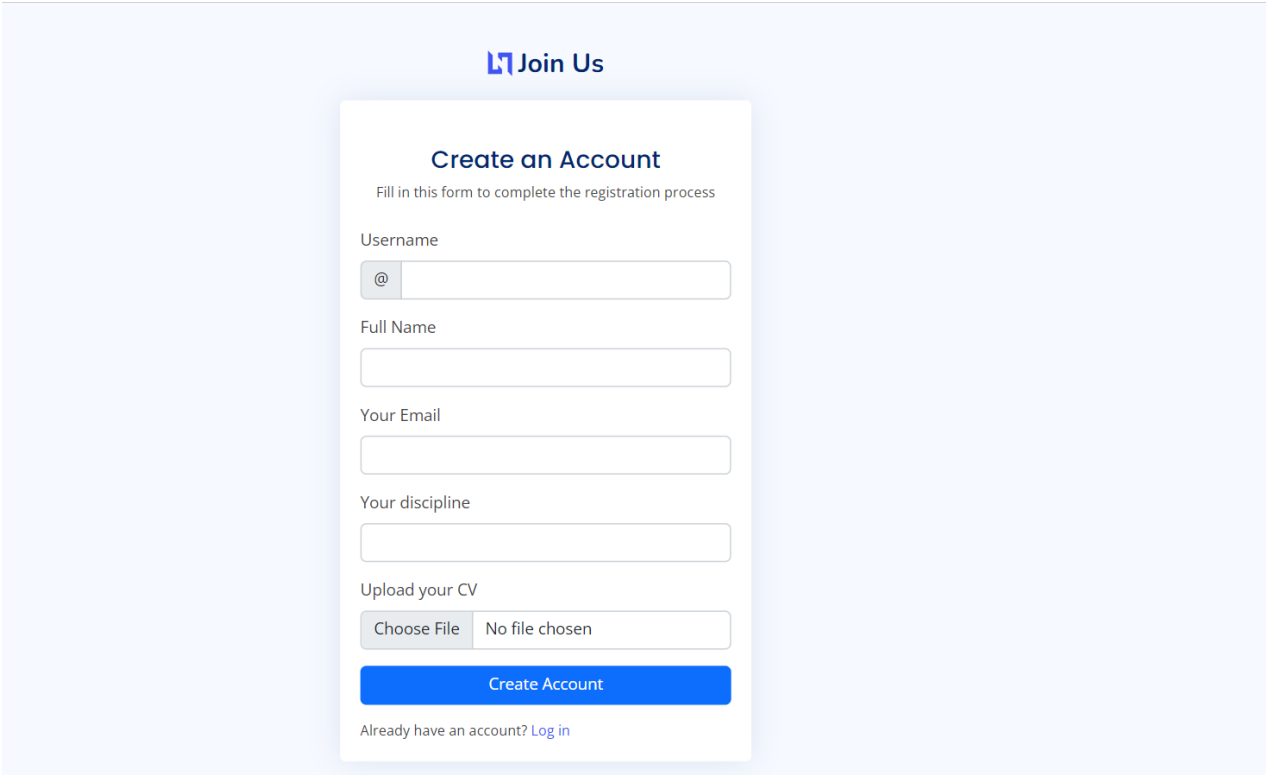
- **Scalability and Performance Requirements:**
  - TMS should be designed to handle a growing number of users, training documents, and concurrent interactions.
  - Performance optimizations should be implemented to ensure fast response times and efficient data retrieval.

- [User Interfaces]

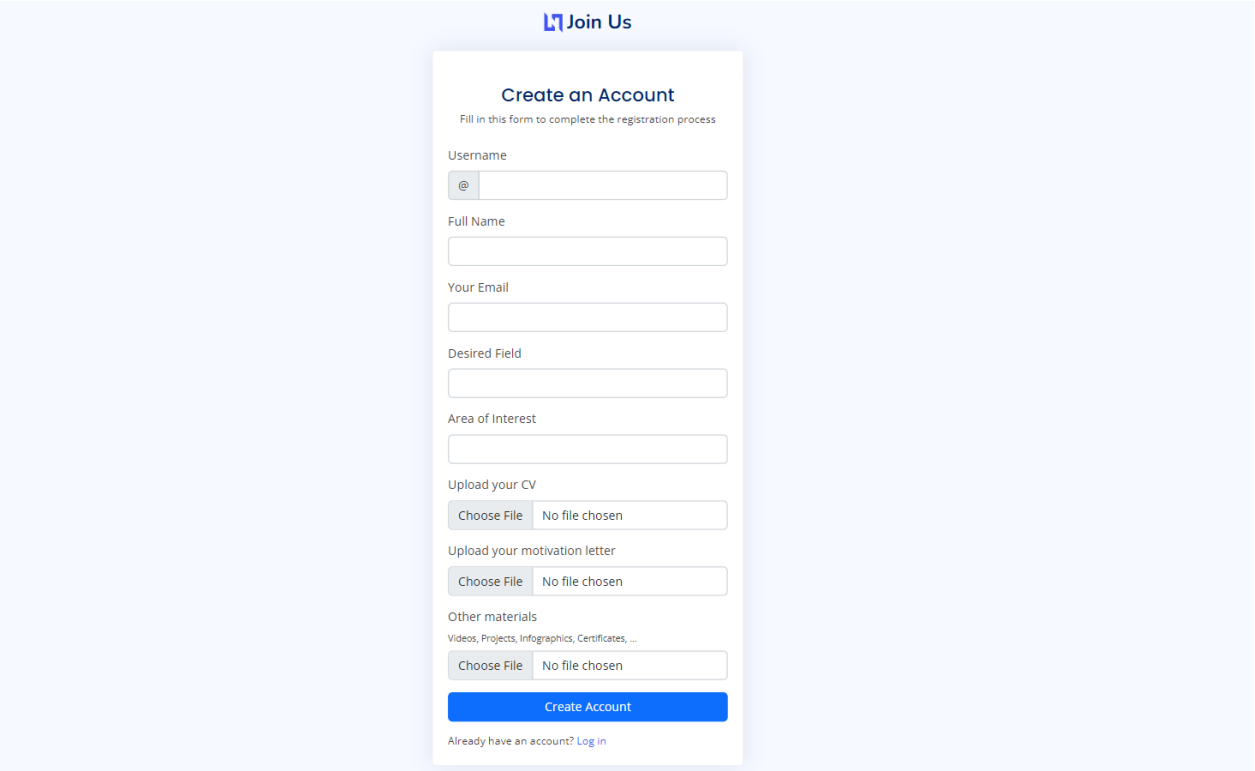
In this section, we will showcase some of TMS's interface for readability reasons. Please feel free to check out TMS and experience being an advisor or a trainee and see real time interfaces for yourself!



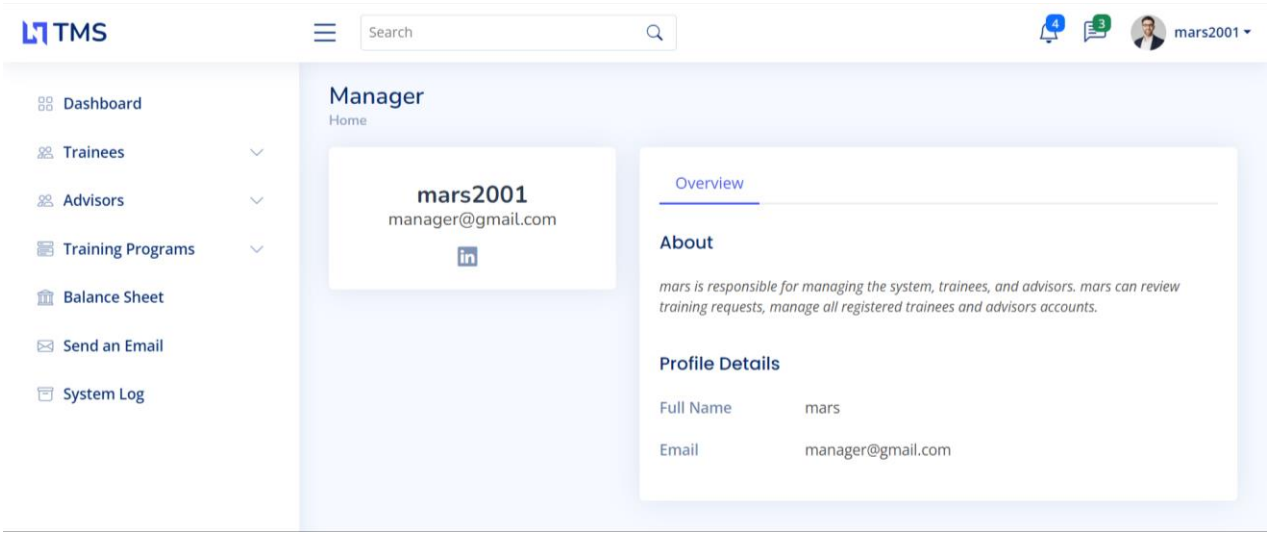
The Login Interface for all User



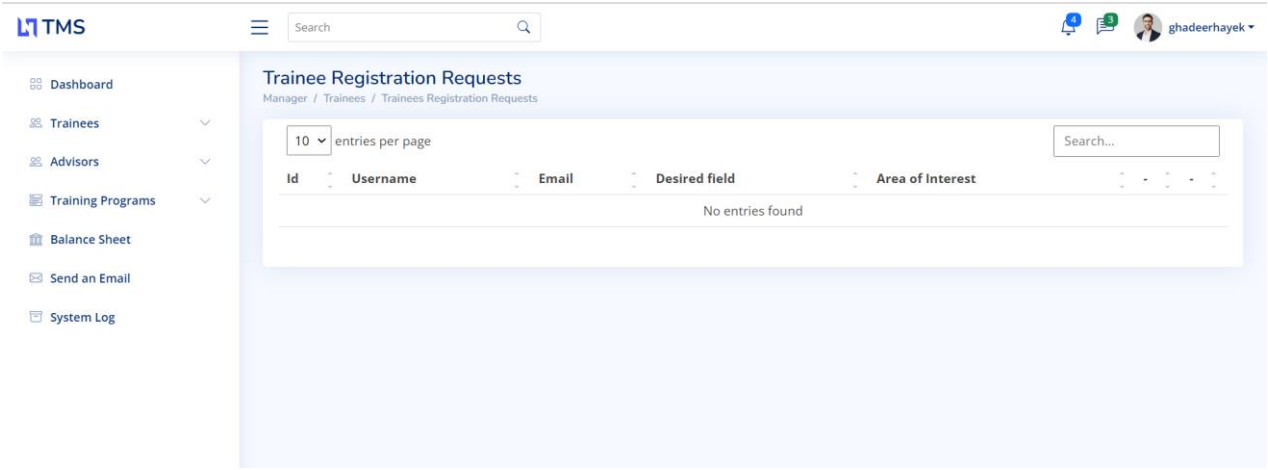
Create Advisor Account



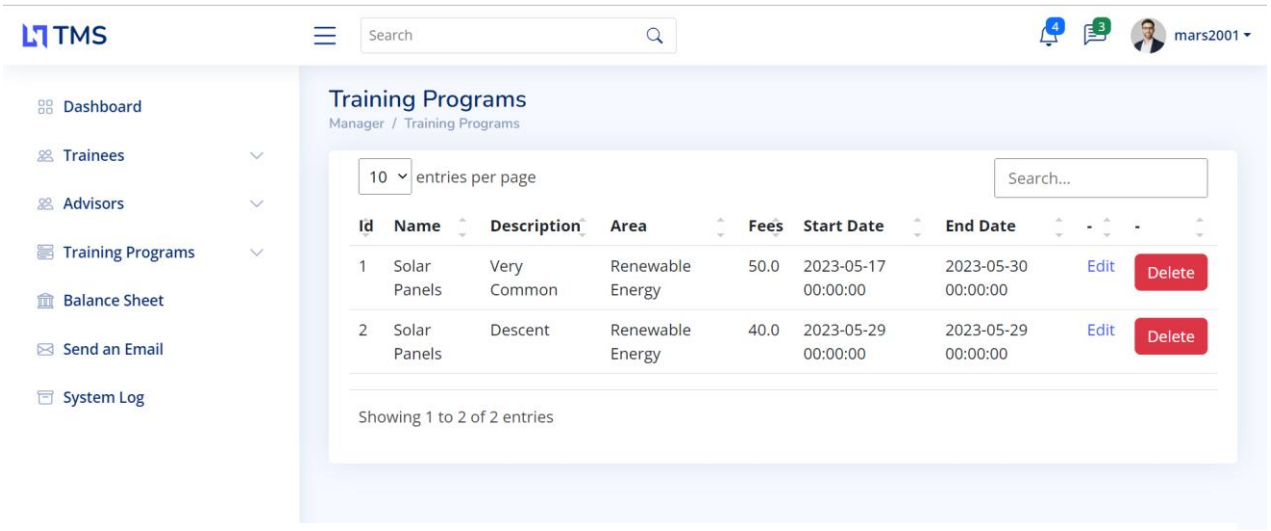
Create Trainee Account



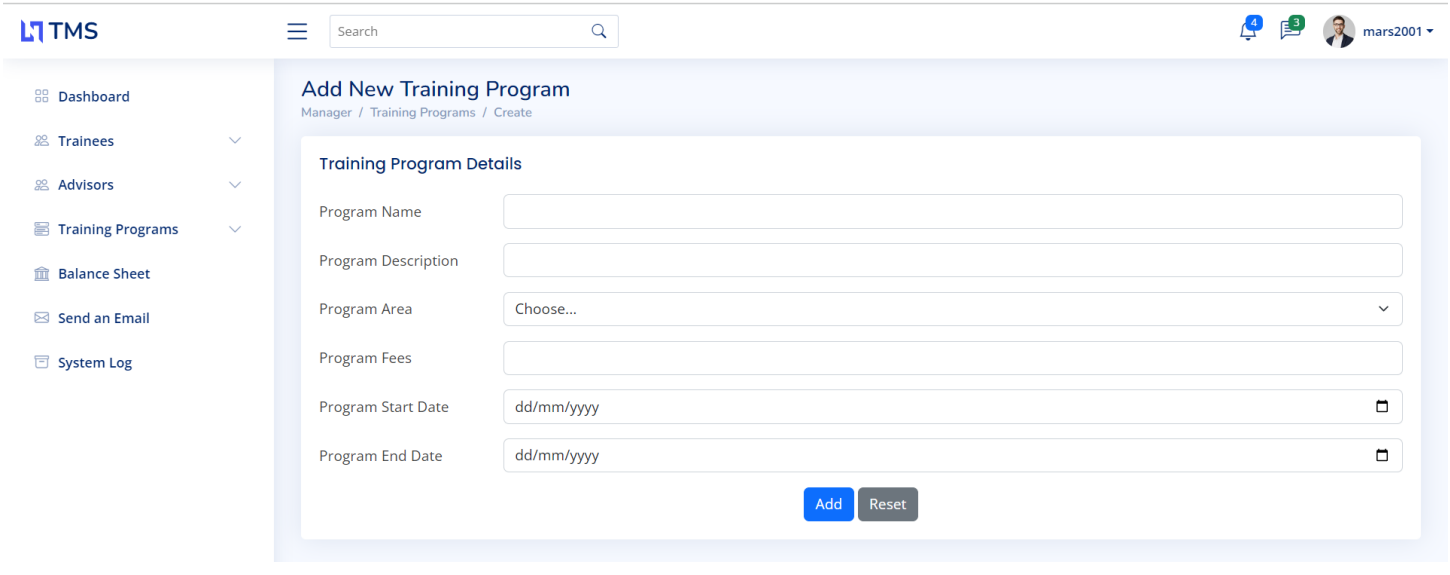
Manager Dashboard



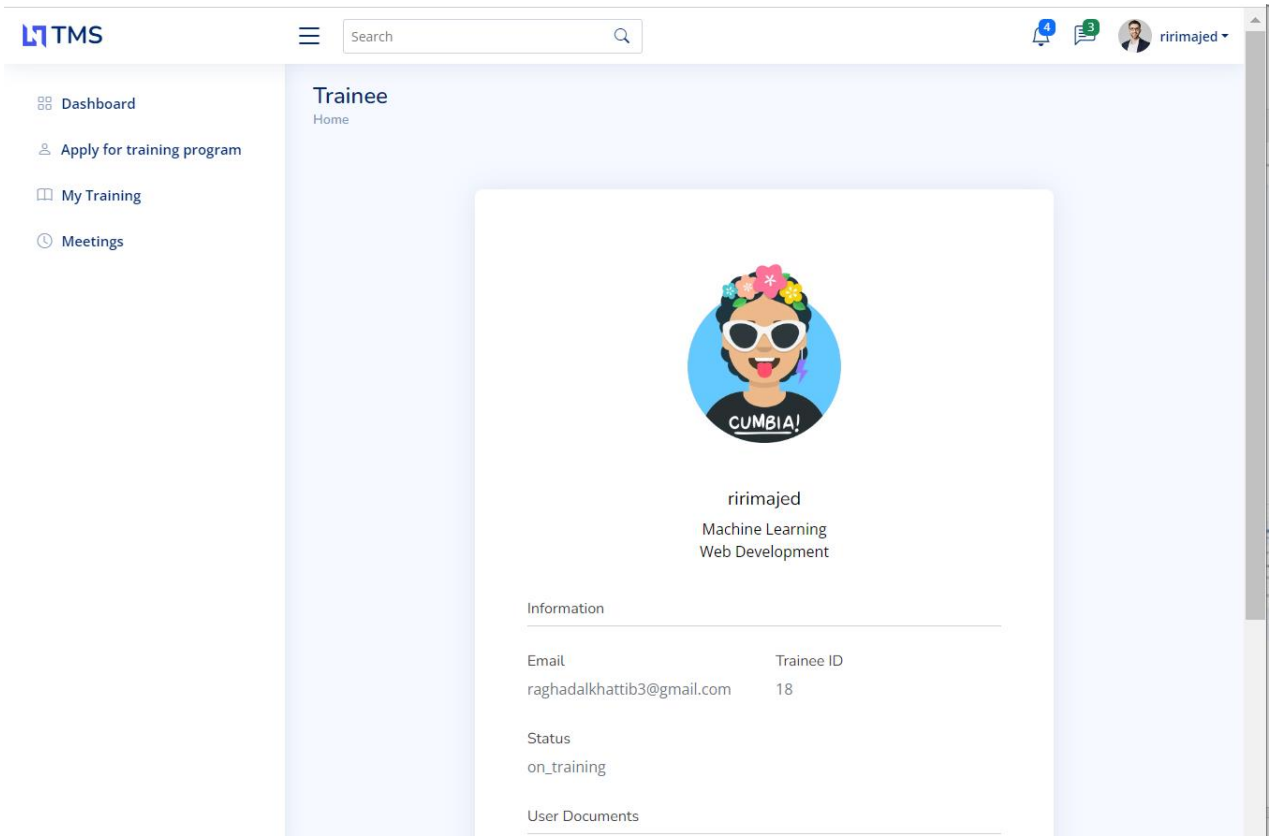
All Trainee Registration Requests (No new Trainees at the moment ☹)



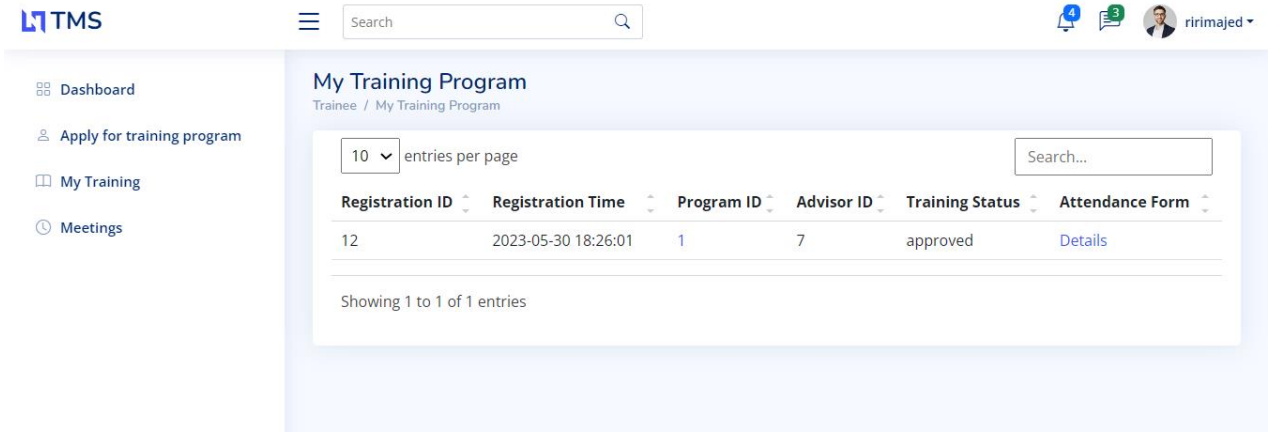
All Programs



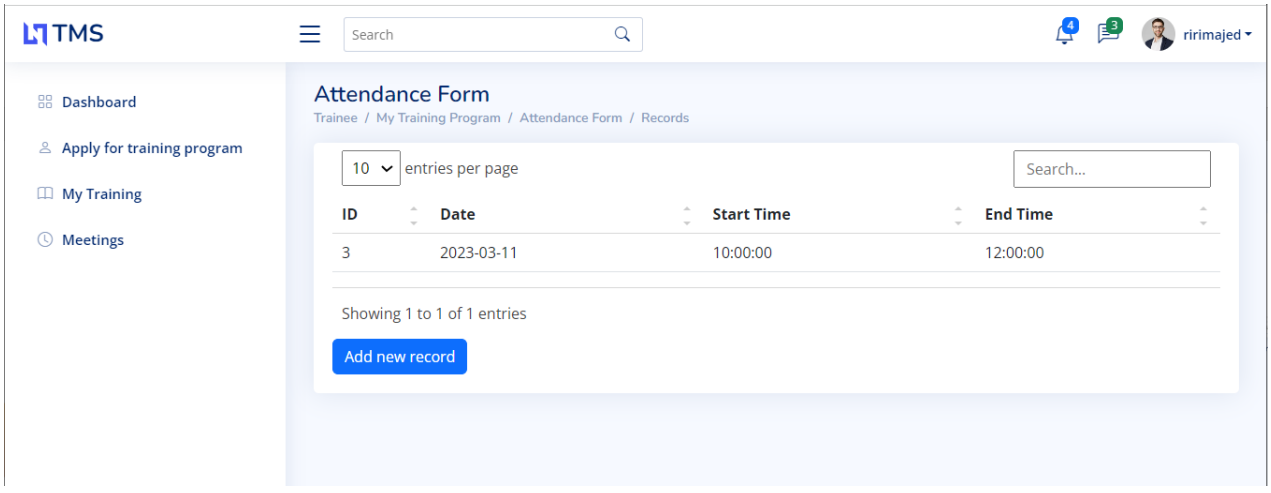
Add Programs



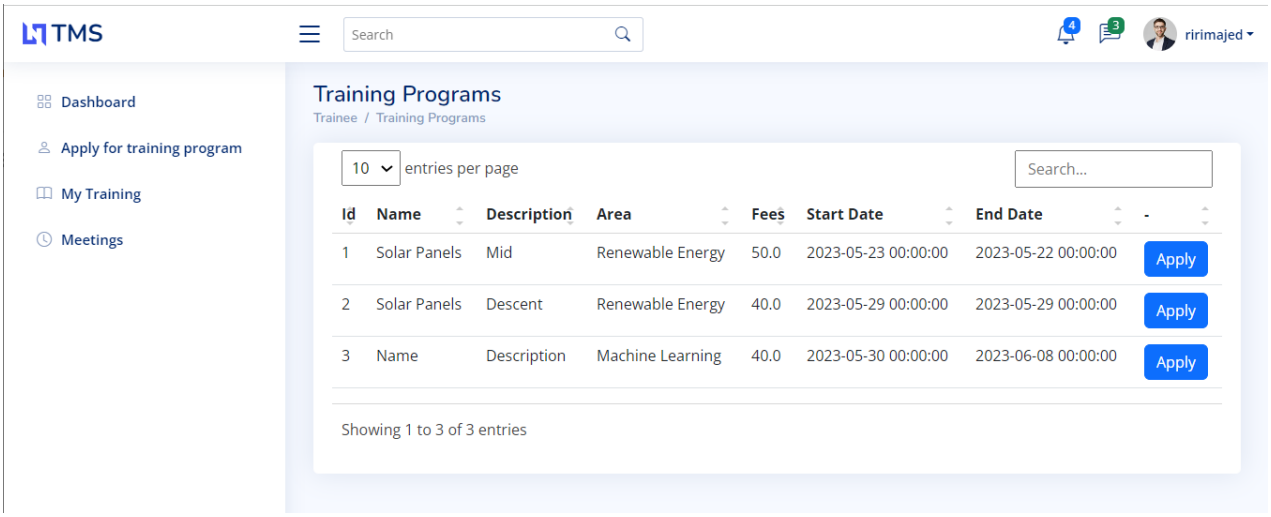
Trainee Dashboard



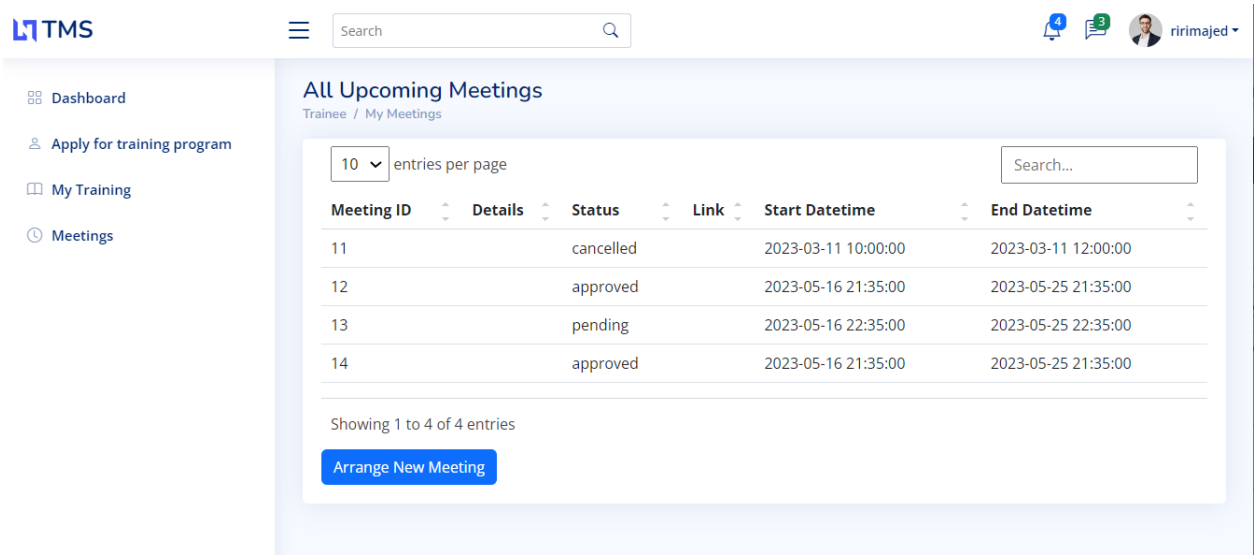
Trainee's Training



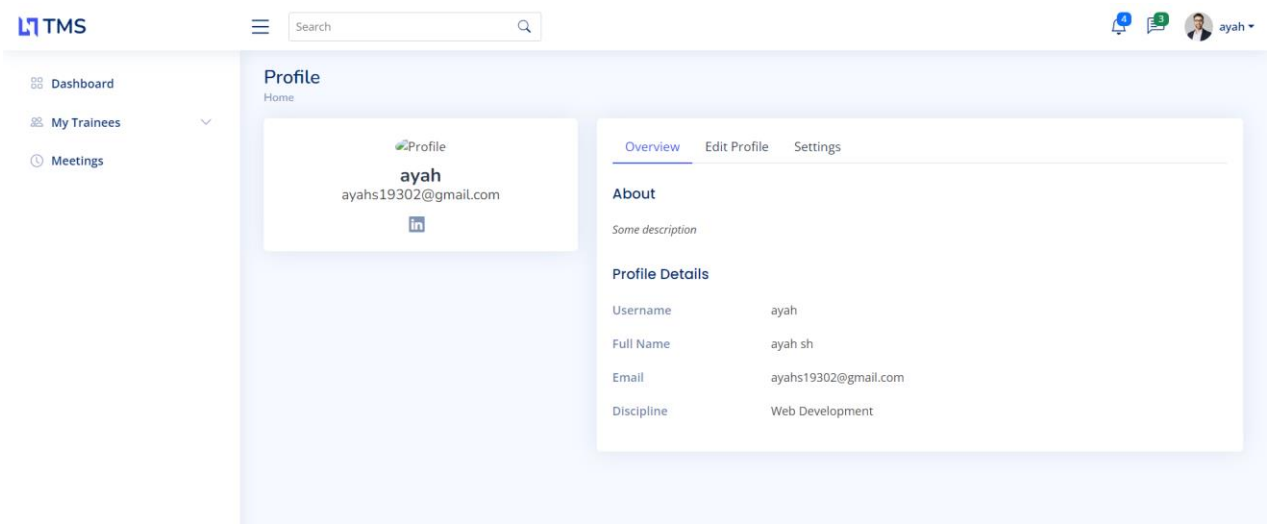
Trainee's Attendance Form



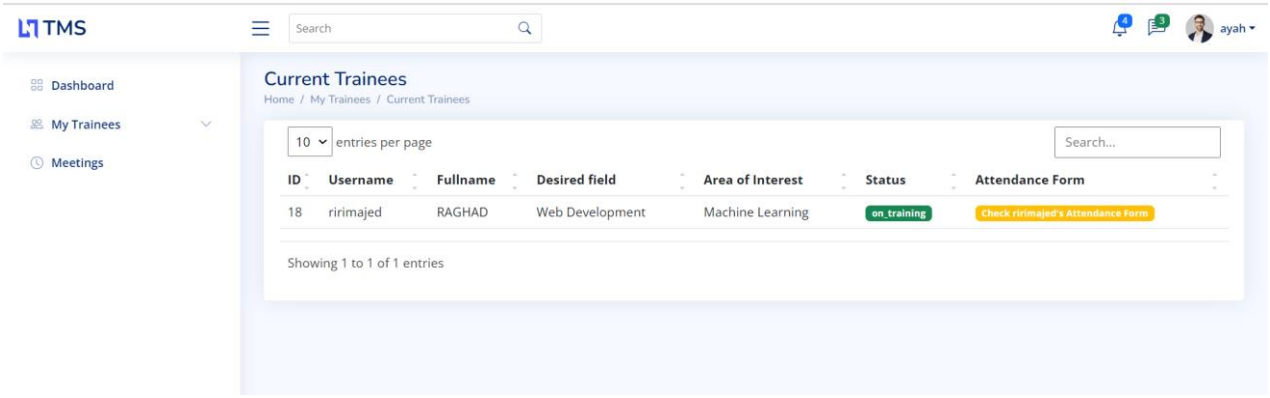
Training Programs Trainees Can Apply To



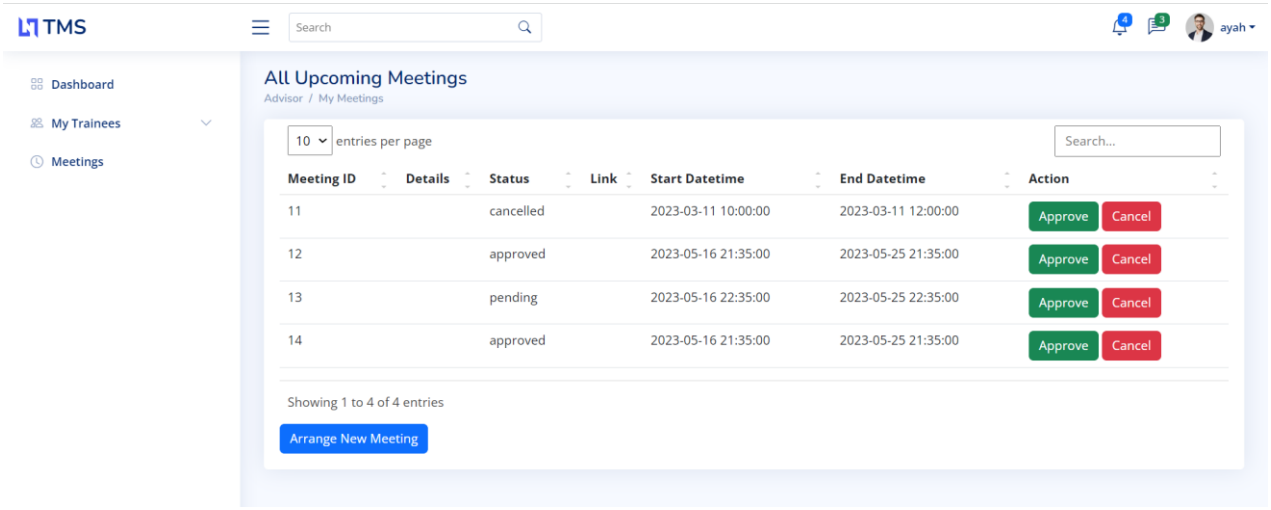
Trainees Meetings



Advisor Dashboard



Current Advisor's Trainees



Upcoming Advisor Meetings

- **Real-world constraints of AWS**

In AWS, the EC2 instances run by default using the public IPv4 address that is generated by AWS. However, the public IPv4 DNS address provided by AWS doesn't run by default and needs to be configured from the EC2 console, which is rather complicated and risky as it might shut down the instance. This is because the server runs using nginx which is not as common as Apache for instance, so doing all that configuration is not very convenient as developers use SaaS providers so that they don't have to go through the configuration process. We think that configuring the DNS address by default from AWS would make it a lot easier for developers to test their instances using DNS address rather than IPv4 addresses.

4. Used AWS Services

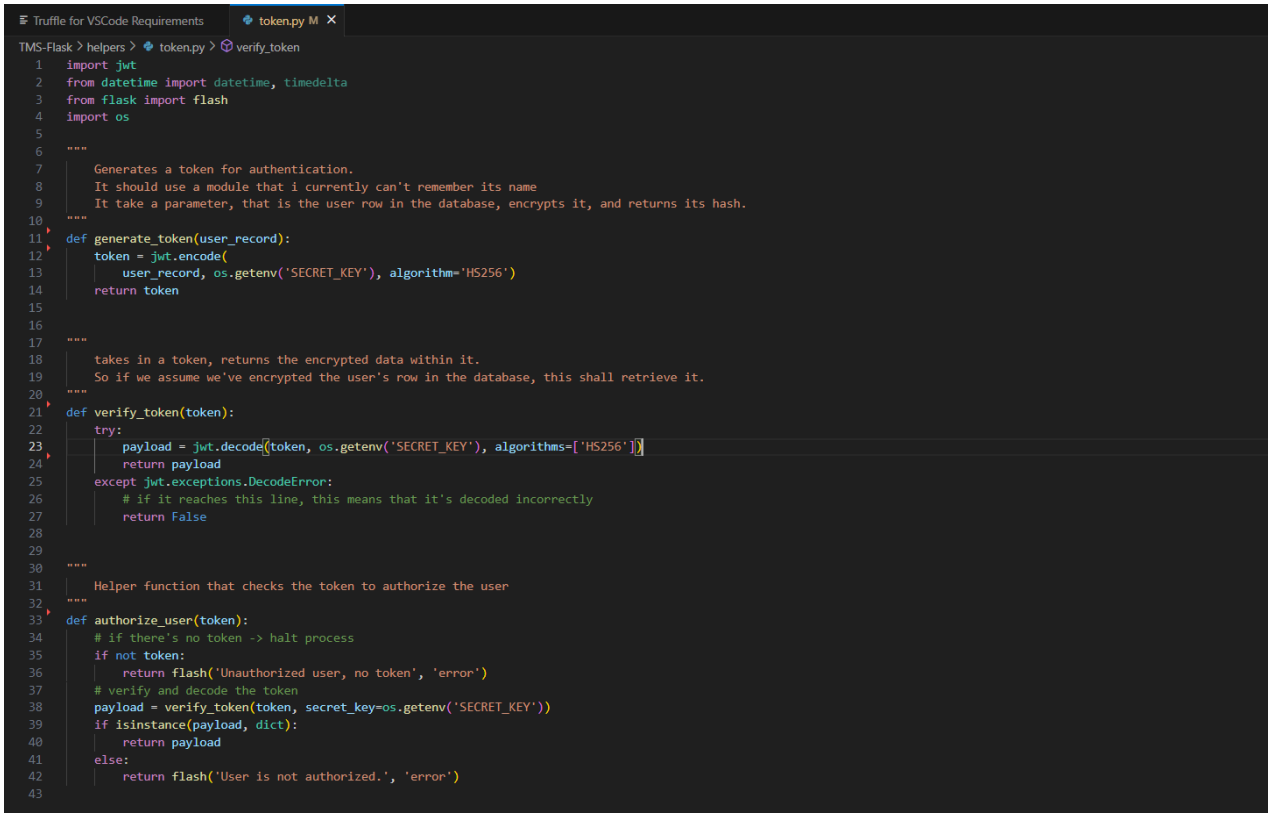
We utilized a range of AWS services to support the implementation and deployment of our system. These services included Amazon Relational Database Service (RDS), Amazon Elastic Compute Cloud (EC2) instances, security groups, Amazon Simple Email Service (SES), AWS Identity and Access Management (IAM), and AWS Lambda Function.

We leveraged the RDS service as our managed database solution, to enable us to store and access relational MYSQL data efficiently and securely. We also employed EC2 instances to host and run our application components, and provide scalable compute resources to handle varying workloads.
To ensure robust security measures, we utilized security groups to define network access rules and control inbound and outbound traffic.

IAM played a vital role in managing user access and permissions, allowing us to define fine-grained access controls and ensure secure authentication in the sending emails process. SES was employed for reliable and scalable email communication, enabling us to send emails to users for password authentication once their registration is approved by the manager. It provided an abstraction for the email delivery functionality using APIs that made the whole process easier. Lastly, we utilized AWS Lambda, a server less compute service, to invoke the send email functions in response to the manager's events. These AWS services collectively provided us with a scalable, secure, and reliable infrastructure foundation to build and deploy our application in the cloud environment.

5. Implementation

In this section, we will preview some of the most important code and the logic behind. Please feel free to check out all the source code of TMS through the provided GitHub link in the references.



Token Generation, Verification & Authorization

Truffle for VSCode Requirements | token.py M | auth_controller.py

TMS-Flask > controller > auth_controller.py > handle_login

```python
def handle_login(request):
    # get information from request
    email = request.form['email']
    password = request.form['password']
    if not email or not password:
        flash('Missing email or password', 'error')
        return redirect(url_for('auth.login_view'))
    # let's look whether we can find those credentials
    query = text("SELECT * from users where email = :email and password = :password")
    params = {"email": email, "password": password}
    result = db.session().execute(query, params)
    row = result.fetchone()
    # return jsonify('querying trainee returned nothing')
    # print(row)
    if not row:
        return jsonify("nothing");
        flash('Email or password are incorrect, please try again', 'error')
        return redirect(url_for('auth.login_view'))
    # NOTE: We need to check the statuses of the trainee and the advisor before authorizing them to the system
    # Which means we'll fetch the trainee and advisor records
    # which also means deleting the trainee, advisor helpers :))))))
    classification = row[3]
    # print(classification)
    if classification == "manager":
        # get the row from the database
        manager_record = db.session.execute(text(
            "SELECT * from managers where userID = :userID"),
            {"userID": row[0]}
        ).fetchone()
        manager = {
            "managerID": manager_record[0],
            "username": manager_record[1],
            "fullname": manager_record[2],
            "email": manager_record[3],
            "userID": manager_record[4],
        }
        # generate token
        token = tokenHelper.generate_token(manager)
        # print("inside manager")
        response = redirect(url_for('manager.dashboard_view'))
        response.set_cookie('token', token)
        return response
    elif classification == "advisor":
        # get the row from the database
        advisor_record = db.session.execute(
            text("SELECT * from advisors where userID = :userID and status in ('active','training')"),
            {"userID": row[0]}
        ).fetchone()
        if not advisor_record:
            # it means the satuts match failed, so the user can not be authorized
            flash("you are not authorized to the system.")
            return redirect(url_for('auth.login_view'))
        advisor = {
            "advisorID": advisor_record[0],
```

Truffle for VSCode Requirements | token.py M | auth_controller.py

TMS-Flask > controller > auth_controller.py > handle_login

```python
            text("SELECT * from advisors where userID = :userID and status in ('active','training')"),
            {"userID": row[0]}
        ).fetchone()
        if not advisor_record:
            # it means the satuts match failed, so the user can not be authorized
            flash("you are not authorized to the system.")
            return redirect(url_for('auth.login_view'))
        advisor = {
            "advisorID": advisor_record[0],
            "username": advisor_record[1],
            "fullname": advisor_record[2],
            "email": advisor_record[3],
            "discipline": advisor_record[4],
            "status": advisor_record[5],
            "userID": advisor_record[6],
        }
        # generate token
        token = tokenHelper.generate_token(advisor)
        # print("inside manager")
        response = redirect(url_for('advisor.dashboard_view'))
        response.set_cookie('token', token)
        return response
    elif classification == "trainee":
        # get the row from the database
        trainee_record = db.session.execute(
            text("SELECT * from trainees where userID = :userID and status in ('active','on_training')"),
            {"userID": row[0]}
        ).fetchone()
        if not trainee_record:
            # it means the satuts match failed, so the user can not be authorized
            flash("you are not authorized to the system yet, wait for admin approval")
            return redirect(url_for('auth.login_view'))

        trainee = {
            "traineeID": trainee_record[0],
            "username": trainee_record[1],
            "fullname": trainee_record[2],
            "email": trainee_record[3],
            "desired_field": trainee_record[4],
            "area_of_training": trainee_record[5],
            "status": trainee_record[6],
            "balance": trainee_record[7],
            "training_materials": trainee_record[8],
            "userID": trainee_record[9],
        }
        # generate token
        token = tokenHelper.generate_token(trainee)
        # print("inside trainee")
        response = redirect(url_for('trainee.dashboard_view'))
        response.set_cookie('token', token)
        return response
    else:
```

The Login Implementation

Truffle for VSCode Requirements | logger.py

TMS-Flask > helpers > logger.py > ...

```python
import logging


class Logger:
    _instance = None

    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = super(Logger, cls).__new__(cls, *args, **kwargs)
        return cls._instance

    def __init__(self):
        self.logger = logging.getLogger('my_logger')
        self.logger.setLevel(logging.DEBUG)

        # Create a file handler and set the log level
        file_handler = logging.FileHandler('app.log')
        file_handler.setLevel(logging.DEBUG)

        # Create a log formatter and add it to the file handler
        formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')
        file_handler.setFormatter(formatter)

        # Add the file handler to the logger
        self.logger.addHandler(file_handler)

    def log(self, message):
```

The Logger Class Used to Help Track User's Activities

controller > manager > manager_advisor_controller.py > ...

```python
from flask import request, render_template, jsonify, flash, url_for, redirect
from sqlalchemy import text
from app import db
import secrets
import helpers.manager_helper as mghelper
import helpers.token as token_helper


"""
    This is the function that prepares data for the 'pending advisors' view, and returns the view with its data
"""
def get_pending_advisors(request):
    # token is the manager id or the manager record
    token = request.cookies['token']
    if not token:
        flash('Token not found, invalid request', 'error')
        return redirect(url_for('auth.login_view'))
    manager = token_helper.verify_token(token)
    # if the query returned nothing -> it actually means we can't render the dashboard
    if not manager:
        flash('Invalid token', 'error')
        return redirect(url_for('auth.login_view'))
    query = text("SELECT * from advisors where status = 'pending'")
    result_cursor = db.session.execute(query)
    rows = result_cursor.fetchall()
    advisors = []
    for row in rows:
        advisors.append(row._data)
    return render_template('manager/advisor/pending_advisors.html', manager=manager, advisors=advisors)

```

Get Pending Advisors

Approving Trainees Registration Request

(The Same Logic is Applied To Advisor's Registration Request)

controller > manager > manager_advisor_controller.py > ...

```python
def reject_advisors_registration(request):
    token = request.cookies['token']
    # make sure manager is authorized
    if not token:
        flash('Token not found, invalid request', 'error')
        return redirect(url_for('auth.login_view'))
    manager = token_helper.verify_token(token)
    # if the query returned nothing -> it actually means we can't render the dashboard
    if not manager:
        flash('Invalid token', 'error')
        return redirect(url_for('auth.login_view'))
    # get hidden form data
    advisorID = request.form['advisorID']
    # update advisor table with userID
    advisor_query= text("UPDATE advisors SET status = 'rejected' WHERE advisorID = :advisorID")
    advisor_cursor = db.session.execute(advisor_query, {'advisorID': advisorID})
    advisor_rows = advisor_cursor.rowcount
    # commit changes to db
    db.session.commit()
    if not advisor_rows:
        flash('Failed to approve advisor', 'error')
        return redirect(url_for('manager.get_pending_advisors_view'))
    flash('Advisor rejected successfully', 'success')
    return redirect(url_for('manager.get_pending_advisors_view'))

```

Reject Pending Advisors

controller > manager > manager_advisor_controller.py > ...

```python
def accept_advisor_modifications(request):
    token = request.cookies['token']
    # make sure manager is authorized
    if not token:
        flash('Token not found, invalid request', 'error')
        return redirect(url_for('auth.login_view'))
    manager = token_helper.verify_token(token)
    # if the query returned nothing -> it actually means we can't render the dashboard
    if not manager:
        flash('Invalid token', 'error')
        return redirect(url_for('auth.login_view'))
    # get hidden form data
    advisorID = request.form['advisorID']
    # update advisor table with userID
    advisor_query= text("UPDATE advisors SET status = 'active' where advisorID = :advisorID")
    print("^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^")
    print(advisorID[0])
    advisor_cursor = db.session.execute(advisor_query, {'advisorID': advisorID})
    # commit changes to db
    db.session.commit()
    if not advisor_cursor:
        flash('Failed to approve advisor modification request', 'error')
        return redirect(url_for('manager.get_advisors_accounts_view', manager=manager))
    flash('Advisor modifications approved successfully', 'success')
    return redirect(url_for('manager.get_advisors_accounts_view', manager=manager))

```

Approve Advisors Modification Request

```
controller > manager > manager_advisor_controller.py > ...

def reject_advisor_modifications(request):
    token = request.cookies['token']
    # make sure manager is authorized
    if not token:
        flash('Token not found, invalid request', 'error')
        return redirect(url_for('auth.login_view'))
    manager = token_helper.verify_token(token)
    # if the query returned nothing -> it actually means we can't render the dashboard
    if not manager:
        flash('Invalid token', 'error')
        return redirect(url_for('auth.login_view'))
    # get hidden form data
    advisorID = request.form['advisorID']
    # update advisor table with userID
    advisor_query= text("UPDATE advisors SET status = 'active' where advisorID = :advisorID")
    print("^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^")
    print(advisorID[0])
    advisor_cursor = db.session.execute(advisor_query, {'advisorID': advisorID})
    # commit changes to db
    db.session.commit()
    result = advisor_cursor.rowcount
    if not result:
        flash('Failed to approve advisor modification request', 'error')
        return redirect(url_for('manager.get_advisors_accounts_view', manager=manager))
    flash('Advisor modifications rejected successfully', 'success')
    return redirect(url_for('manager.get_advisors_accounts_view', manager=manager))
```

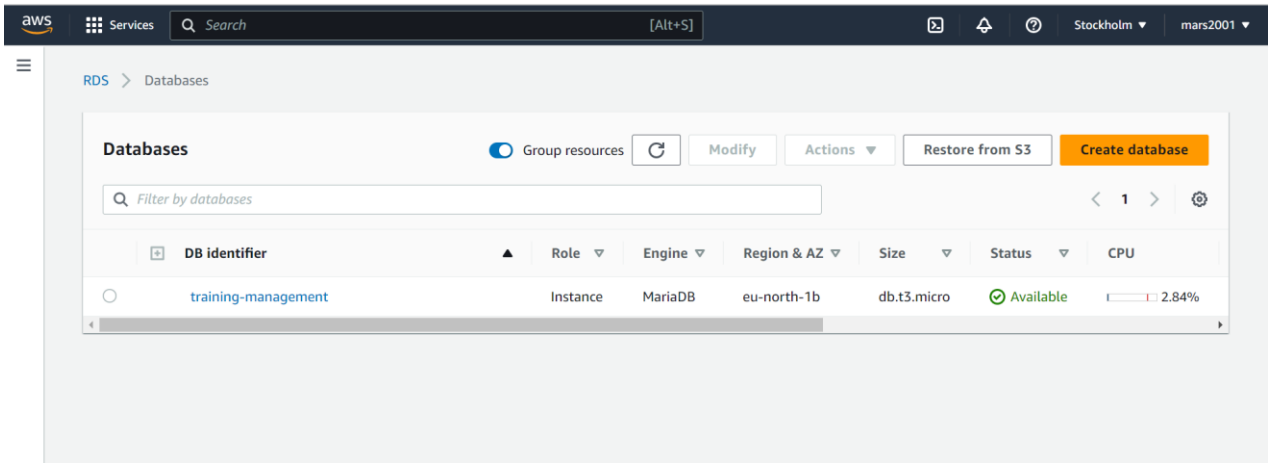
Reject Account Modification Request

The same logic that is implemented in the manager-advisor controller is implemented in the manager-trainee controller.

For further insights on the code, again, you can freely check the GitHub repo provided in the references. There you can see the code and file structure and any details you would like to enquire about. The code is provided with additional line-by-line comments that explain the implementation.

6. Data

The data model of our system is implemented using MySQL, which is hosted on Amazon RDS (Relational Database Service).

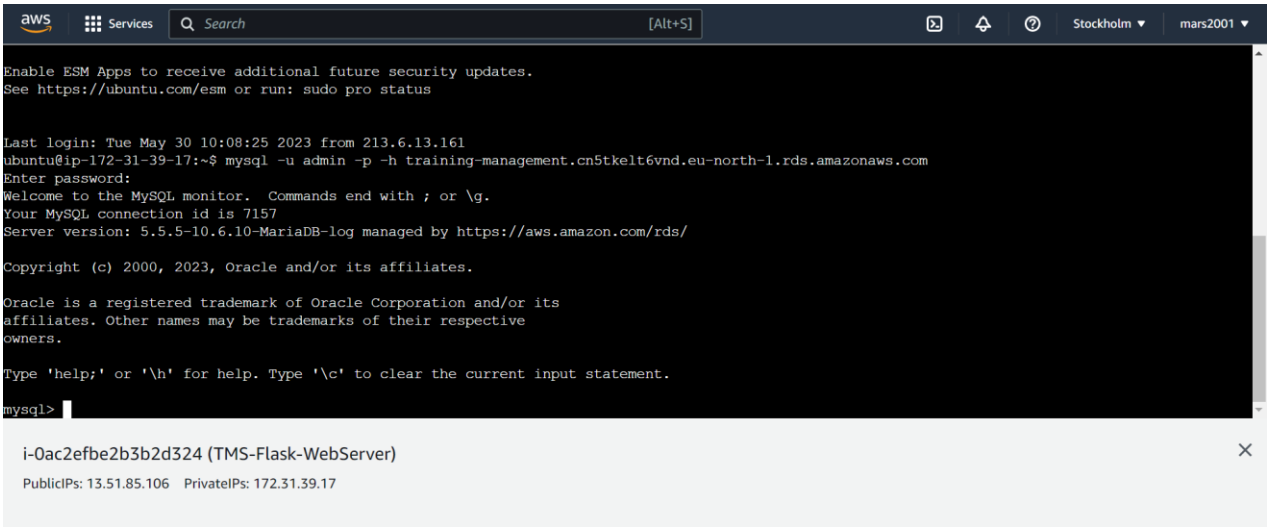


The Database Instance in RDS

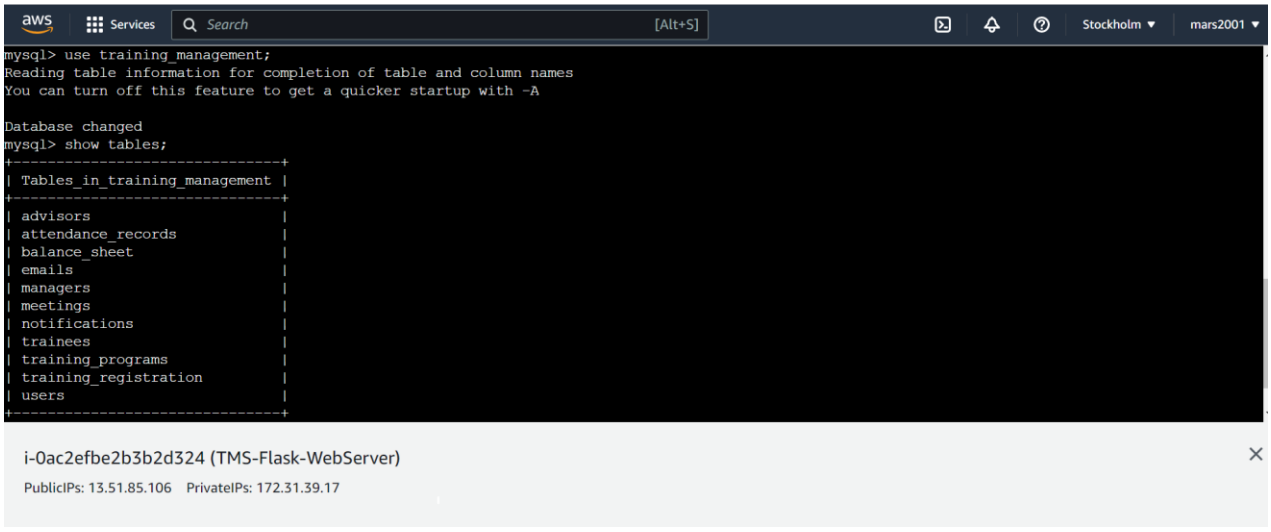
In RDS, the MySQL database is stored and its various entities and their related information are managed. This includes data related to trainees, managers, advisors, and other system-specific entities. The data model consists of table, as shown in the previous figures, that represent different entities, and the relationships between these entities define the structure and organization of the data.

As shown in the previous figures, the data itself is stored within the MySQL database instance provided by Amazon RDS, which is a fully managed database service that simplifies the deployment, management, and scaling of databases in the cloud. It offers features such as automated backups, high availability, and monitoring capabilities.

By utilizing MySQL with Amazon RDS, TMS benefits from a reliable and scalable database solution that ensures data integrity, availability, and performance. The data model is designed to meet the requirements of TMS, allowing efficient storage, retrieval, and manipulation of data to support the functionality and operations of the training management system.

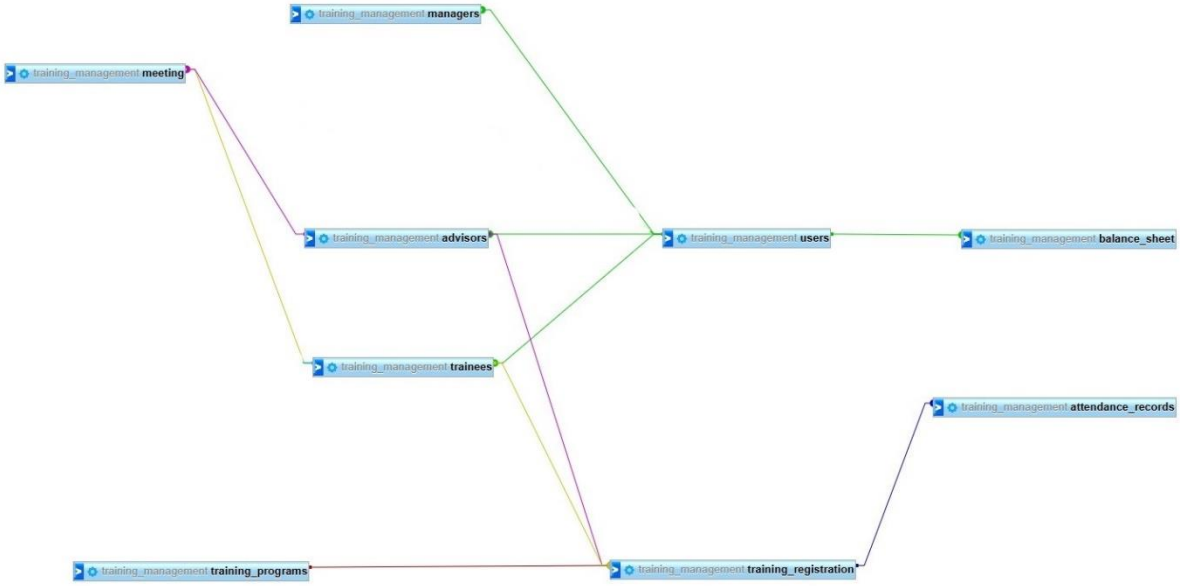


Executing and Connecting RDS using the Connection Endpoint in EC2 Console

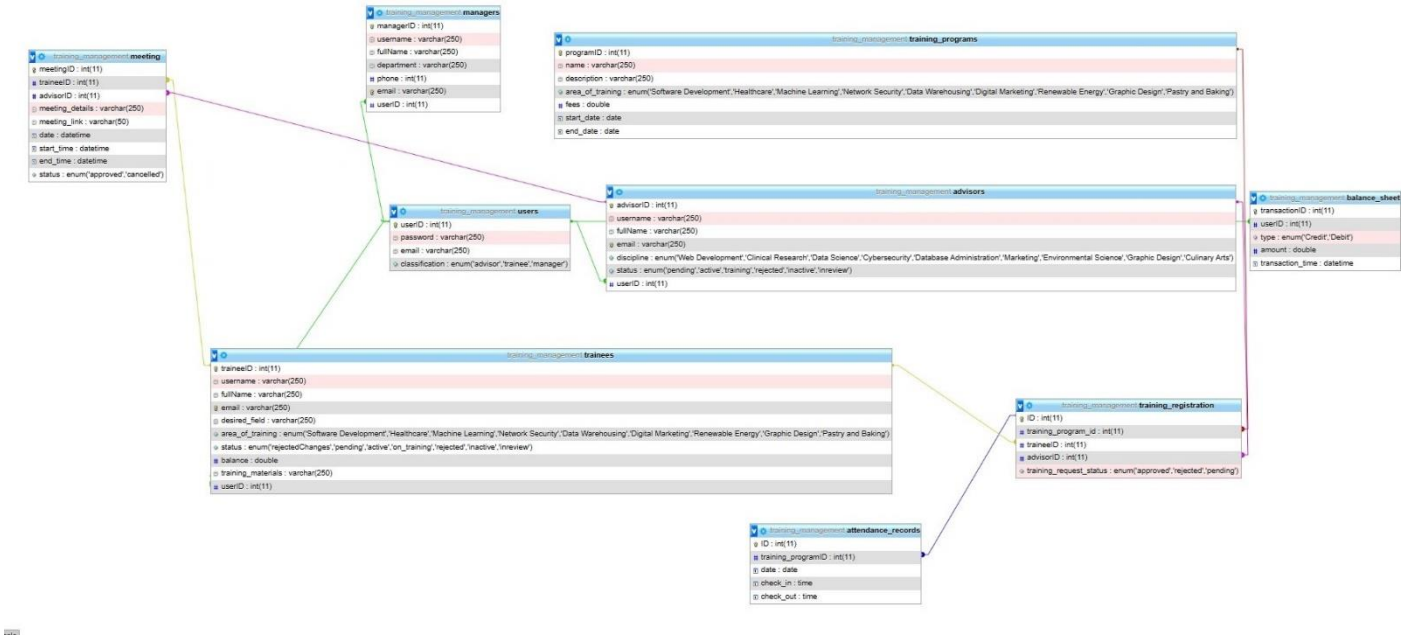


All Tables in the Database

- **The Database Schema**



Database Schema Pt.1



Database Schema Pt.2 (Detailed)

7. The Architecture of AWS

Amazon Web Services (AWS) is the world's most widely adopted and comprehensive cloud, offering more than 200 end-to-end data center services globally. Millions of customers, including the fastest growing startups, largest companies, and leading government agencies, are using AWS to lower costs, increase agility, and innovate faster.

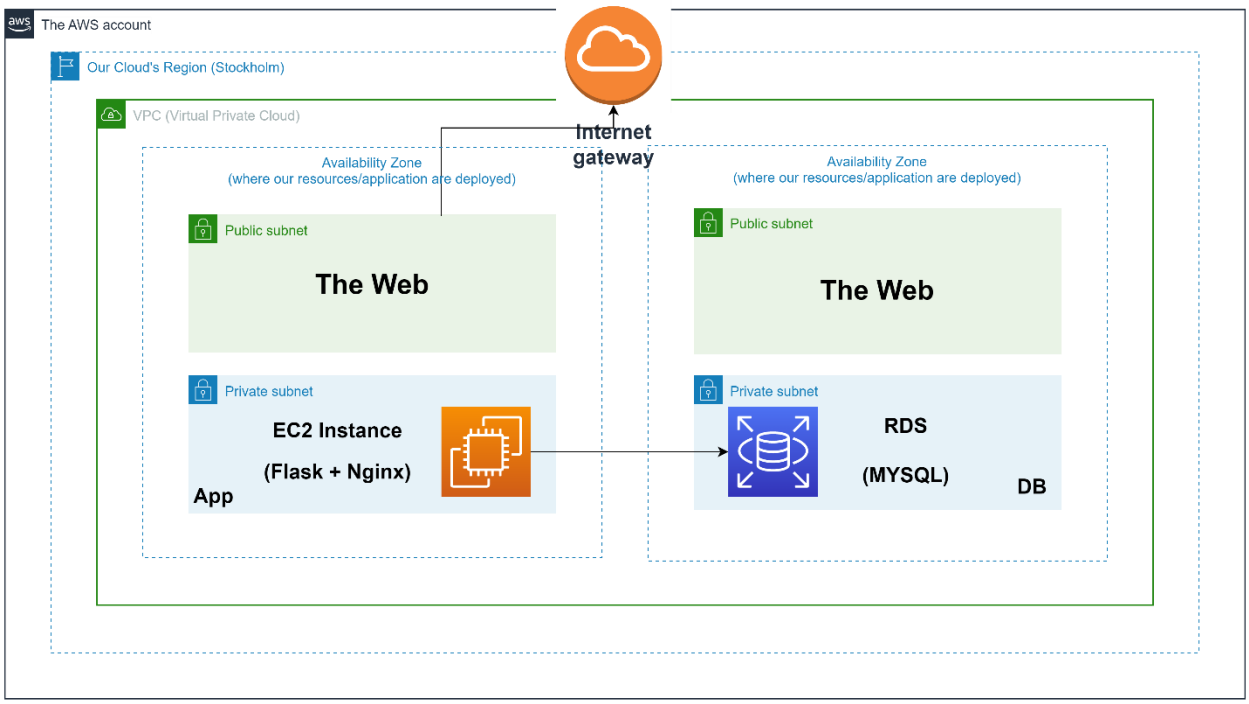


Figure4 - The Architecture of TMS on AWS

AWS is designed to be the most flexible and secure cloud computing environment available today. Our core infrastructure was built to meet the security requirements of the military, international banks, and other organizations that must adhere to strict confidentiality requirements. It is backed by a comprehensive set of cloud security tools, with more than 300 security, compliance, and governance services and features. AWS supports 98 security standards and compliance certifications, and all 117 AWS services that store customer data offer the ability to encrypt that data.

Having said that, the AWS architecture for TMS is designed as follows:

1. EC2 Instances: TMS requires one EC2 instances to host all its needed components. This instance is be deployed in a sperate availability zone than RDS for high availability and fault tolerance.

2. RDS (Relational Database Service): TMS uses RDS to store and manage the MySQL database. The RDS instance is configured with appropriate storage capacity, instance type, and backups to ensure data persistence and reliability.

3. VPC (Virtual Private Cloud): TMS is deployed within a VPC, which provides network isolation and security. It allowed us to define subnets, route tables, and network access control lists (ACLs) to control inbound and outbound traffic.

4. Subnets: TMS utilizes both public and private subnets. Public subnets are associated with internet gateways, allowing external access to TMS's components. Private subnets, on the other hand, are not directly accessible from the internet. Thus, we stored our RDS database and EC2 instance to provide an additional layer of security

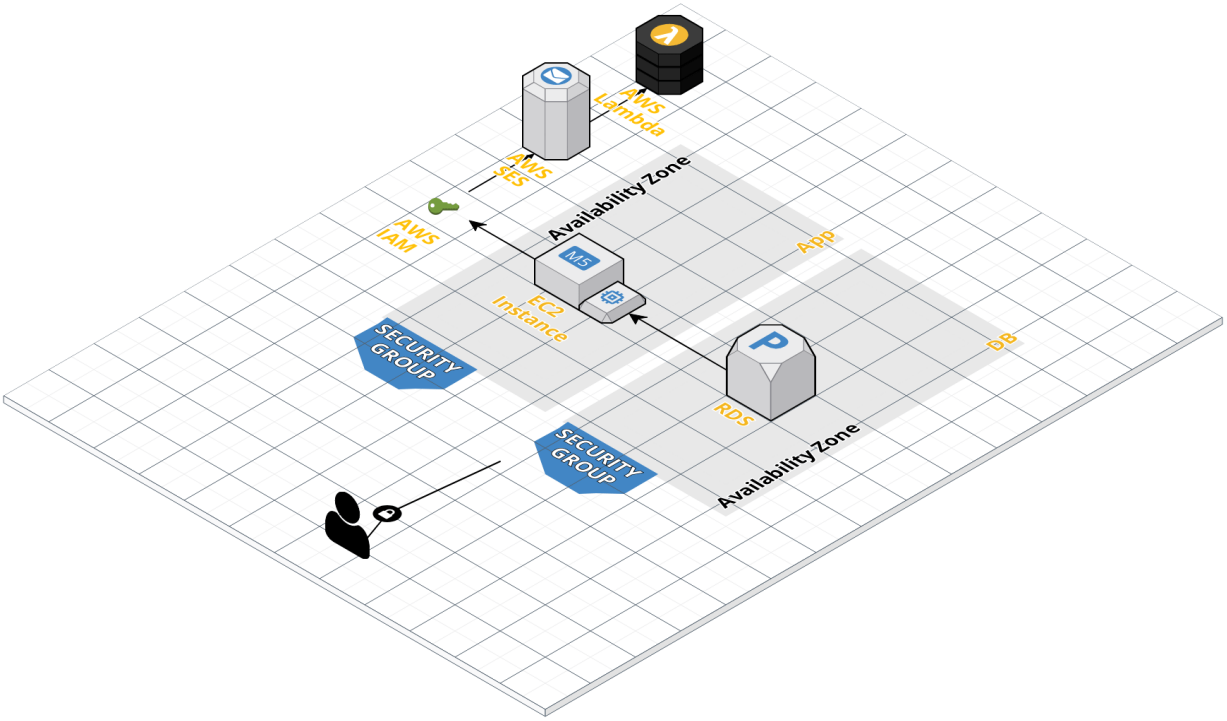
5. Security Groups: we used security groups to control inbound and outbound traffic for the EC2 instances and RDS. They act as virtual firewalls, which allowed us to specify allowed protocols, ports, and source IP ranges.

6. IAM (Identity and Access Management): TMS uses IAM to manage user access and permissions to AWS resources. It allowed us to verify users and define fine-grained access controls for all our user types, the managers, advisors, and trainees.

7. SES (Simple Email Service): SES is utilized for sending email notifications to users, such as trainees and advisors. It provided a reliable and scalable email delivery service.

8. Lambda: AWS Lambda functions were used to perform serverless computing tasks within TMS. We used it to trigger automated email sending actions.

These are some of the key components and services that are a part of the AWS architecture for TMS. The figure below showcases how components are related to each other.



A 3D and More Detailed Representation of TMS's Cloud Architecture

8. Deployment on the Platform

Here are the steps in which we deployed TMS on AWS:

1. Launched the Instance: In the AWS Management Console, we selected EC2 and chose "Launch Instance."

2. Chose Instance Type: we chose the instance type that met our requirements in terms of CPU, memory, storage, and networking capacity.

3. Configured the Instance: we set up additional configuration options, such as the number of instances, network settings, security groups, and storage options.

4. Added Storage: we specified the size and type of storage for our instance.

5. Configured Security Groups: we defined the inbound and outbound rules for the security group associated with our instance. This controled the network traffic that could reach our instance.

6. Review and Launch: Double-check your instance configuration and settings, and then launch the instance.

7. Key Pair Selection: we created a key pair to securely connect to your instance from our local terminal.

8. Launch Status: Once our instance was launched, we received a confirmation where we were able to view the status and details of our instance.

9. User Support

User service in TMS is more than just the time in which trainee spends interacting with the provider. The end-to-end user experience includes many touchpoints and interactions between the trainee and the TMS application. Each of these interactions provide an opportunity for the TMS application to influence the user's perception about the quality that they are receiving.

1- Trainees will have the comfort in using this application: Trainees can apply for training and can select one or more training programs based on their desired field and area of training, they can fill-in training attendance forms. Also request for a meeting with his advisor. In requesting a meeting, TMS must resolve any conflicts.

2- Manager is the person who has a training background: They can review training requests and decide whether to accept them or not based on a given criteria, the manager registers the trainee as an authorized user and sends a unique trainee ID for future login, Whenever the trainee wants to login again, the manager first verifies the entered trainee ID.

3- Easy way to update the data in cloud: The trainee or the advisor cannot update or perform any modification on the data stored in the cloud. Only the manager has the rights to update the cloud's data to ensure data security.

4- No need to worry about the data management: Cloud application stores trainees' records, performs computation and any needed functionality such as

trainee-advisor appointment management, notifications, emailing, searching, advertising such as pop-up general medical advices.

5- Advisor has the right to choose: An advisor manages his own trainees, e.g., accepts and schedules meetings with his trainees, follows up his trainees. He also can see a list of his trainees such as new trainees, on training trainees, etc. Also, he can send notifications or emails to his trainees as needed.

10. Conclusion

TMS is a comprehensive platform for trainees, advisors, and managers to efficiently manage training programs. With features such as registration, document upload, training selection, meeting scheduling, and emailing, the system offers a seamless user experience. Leveraging AWS services like EC2 instances, RDS, IAM, and SES, the system ensures scalability, security, and reliable data storage. Its three-tier architecture enables efficient data management, analytics, and tracking of user activities. Overall, TMS streamlines training processes, enhances collaboration, and delivers a user-friendly interface for effective training management.

References/Links

The GitHub Repo for TMS

http://www.tmstrainninghub.click/login