Trello Fellows

Iteration 3 Summary Document

Table of Contents

Trello Fellows

Cardio B
Vision (Small Project)

Version <1.3>

Revision History

Date	Version	Description	Author
02/04/2025	1.0	Initial information	Kiera Shepperd
02/06/2025	1.1	Furthering information	Emily Wokoek Lawson Hale
02/09/2025	1.2	Finished preliminary input of information	Kiera Shepperd
2/20/2025	1.3	Formatted for changes to design	Kiera Shepperd Emily Wokoek

Table of Contents

Vision (Small Project)

1. Introduction

The purpose of this document is to collect, analyze, and define high-level needs and features of the CardioB app. It focuses on the capabilities needed by the stakeholders and the target users, and why these needs exist. The details of how the CardioB app fulfills these needs are detailed in the use-case and supplementary specifications.

1.1 References

PDF-Use Cases

Project Deliverable 2

2. Positioning

2.1 Problem Statement

The problem of	Procrastination in health
affects	People around the world
the impact of which is	Increased poor health
a successful solution would be	An app that allows people to track their health and provide encouragement and accountability

The problem of procrastination in health affects many people around the world, the impact of which is increased poor health and obesity. A successful solution would be an app that allows people to track their health and provide encouragement and accountability.

2.2 Product Position Statement

For	Adolescents and adults
Who	Want to track their health
The (product name)	Cardio B
That	Tracks health and provides plans to provide accountability
Unlike	No other health app on the market
Our product	Tracks and monitors health

For adolescents and adults who want to track their health. CardioB is a health app that tracks health and lifestyle activities and provides plans to encourage users. Unlike Strava, our product tracks and monitors health.

3. Stakeholder and User Descriptions

We at CardioB are dedicated to providing the best, most high-quality services to our customers. We exist to provide users with a multifaceted app that allows for logging health information and attending workout classes. In addition, we are committed to keeping user feedback in consideration as we continue to develop our application.

3.1 Stakeholder Summary

Name	Description	Responsibilities
Professor	Professor Kirk wants us to make a health app. He will personally use the greatest health app of all the competition.	Ensures that the system will be maintainable Monitors the project's progress Provides feedback after each iteration

3.2 User Summary

Name	Description	Responsibilities	Stakeholder
User	Someone who wants to gain control of their health and participate in workout classes while tracking their health.	Logs information Registers for and attends classes Registers and logs completion of self-paced classes	The Professor is responsible for representing the user's interest.
Trainer	Uses the app to create self-paced plans and manage classes	Creates, manages, and hosts classes Creates and manages self-paced plans	The Professor is responsible for representing the trainer's interest.
Admin	Assist users with problems or inquiries about the app	Changes passwords Responds to emails	The Professor is responsible for representing the admin's interest.

3.3 User Environment

The target user is generally operating independently but can interact with others, such as joining a trainer's class or joining a self-paced workout. This app is intended to be used in tandem with an internet connection, and will not operate as intended without one. However, the app will still open to see recent stats and upcoming classes without an internet connection. Some system platforms such as MacOS and Windows are in use today. There are no plans for an expansion to include Linux. Other applications can be in use while the health app is running, but a user does not need a secondary app for the health app to function as intended.

3.4 Summary of Key Stakeholder or User Needs

Need	Priority	Concerns	Current Solution	Proposed Solutions
Postponing classes	Medium	Changing the class listing, why to change the time	Object class for classes that both the trainer and the user interact with	Change the time if the trainer does not start the class before the class time begins.

Notifying users of class changes	Low	Notification system may be dependent upon whether or not the user is logged into the system	Keep the notification saved in the user's profile until they log in	Send an email to alert the user when a class that they are registered for changes the time
Reporting logged information	High	What kind of graph should be created to accurately display the information	Graph of caloric intake, water intake, and the amount of time worked out within the last month	Graph all inputted information
Viewing group progress in self-paced classes	Medium	What information should be shared across the group	Display the completion of days of other users	Display the completion of days and the time spent of other users

3.5 Alternatives and Competition

Strava and the IOS Health app are competitors to our application. However, Strava is more geared towards outdoor fitness, specifically cycling, hiking, and running. The IOS Health app does not allow for a connection between trainer and user, relying on preset exercises and generic information. As a small business, we are able to create more personal connections with the users of our app while not attempting to widen the profit margin.

4. Product Overview

4.1 Product Perspective

This product is independent and totally self-contained.

4.2 Assumptions and Dependencies

Assumptions

- User/Trainer/Admin has an internet connection
- User/Trainer/Admin has a valid email address
- User/Trainer/Admin has device that supports the application

5. Product Features

Features for the Users:

Personal Goals

- Users can create daily, weekly, or monthly personal goals for weight, fitness, and diet

Data

- Users can view a record of past data recorded
- Users can document health data

Class Registration

- Users can sign up for classes created by trainers
- Users can report data to their trainer
- The user can leave a class

Goal Tracking

- Users can view if they are on track towards goals
- Users can set custom goals for themselves

Reminders

- Users can create and set reminders for daily tasks

Features for the Trainers:

Exercise Plans

- Trainers can create exercise plans for users

Scheduled Classes

- Trainers can create scheduled classes that include date, time, and days of the week the class occurs, length of class, recommended fitness level, and required equipment
- Trainers can limit the number of participants in the class

Class Data

- Trainers can view information about their plans and classes

Features for the Admin:

Change Password

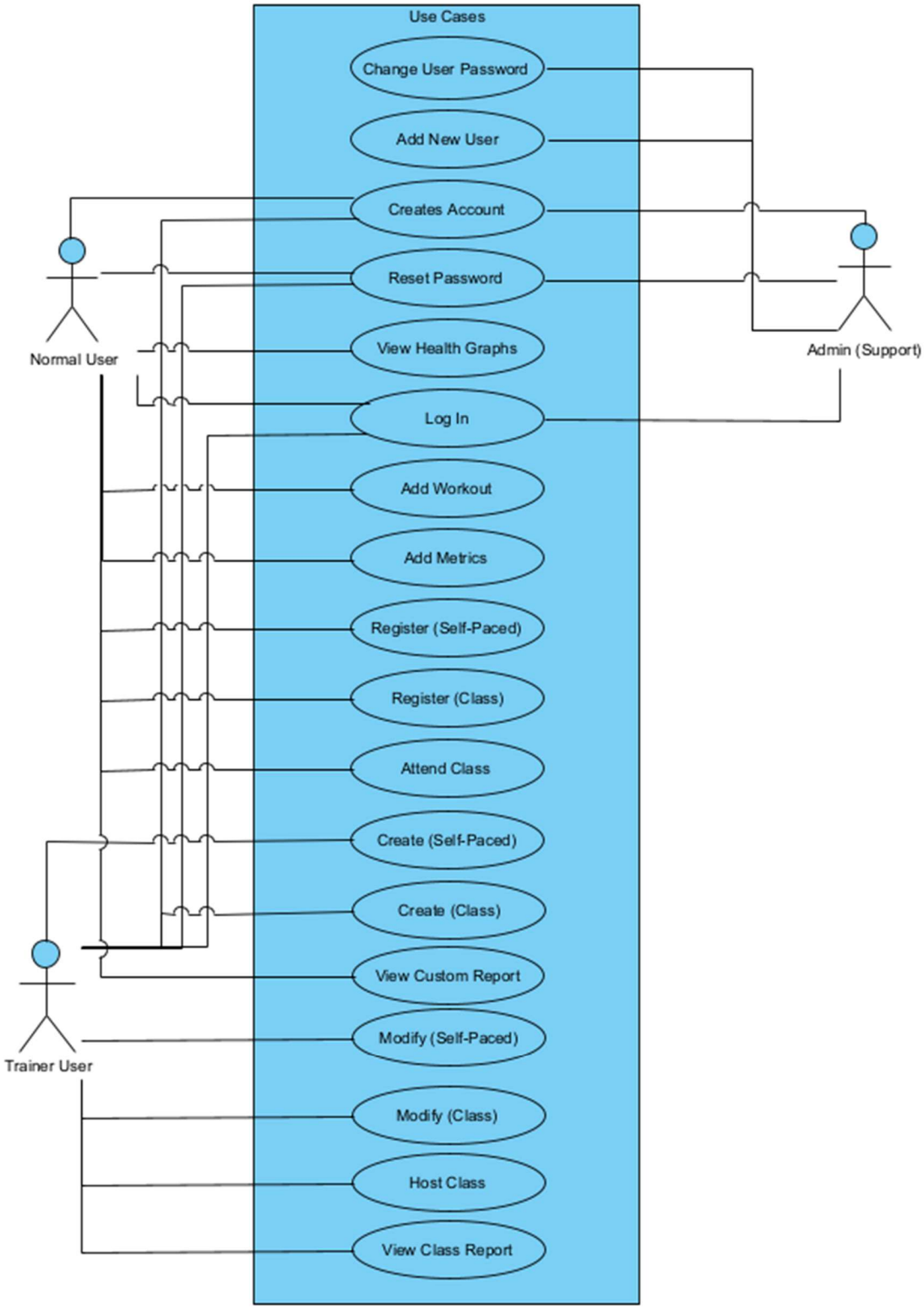
- The Admin can reset a user's password
- The Admin can reset a trainer's password

Management

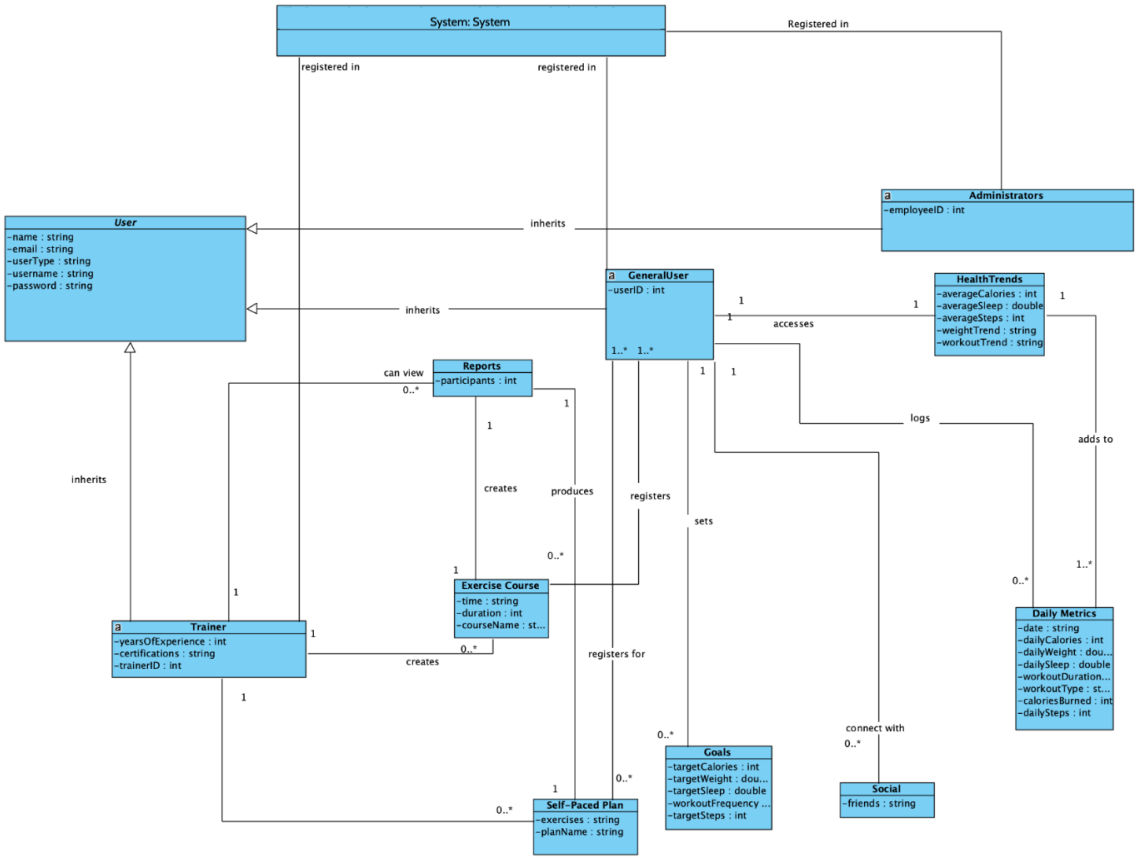
- The Admin can access the information of user accounts
- The Admin can access the information of trainer accounts

6. Other Product Requirements

- Operating System: MacOS or Windows
- Thorough error messaging
- Online informational web page with helpful tips
- Standard industrial speed for all operations
- Fully usable web app


Use Cases
Change User Password
Add New User
Creates Account
Reset Password
View Health Graphs
Log In
Add Workout
Add Metrics
Register (Self-Paced)
Register (Class)
Attend Class
Create (Self-Paced)
Create (Class)
View Custom Report
Modify (Self-Paced)
Modify (Class)
Host Class
View Class Report
Normal User
Trainer User
Admin (Support)

Domain Model



#	Name of Use Case	ID	Author
01	User wants to create account	UC-U.ACCT-01	Noah Mathew
02	User wants to reset password	UC-U.ACCT-02	Noah Mathew
03	Log-in	UC-U.ACCT-03	Josh Fulton
04	Create Trainer Account	UC-T.ACCT-01	Carter Lewis
05	Admin makes account	UC-A.ACCT-01	Josh Fulton
06	Adds new user	UC-A.ACCT-02	Josh Fulton
07	Change user password	UC-A.ACCT-03	Josh Fulton
08	Add workout (duration, type of exercise, calories burned)	UC-TRACK-01	Emily Wokeok
09	Add metrics (sleep, weight, calories burned)	UC-TRACK-02	Kiera Shepperd
10	Register for self-paced exercise plan	UC-U.CLASS-01	Lawson Hale
11	Register for class	UC-U.CLASS-02	Lawson Hale
12	Attend class	UC-U.CLASS-03	Lawson Hale
13	Create self-paced exercise plan	UC-T.CLASS-01	Emily Wokeok
14	Create class	UC-T.CLASS-02	Carter Lewis
15	Modify existing self-paced exercise plan	UC-T.CLASS-03	Emily Wokeok
16	Modify existing class	UC-T.CLASS-04	Carter Lewis
17	Host class	UC-T.CLASS-05	Kiera Shepperd
18	View self-report	UC-REPORT-01	Noah Mathew
19	View class report	UC-REPORT-02	Kiera Shepperd

Author: Noah Mathew
ID: UC-U.ACCT-01
Name: Create User Account
Scope: App Account Creation System
Level: User Goal
Primary Actor: User

Stakeholders and interests:

- Customer
    - User of the health app who is interested in tracking their health data, joining classes, and setting goals

Precondition:

- The user has downloaded the application
- The user has an internet connection
- The user has a valid email address

Postcondition:

- A new username and password is stored in the system
- System stores their preferences/personal health information that was filled out during account setup
- The user is able to login to their account again using the same credentials
- The user is able to reset their account password with the linked email address

Minimal Guarantee:

- The same amount of users is still stored in the system as before
- No corrupted or half filled out data is stored in the system, user must finish set up to create an account
- The system should provide an error message explaining what went wrong

Main success scenario:

1. The app should display a login page, with the option to create an account
2. After clicking the create account button, app will display fields for an email and password
3. The user enters information into the prompted fields
4. The system should evaluate whether the email is valid
5. The system should evaluate whether the password meets security requirements

6. The system should check whether the email is already in the database
7. If everything looks good, the system should send an email to the user's email address to verify it is correct
8. The user is able to click the verification email and be led back to the app
9. The user only then be able to login with that email address/username entered
10. Once validated login, the user fills out personal health information and other data or goals
11. The system is able to save the data the user fills out
12. The user is able to log out and log back into the same or different account

Alternate paths:

3a. The user doesn't fill in all the fields during the account creation process
   i. The system will prompt the user to enter an email and password

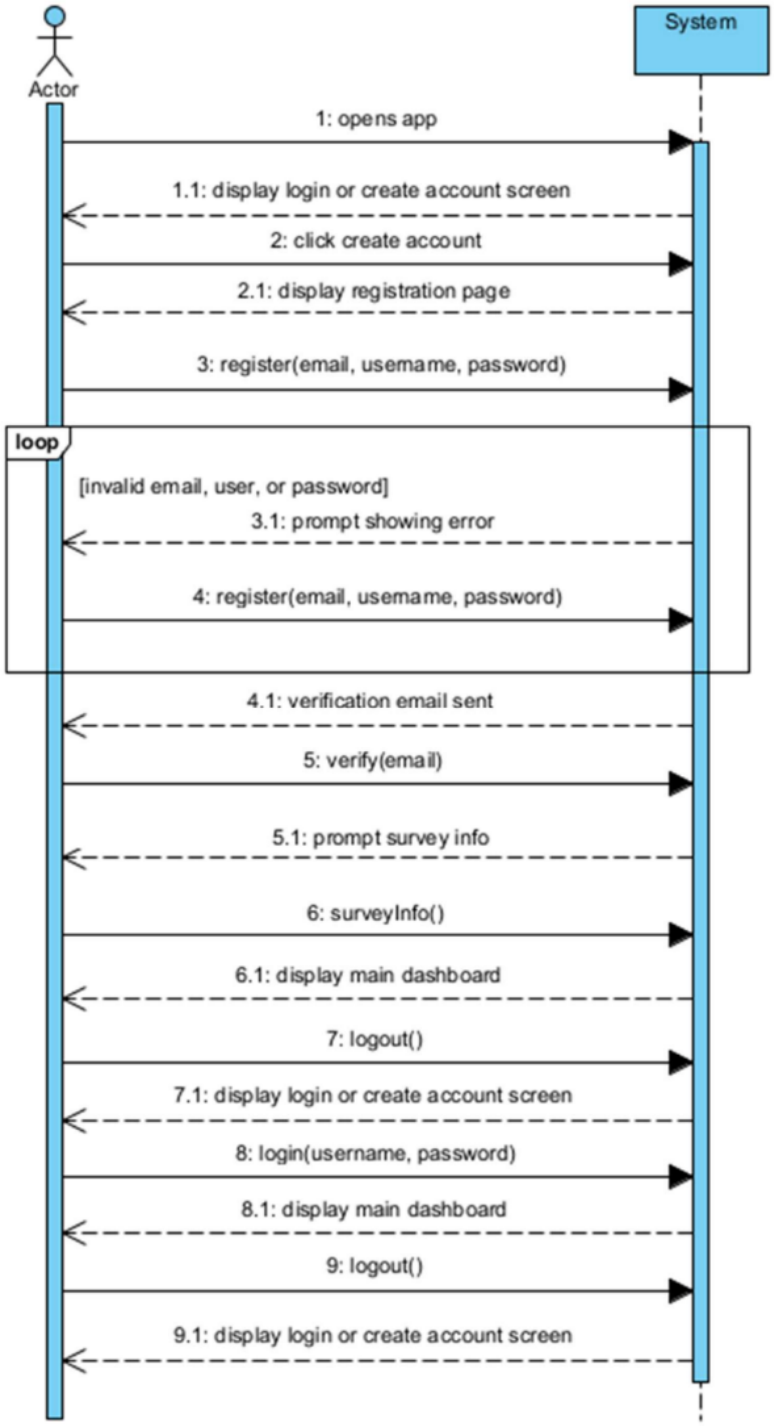
4a. Invalid email format is inputted
   i. The system will prompt the user to re-enter the email address

4b. The email entered is already in the system
   ii. The system will prompt the log-in scree

5a. The password entered isn't secure
   i. The system will prompt the user to re-enter a password that meets the security requirements, which will also be listed

8a. The user does not receive a verification email
   i. The user is able to request another be sent
   ii. The user can create a new account

10a. The user enters no data during the survey process
   i. The user can fill out the information later


Actor
System
1: opens app
1.1: display login or create account screen
2: click create account
2.1: display registration page
3: register(email, username, password)
loop
[invalid email, user, or password]
3.1: prompt showing error
4: register(email, username, password)
4.1: verification email sent
5: verify(email)
5.1: prompt survey info
6: surveyInfo()
6.1: display main dashboard
7: logout()
7.1: display login or create account screen
8: login(username, password)
8.1: display main dashboard
9: logout()
9.1: display login or create account screen

Operation: register(email, username, password)

Cross References

- Use Case: User Account Creation

Preconditions:

- none

Postconditions

- User instance *user* was created (object creation)
- *user.username* was set to username (attribute modified)
- *user.email* was set to email (attribute modified)
- *user.password* was set to password (attribute modified)
- *user.emailVerified* was set to false (attribute modified)

Exceptions

- None

Operation: verify(email)

Cross References

- Use Case: User Account Creation

Preconditions:

- none

Postconditions

- *user.emailVerified* was set to true (attribute modified)

Exceptions

- None

Operation: surveyInfo()

Cross References

- Use Case: User Account Creation

Preconditions:

- A User exists

Postconditions

- The User's profile was updated with survey information (attributes modified)

Exceptions

- None

Operation: logout()

Cross References

- Use Case: User Account Creation

Preconditions:

- User is logged in

Postconditions

- The User's session was terminated (association broken)
- *user.loggedIn* was set to false (attribute modified)

Exceptions

- None

Operation: login(username, password)

Cross References

- Use Case: User Account Creation

Preconditions:

- none

Postconditions

- A new session is created and associated with the User
- *user.loggedIn* was set to true (attribute modification)

Exceptions

- None

Author: Noah Mathew
ID: UC-U.ACCT-02
Name: Reset Password
Scope: User Actions
Level: User Goal
Primary Actor: User

Stakeholders and interests:

- User
    - User of the health app who is interested in gaining access to their account and being able to use its features
- System Admin
    - Employees interested in making sure users have support when trying to reset their password and running into issues

Precondition

- The user has an existing account with the app
- The user has a stable internet connection
- The user has access to the email address they registered their account with

Postcondition

- The user's password is reset to a new password they choose successfully
- The user is able to login to their account with the new password
- The system maintains the user's old data with the new password stored
- The user gets an email that their password has been reset

Minimal Guarantee

- The same amount of users is still stored in the system as before
- No corrupted or half filled out data is stored in the system, user must finish set up to create an account
- The system should provide an error message explaining what went wrong

Main success scenario

1. The user opens the app and navigates to the login page
2. The user clicks on Forgot Password button
3. The system prompts the user to provide the email they registered with
4. The system verifies that it is a registered email within the app
5. The system generates a reset password link and email that is sent to the corresponding email address
6. The user opens the email and clicks the reset password link
7. The link brings the user back to the app within a page to enter a new password
8. The user enters a new password that meets security requirements
9. The system updates the password in the database
10. The system emails the user that their password was reset
11. The user returns to the login page and logs in with their new password

Alternate paths

4a. The email address entered on the forgot password page is not associated with an account,

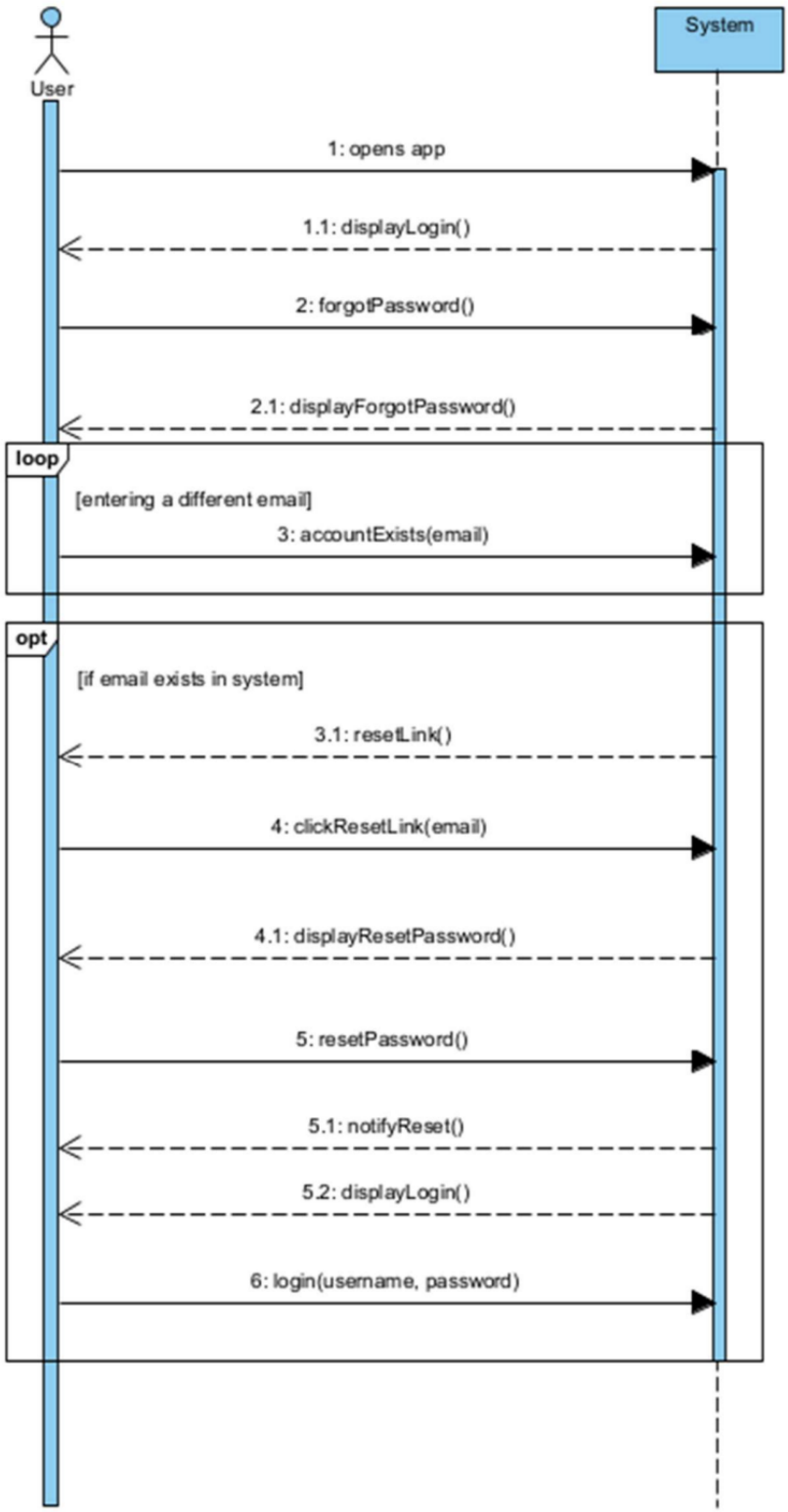
i. the system will re-prompt the user to enter a different email address

6a. The user does not receive the reset password email,

i. they can re-enter their email address

8a. The user's password does not meet security requirements

i. the system will prompt the user with the password requirements


User
System
1: opens app
1.1: displayLogin()
2: forgotPassword()
2.1: displayForgotPassword()
loop
[entering a different email]
3: accountExists(email)
opt
[if email exists in system]
3.1: resetLink()
4: clickResetLink(email)
4.1: displayResetPassword()
5: resetPassword()
5.1: notifyReset()
5.2: displayLogin()
6: login(username, password)

Operation: accountExists(email: String)

Cross References

- Use Case: User Reset Password

Preconditions:

- User provides an email address

Postconditions

- None

Exceptions

- Invalid email format

Operation: sendResetLink(email: String)

Cross References

- Use Case: User Reset Password

Preconditions:

- System has a registered user with that email

Postconditions

- A reset link is created and associated with user
- A reset request timestamp is associated with user

Exceptions

- None

Operation: validateNewPassword(password: String)

Cross References

- Use Case: User Reset Password

Preconditions:

- User has entered a new password

Postconditions

- *user.password* was changed to password (attribute modification)

Exceptions

- None

Operation: notifyReset(email: String)

Cross References

- Use Case: User Reset Password

Preconditions:

- User successfully reset password

Postconditions

- None

Exceptions

- None

Operation: validateLogin(email: String, user: String, password: String)

Cross References

- Use Case: User Reset Password

Preconditions:

- User has successfully reset their password

Postconditions

- A session is created and associated with the User

Exceptions

- None

Author: Josh Fulton
ID: UC-U.ACCT-03
Name: User Log-In
Scope: App Security System
Level: User Goal
Primary Actor: User

Stakeholders and interests:

- User
    - Wants to access account that was created using a username and password

Precondition:

- The account is created

Postcondition:

- The user gains access to the account

Minimal Guarantee:

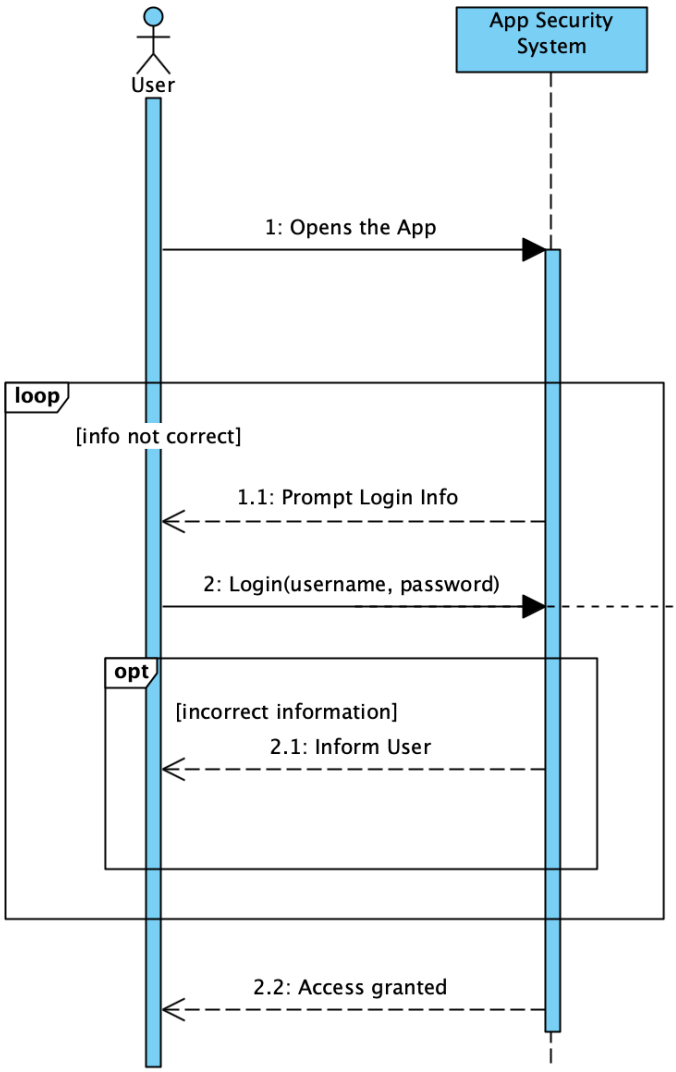
- The account is still existent
- The user can attempt to login after
- The login information for the account has not changed

Main Success Scenario:

1. The app prompts the user to enter a username and password
2. The user enters the correct username and password
3. The system verifies that the information provided is attached to an account and is correct
4. The user gains access to the account
5. The user logs out of the account

Alternate Paths:

2a. The user enters the incorrect information.
    i. The system notifies the user
    ii. The user is able to attempt again
3a. The system finds that the username is correct but the password is not
    i. The system notifies the user of this error
    ii. The user is able to attempt again



Operation: Login(username: string, password: string)

Cross-References:

- User Creates an Account
- Admin Resets Password

Preconditions:

- Account exists
- Username and password are correct

Postconditions:

- A new session is created and associated with the user

Exceptions:

- IncorrectPassword if the password is incorrect

Author: Carter Lewis
ID: UC-T.ACCT-01
Name: Create Trainer Account
Scope: Website System
Level: User Goal
Primary Actor: Trainer

Stakeholders and interests:

- Trainer
    - Create an account with affiliated username and password
    - Have a secure password, which is hashed upon entry (SHA-256)

Precondition

- System is fully functional with the ability to add user profiles and accounts
- Trainer has valid email address
- The trainer has opened the application

Postcondition

- Trainer is logged into his/her account, able to start training routines for users who follow them

Minimal Guarantee

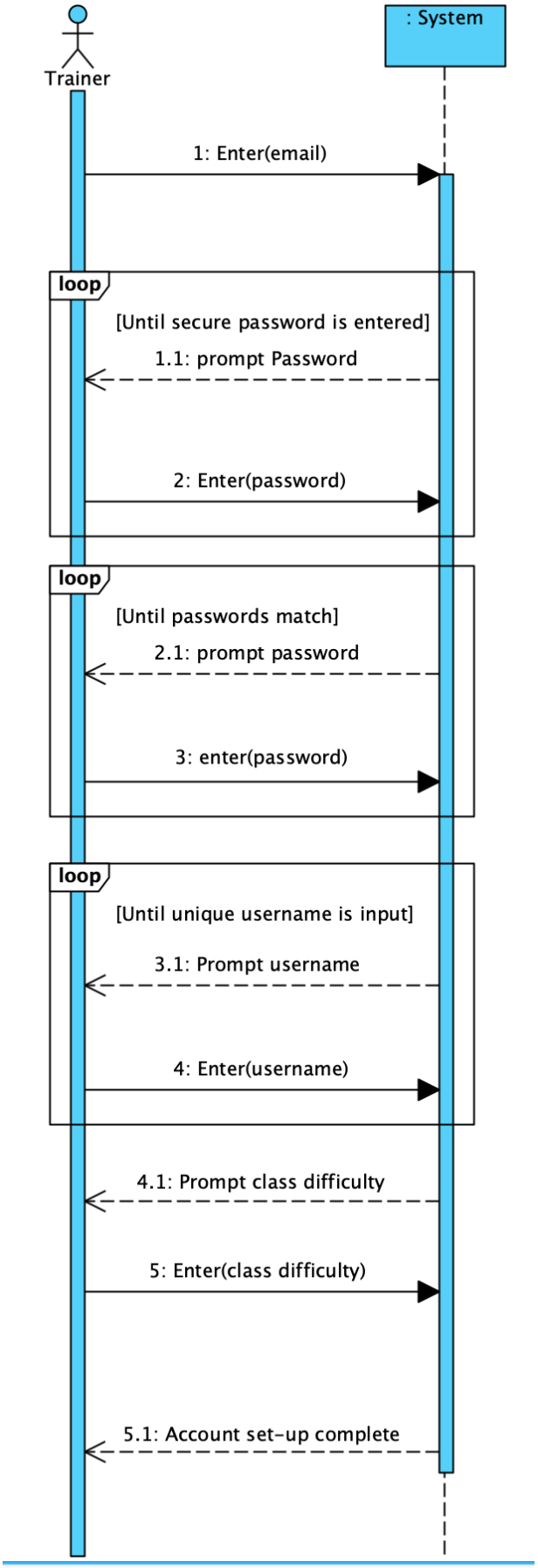
- The security of existing user/trainer accounts will not be compromised

Main success scenario

1. New trainer signs up for account using unique email
2. New trainer sets a password
3. Password is deemed secure by our password-checking algorithm
4. Password is hashed using SHA-256 hashing algorithm
5. Password is stored in database and affiliated with the new user's email
6. User is brought to an "account set-up" page to enter basic details about themself
7. These details are stored in their account
8. Trainer is brought to the home view of the website

Alternate paths

1a. New trainer enters existing email
    i. Trainer is given the option to log in to existing account or use a different email
3a. Password is NOT deemed secure
    ii. Trainer is asked to enter a new password until all security requirements are met


Trainer
: System
1: Enter(email)
loop
[Until secure password is entered]
1.1: prompt Password
2: Enter(password)
loop
[Until passwords match]
2.1: prompt password
3: enter(password)
loop
[Until unique username is input]
3.1: Prompt username
4: Enter(username)
4.1: Prompt class difficulty
5: Enter(class difficulty)
5.1: Account set-up complete

Operation: Enter(email)

Cross References

- Use Case: Trainer Account Creation

Preconditions:

- none

Postconditions

- new trainer instance *trainer* was created (instane created)
- *trainer.email* was set to valid email (attribute modification)

Exceptions

- None

Operation: Enter(Password)

Cross References

- Use Case: Trainer Account Creation

Preconditions

- Valid email is in the system

Postconditions

- *trainer.password* was set to hashed password (attribue modification)

Exceptions

- none

Operation: Enter(Username)

Cross References

- Use Case: Trainer Account Creation

Preconditions

- Valid password is saved in the system

Postconditions

- *trainer.username* was set to hashed password (attribue modification)

Exception

- None

Operation: Enter(Class difficulty)

Cross References

- Use Case: Trainer Account Creation

Preconditions

- Valid email, password, and username entered into the systems

Postconditions:

- *trainer.classDiff* was set to class difficulty (attribute modification)

Exception

- None

Author: Josh Fulton
ID: UC-A.ACCT-01
Name: Create and Set Up an Admin Account
Scope: App Account Creation System
Level: Admin Goal
Primary Actor: Admin

Stakeholders and interests:

- Admin
    - Admin wants to create an account

Precondition:

- The admin has downloaded the application.
- The admin has an internet connection.
- The admin has an email that is part of the list of valid emails that are able to create admin accounts.

Postcondition:

- The admin's account is created.
- The admin is able to view all general user and trainer accounts from the admin's account.

Minimal Guarantee

- The accounts previously created remain within the system.
- The admin can attempt to create an account again

Main Success Scenario:

1. The admin selects create an account.
2. The admin is shown three account types: general user, trainer, and admin.
3. The admin selects "admin".
4. The system provides fields in which the admin must provide information (username, email address, and password).
5. The system checks to see if the account can be an admin account my validating email with a register of emails that are acceptable.
6. The system verifies that the individual can be an admin.
7. The system creates the admin account.
8. The admin is able to login using the information they provided.
9. The admin logs out and exits the application.

Alternate Paths:

2a. The admin selects the wrong type of account.

i. They back out of the page they are on and reselect correct option

5a. The admin provides an invalid email. T

i. The system notifies the admin and requests to enter a different email.

8a. The admin misremembers their password.

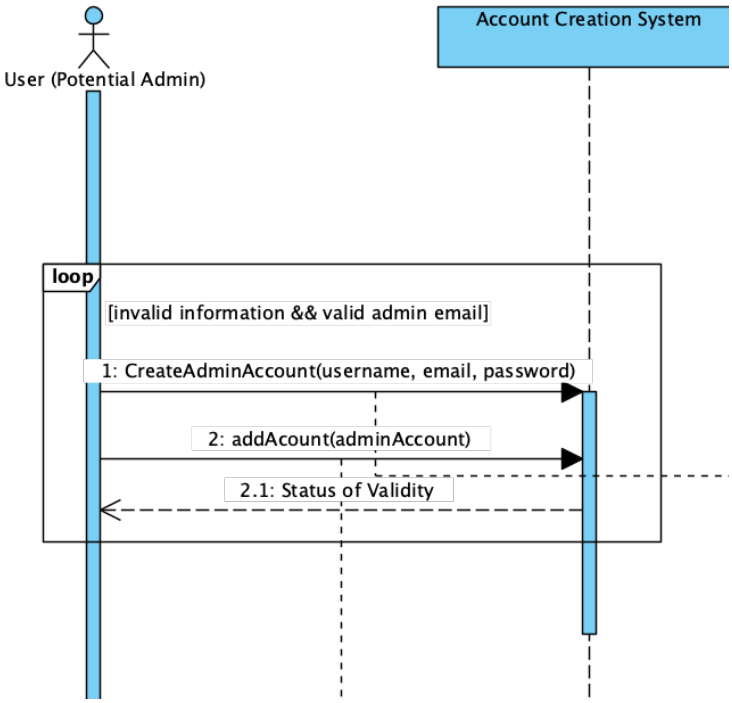
i. They request to reset their password.

ii. After it is reset, the admin logs in

8b. The admin mistypes their email.

i. They are notified that the username is incorrect

ii. The admin is prompted with the log-in screen again


Account Creation System
User (Potential Admin)
loop
[invalid information && valid admin email]
1: CreateAdminAccount(username, email, password)
2: addAcount(adminAccount)
2.1: Status of Validity

Operation: CreateAdminAccount(username: string, email: string, password: string)

Cross-references:

- Use case- Create Admin Account

Preconditions:

- none

Postconditions:

- Admin account *admin* was created (object creation)
- *admin.username* was set to username (attribute modified)
- *admin.password* was set to password (attribute modified)

Exceptions: none

Operation: addAccount(adminAccount: adminAccount)

Cross references:

- Use case- create Admin Account

Preconditions:

- Admin account exists

Postconditions:

- *admin* is added to the system (association formed)

Exceptions: none

Author: Josh Fulton
ID: UC-A.ACCT-02
Name: Add New User
Scope: App Account Creation System
Level: Admin Goal
Primary Actor: Admin

Stakeholders and interests:

- Admin
    - Wants to create an account for a user
- User
    - Wants an account to be created by admin

Precondition:

- Admin is logged into an admin account
- User notifies admin of force creation of account wanted

Postcondition:

- The admin creates the user account
- The user is able to access the new account

Minimal Guarantee

- The accounts previously created remain within the system.
- The admin can attempt to create an account again
- The admin account still exists

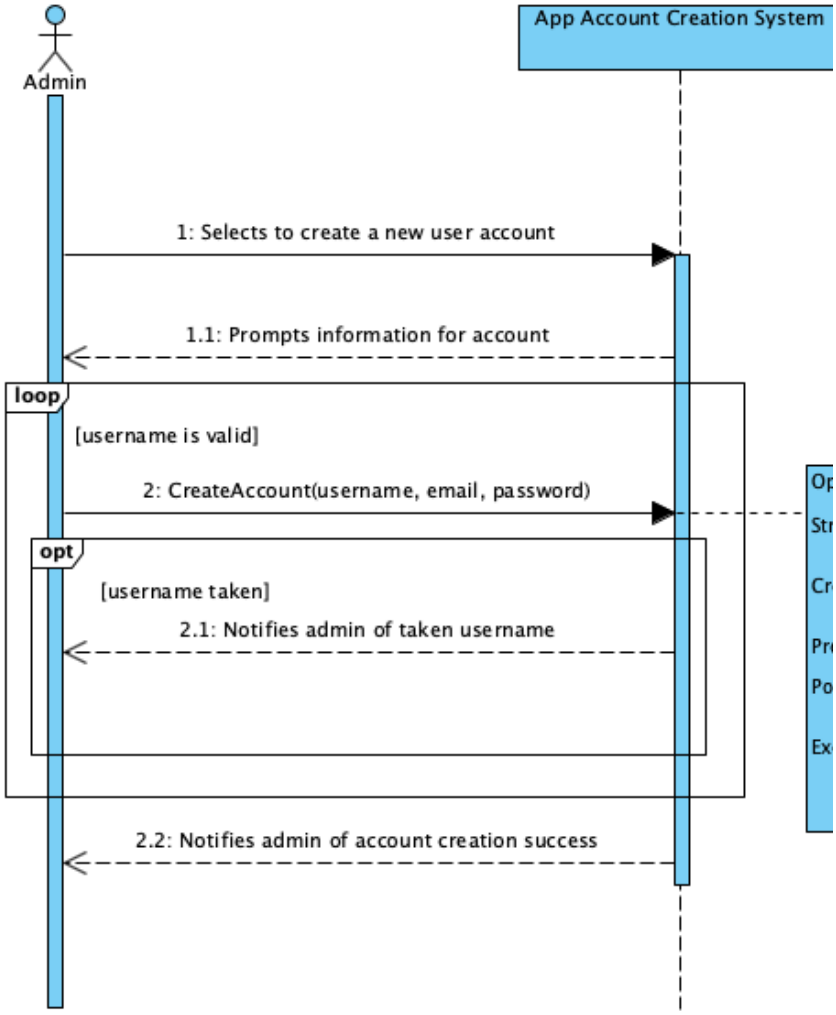
Main Success Scenario:

1. The admin selects “create user account”
2. The admin fills in login information for the user account
3. The admin selects “create”
4. The system notifies admin of success
5. The user is able to use the login information to access new account

Alternate Paths:

3a. The username the admin chooses is already in use
    i. The admin is able to enter another username
    ii. Repeats process until useable username is entered

4a. Due to unforeseen circumstances, the system says there is a failure and the account could not be created.
    i. The admin can attempt to create the account again


Admin
App Account Creation System
1: Selects to create a new user account
1.1: Prompts information for account
loop
[username is valid]
2: CreateAccount(username, email, password)
opt
[username taken]
2.1: Notifies admin of taken username
2.2: Notifies admin of account creation success

Operation Contract

CreateAccount(username: String, email: String, password: String)

Cross References:

- User Creates Account
- Admin Creates User Account

Preconditions:

- Admin account exists

Postconditions:

- User account *user* was created (object creation)
- *user.username* was set to username (attribute modified)
- *user.email* was set to email (attribute modified)
- *user.password* was set to password (attribute modified)

Exceptions:

- UsernameTaken if username is already taken

Author: Josh Fulton
ID: UC-A.ACCT-03
Name: Admin Changes User's Password
Scope: App Administrative System
Level: Admin Goal
Primary Actor: Admin

Stakeholders and interests:

- Customer
    - User wants to have their password changed
- System Admin
    - Employee wants to ensure the user has access to their account and updates password to allow user access.

Precondition:

- The admin is logged in
- The user has an account
- The user requests for a password change

Postcondition:

- The user's password is changed by the Admin
- The user is notified
- The user is able to login after the password reset

Minimal Guarantee:

- The account is still able to be access by the admin
- The user's account is not deleted
- The user has not entered eternal rest

Main Success Scenario:

1. The administrator navigates to the account of the user who requested the password reset
2. The administrator successfully resets the password
3. The user is notified of the password reset
4. The user is able to login to their account by using the new password

Alternate Paths:

1a. The admin is not notified for an unidentified reason and the user asks for another reset.

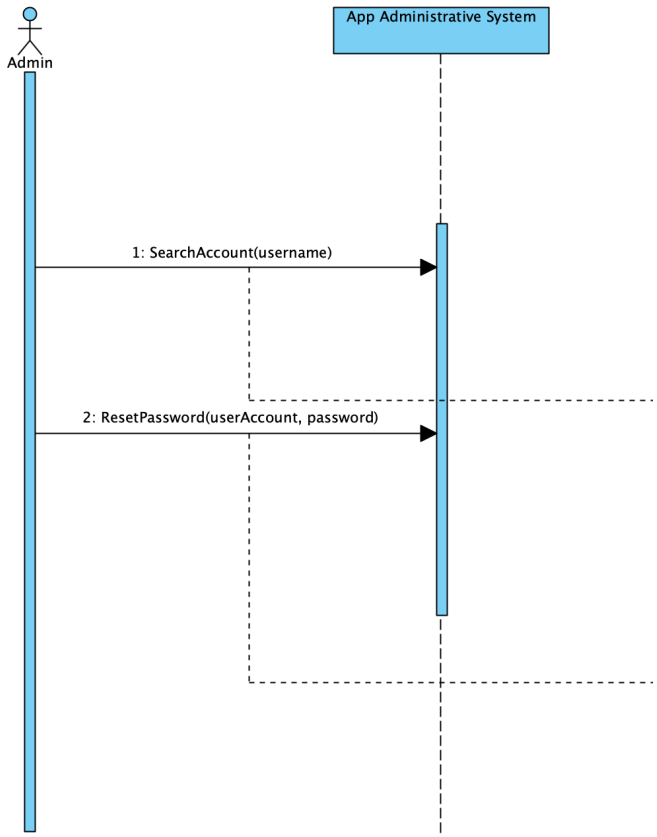
2a. The admin changes the wrong user's password and resets the password of another

user that was not intended.

3a. The admin changes the user's password but it is not the same as told to the user, causing confusion.

3b. The user is not notified and therefore asks for another reset.

4a. The user forgets the new password and asks for another password reset


Admin
App Administrative System
1: SearchAccount(username)
2: ResetPassword(userAccount, password)

Operation: SearchAccount(username: string)

Cross-References:

- User Case - Admin changes user's password

Preconditions:

- The userAccount exists
- The adminAccount exists

Postconditions:

- none

Exceptions:

- InvalidUsername if an account does not exist with the input username

Operation: RegisterPassword(userAccount: userAccount, password: string)

Cross-references:

- Use Case - Admin Changes User's password

Preconditions:

- userAccount exists

Postconditions:

- *user.password* was changed to password (attribute modification)

Exceptions:

- SamePassword if input password is same as current password

Author: Emily Wokoek
ID: UC-TRACK-01
Name: UC Add Workout
Scope: App Health Data Records System
Level: User Goal
Primary Actor: User

Stakeholders and interests:

- Customer
    - User of the health app (NOT A TRAINER) who is interested in tracking their health data, joining classes, and setting goals
- System Admin
    - Employee interested in making sure users have support when running into issues with account creation

Precondition:

- The user has downloaded the application
- The user is logged in

Postcondition:

- New user-inputted information has been saved in their personal workout log
- System maintains workout log information that isn't edited
- The user is able to access the newly logged workout

Minimal Guarantee:

- The same amount of users is still stored in the system as before
- No corrupted or incorrect data is stored in the system
- If failure occurs, the systems produces an error message

Main success scenario:

1. The system presents action options about data and classes
2. The user wishes to record a new workout in their personal workout log and is able to do so
3. The system should evaluate whether the specified inputted values are valid
4. If values are sound, system presents a save option
5. The system saves the information and presents the action options again
6. The user is able to view updated report of their workouts

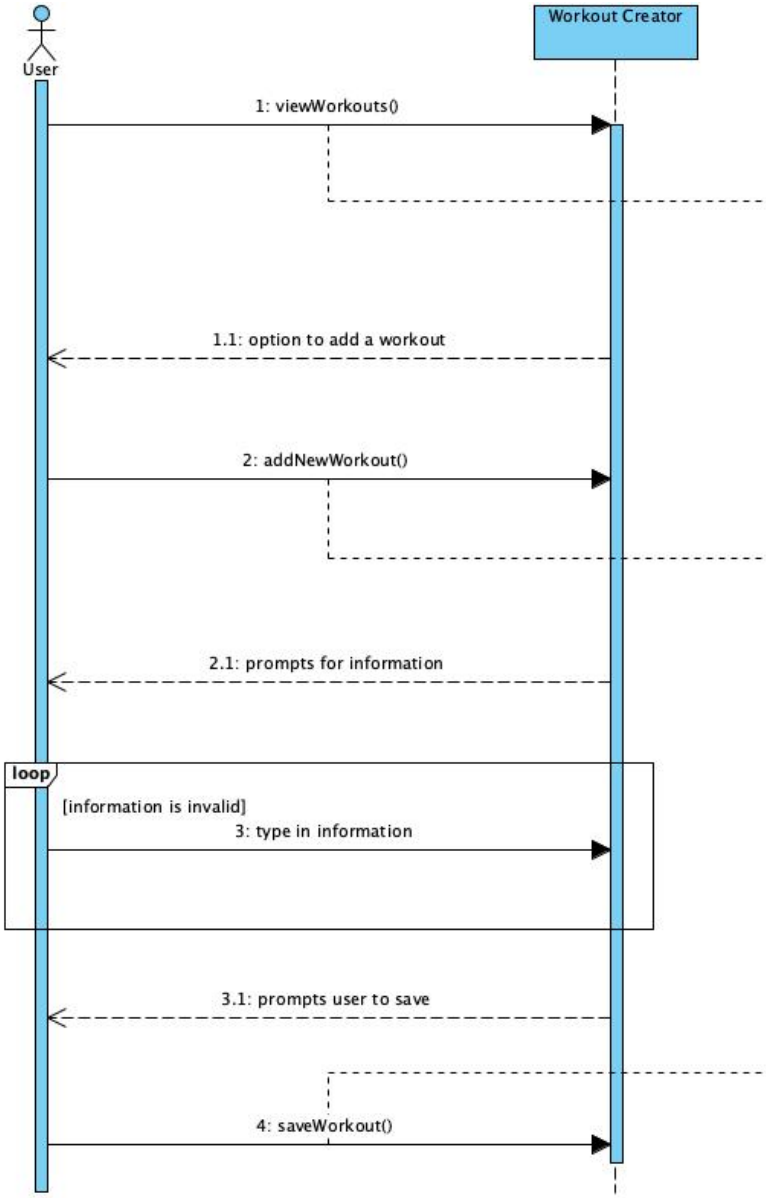
Alternate paths :

3a. There is an invalid value inputted

i. The system will prompt the trainer to re-enter the information

3b. A field is left blank

i. The system will prompt the trainer to either enter a value or cancel the operation


User
Workout Creator
1: viewWorkouts()
1.1: option to add a workout
2: addNewWorkout()
2.1: prompts for information
loop
[information is invalid]
3: type in information
3.1: prompts user to save
4: saveWorkout()

Operation: viewWorkouts()

Cross-References:

- Use Case - Add Workout

Preconditions:

- App is opened to home page

Postconditions:

- none

Exceptions:

- none

Operation: addNewWorkout()

Cross-References:

- Use Case - Add Workout

Preconditions:

- App is opened to workout log

Postconditions:

- Workout instance *workout* was created (instance creation)

Exceptions:

- none

Operation: saveWorkout()

Cross-References:

- Use Case - Add Workout

Preconditions:

- Valid information entered into all relevant fields

Postconditions:

- *workout* attributes are modified was created (attributes modified)
- *workout* associated with *user* (associations created)

Exceptions:

- none

Author: Kiera Shepperd
Name: UC-TRACK-02 Log Information
ID: UC-TRACK-02
Scope: App Health Data Records System
Level: User Goal
Primary Actor: User

Stakeholders and interests:

- User
    - User wants to track their health data (such as water consumption, sleep, caloric intake, and weight)

Precondition:

- The user is logged in

Postcondition:

- New user-inputted information has been saved in their personal log
- System maintains logged information that isn't edited

Minimal Guarantee:

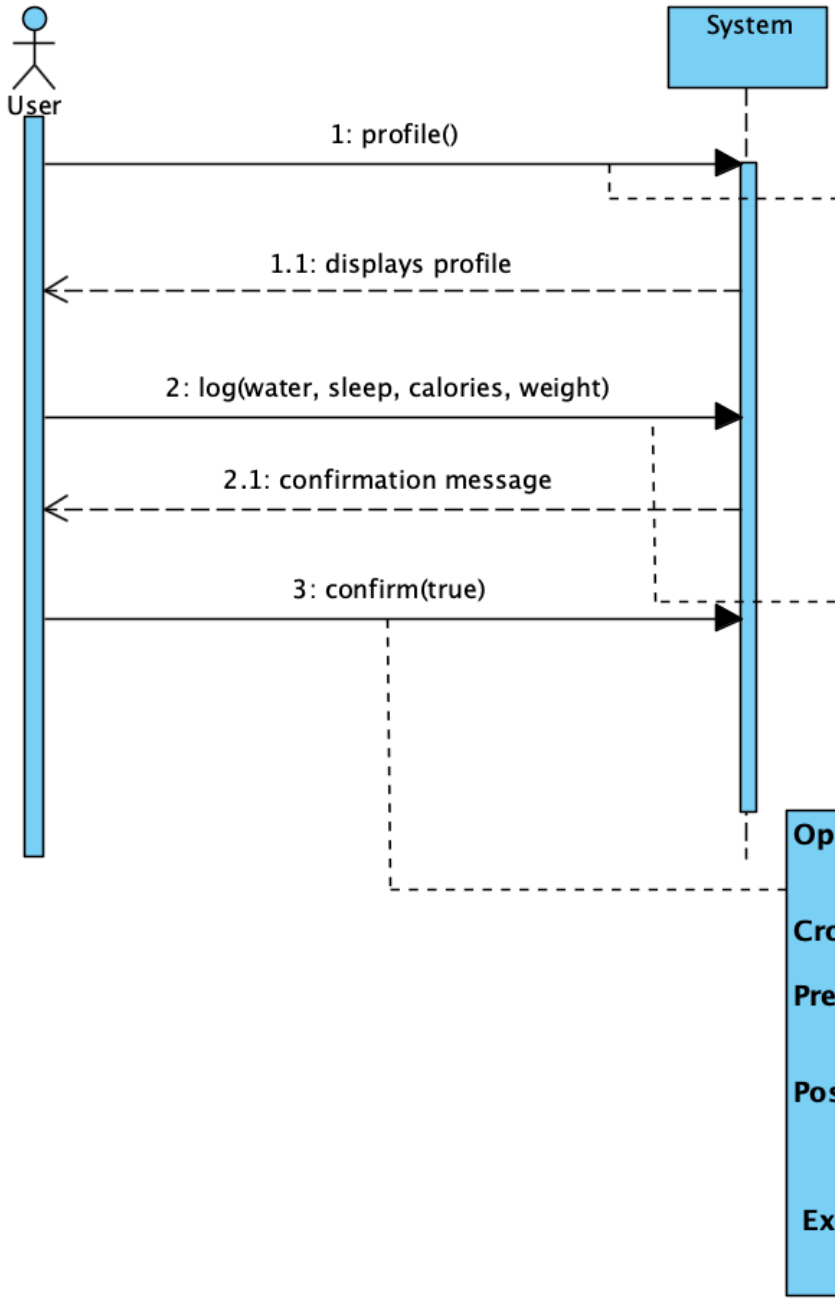
- The same amount of users is still stored in the system as before
- No corrupted or incorrect data is stored in the system
- The system will provide error messaging if something goes wrong

Main success scenario:

1. User navigates to their profile
2. User opens the information logging page
3. User inputs information as desired
4. The system should evaluate whether the specified inputted values are valid
5. If values are sound, system presents a confirmation message and a save option
6. The system saves the information and presents the action options again

Alternate paths:

5a. A field is left blank
    i. The system will automatically fill the field with a -1 to indicate that the field was left blank
    ii. The system will the store the inputted information as well as any blank fields
5b. There is an invalid value inputted
    i. The system will prompt the user to re-enter the information


User
System
1: profile()
1.1: displays profile
2: log(water, sleep, calories, weight)
2.1: confirmation message
3: confirm(true)
Op
Cr
Pre
Pos
Ex

Operation: profile()

Cross-References:

- Use Case - View User Report
- Use Case - Log Information

Preconditions:

- *user* is logged in

Postconditions:

- none

Exceptions:

- none

Operation: log(double water, double sleep, int calories, double weight)

Cross-References:

- Use Case - Log Information

Preconditions:

- *user* is logged in

Postconditions:

- A DailyMetrics instance *dm* was created (instance creation)
- *dm* attributes were set to inputted quantities (attribute modification)

Exceptions:

- none

Operation: confirm(boolean true)

Cross-References:

- Use Case - Log Information

Preconditions:

- *user* is logged in
- *dm* has information store

Postconditions:

- *dm* was associated with *user* (association formed)

Exceptions:

- none

Author: Lawson Hale
ID: UC-U.CLASS-01
Name: Register for self-paced exercise plan
Scope: Self-paced Exercise Plan Choice System
Level: User Goal
Primary Actor: General User

Stakeholders and interests:

- User
  - Wants to register for a self-paced exercise plan a trainer made
- Trainer
  - Wants to create and monitor plans that users use

Precondition

- The user is logged in
- There must be plans and data for available plans

Postcondition

- The user is registered for a self-paced exercise plan

Minimal Guarantee

- The user's information is safe
- The self-paced plan still exists
- The trainer can access the self-paced plan
- No one's personal information is leaked
- No data about the class is corrupted
- The system does not crash

Main success scenario

1. The user navigates to the self-paced plan tab
2. The user chooses a self-paced exercise plan from the available list
3. The user views the overview of the plan
4. The user chooses register
5. The system validates that the plan exists
6. The system adds the user to the list of current users in the plan
7. The user now has a view of the plan that has completed dates and specific workout information for all of the plan
8. The trainer can see the users currently using the plan

Alternate paths

2a. There are no self-paced plans available

i. The user cannot choose a self-paced plan

3a. The user views a self-paced plan and chooses not to register for it

i. The user can see other plans OR exits the app

6a. The user does not want to be in the self-paced plan anymore

i. The user deregisters

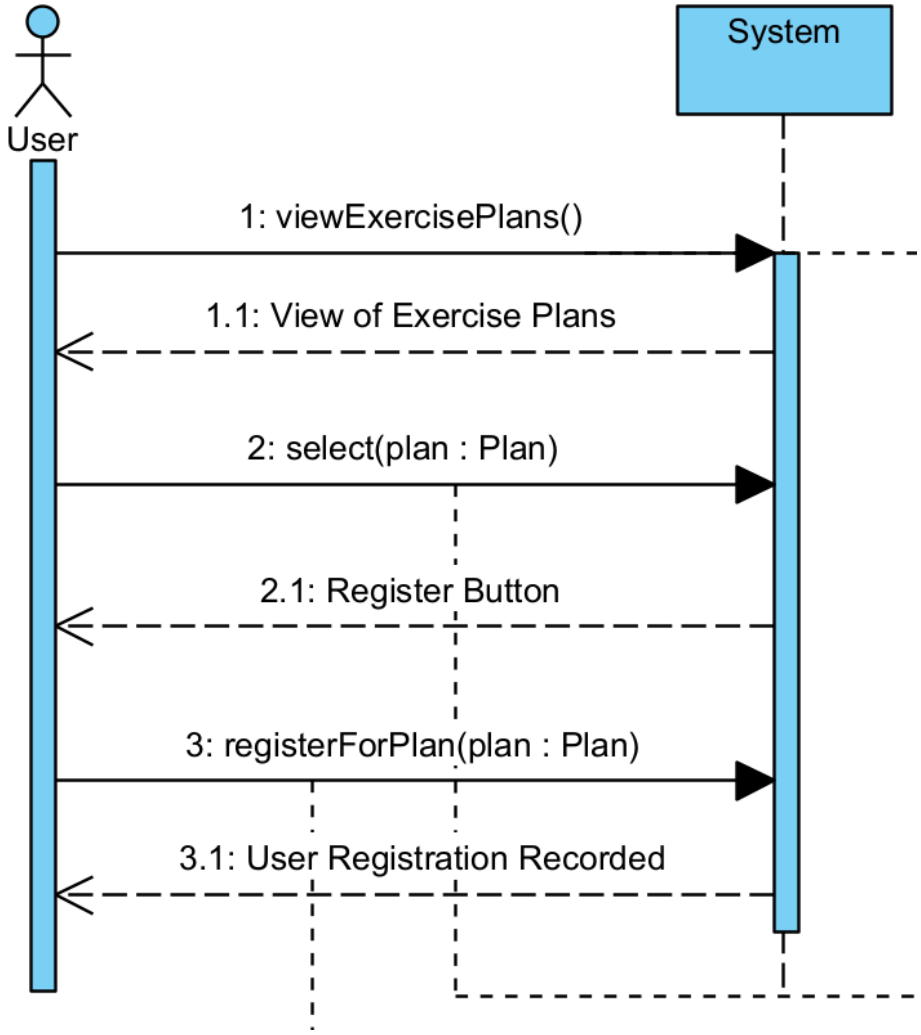
ii. The user is removed from the class list

5a. The user reviews the plan

i. The user can leave 1-5 stars for the self-paced exercise plan

ii. The plan has reviews that average to give it a score

SP Exercise Plan SSD



Operation: viewExercisePlans()

Cross-References:

- Use Case - Register for Self-Paced Workout Plan

Preconditions:

- *user* is logged in

Postconditions:

- none

Exceptions:

- NoPlans if there are no plans

Operation: select()

Cross-References:

- Use Case - Register for Self-Paced Workout Plan

Preconditions:

- *user* is logged in
- *plan* exists

Postconditions:

- none

Exceptions:

- none

Operation: registerForPlan(plan: Plan)

Cross-References:

- Use Case - Register for Self-Paced Workout Plan

Preconditions:

- *user* is logged in
- *plan* exists

Postconditions:

- *plan* was added to the list *user.plans* (association formed)
- *user* was added to the list *plan.roster* (association formed)

Exceptions:

- none

Author: Lawson Hale
ID: UC-U.CLASS-02
Name: Register for Class
Scope: Workout Class Choice System
Level: User Goal
Primary Actor: General User

Stakeholders and interests:

- User
    - Wants to register for and mark attendance for a workout class
- Trainer
    - Wants to make sure their class is available and ready

Precondition

- The user is logged in
- There must be classes and data for available classes

Postcondition

- The user is registered for a class and is on the class roster
- The trainer can see the new registration
- The class can become full if there are no more spaces left

Minimal Guarantee

- The user's information is safe
- The workout class still exists
- The trainer can access the workout class
- No one's personal information is leaked
- No data about the class is corrupted

Main success scenario

1. The user navigates to the workout class tab
2. The user chooses a workout class from the list available
3. The user hits the register button
4. The system validates that the class has available spots in it
5. The system enters the user into the class roster
6. The user can see the class roster with them in it

Alternate paths

2a. There are no workouts available

i. The user cannot choose a workout

4a. The workout class is full

i. The user cannot register for it

ii. The system sends an error message saying the class is full

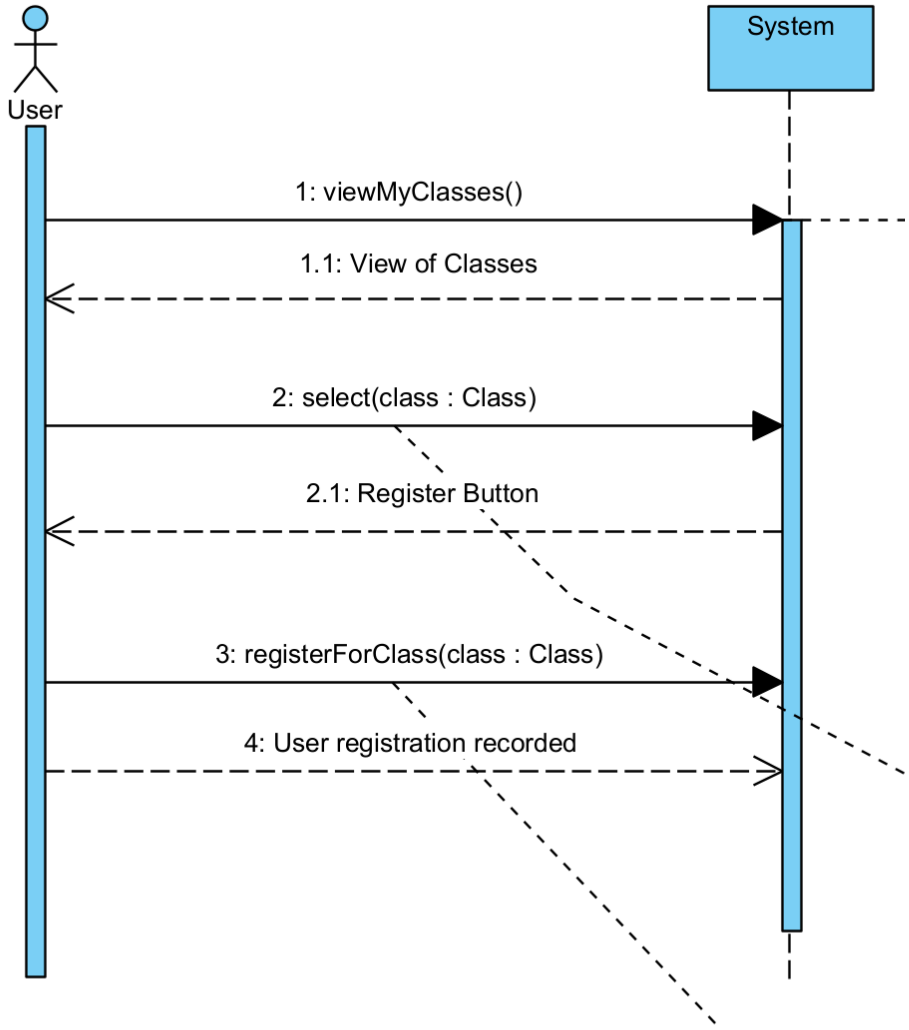
2a. The user views a workout class and chooses not to register for it

i. The user exits the app

6a. The user does not want to be in a class anymore

i. The user deregisters

ii. The user is removed from the class list


User
System
1: viewMyClasses()
1.1: View of Classes
2: select(class : Class)
2.1: Register Button
3: registerForClass(class : Class)
4: User registration recorded

Operation: viewMyClasses()

Cross-References:

- Use Case - Register for Workout Class

Preconditions:

- *user* is logged in

Postconditions:

- none

Exceptions:

- NoClasses if there are no classes

Operation: select()

Cross-References:

- Use Case - Register for Workout Class

Preconditions:

- *user* is logged in
- *class* exists

Postconditions:

- none

Exceptions:

- none

Operation: registerForClass(class: Class)

Cross-References:

- Use Case - Register for Workout Class

Preconditions:

- *user* is logged in
- *class* exists

Postconditions:

- *class* was added to the list *user.classes* (association formed)
- *user* was added to the list *class.roster* (association formed)

Exceptions:

- FullClass if the class is full

Author: Lawson Hale
ID: UC-U.CLASS-03
Name: Attend Workout Class
Scope: Workout Class Choice System
Level: User Goal
Primary Actor: General User

Stakeholders and interests:

- User
  - Wants to attend a workout class
- Trainer
  - Interested in making sure their class is available and ready, with a fleshed-out plan and attendees

Precondition

- The user is logged in
- The user must be registered for the class

Postcondition

- The user has been marked as present for the class

Minimal Guarantee

- The user's information is safe
- The workout class still exists
- The trainer can access the workout class
- No one's personal information is leaked
- No data about the class is corrupted
- All previous attendance records are still there

Main success scenario

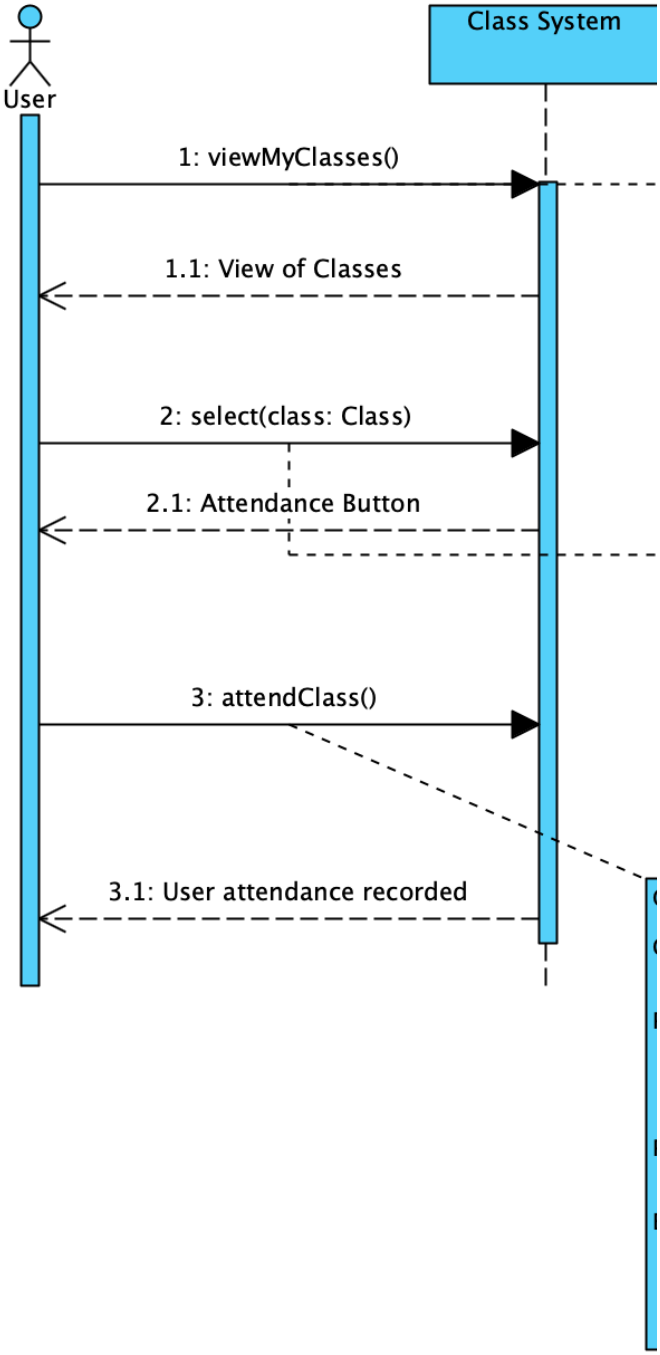
1. The user navigates to their profile
2. The user chooses a workout class from their registered workout classes
3. The system provides them an option to select "attend"
4. The user hits the "Attend" button
5. The system validates that the user is registered for the appropriate class
6. The system marks the user as present
7. The trainer can view that the user is present

Alternate paths

3a. If the user attempts to attend the same class twice

i. The system sends an error message saying the user is already present
ii. The user is not marked present twice

Attend Workout Class



Operation: viewMyClasses()

Cross References:

- Use Case - Attend Workout Class

Preconditions:

- *user.classes* is not empty

Postconditions:

- none

Exceptions:

- NoClasses if *user.classes* is empty

Operation: select(class : Class)

Cross References:

- Use Case - Attend Workout Class

Preconditions:

- *class* exists

Postconditions:

- none

Exceptions:

- none

Operation: attendClass()

Cross References:

- Use Case - Attend Workout Class

Preconditions:

- *class.started* is true

Postconditions:

- *class* is added to the list *user.attendedClasses* (attribution modified)
- *user* is associated with attending the class (association formed)

Exceptions:

- ClassNotStarted if *class.started* is false

Author: Emily Wokoek
ID: UC-T.CLASS-01
Name: UC Create Self-Paced Plan
Scope: App Self-Paced Plan Creation System
Level: User Goal
Primary Actor: Trainer

Stakeholders and interests:

- Trainer
    - Interested in creating a self-paced plan for users to do

Precondition

- Trainer is logged in

Postcondition

- New self-paced plan is created and is stored in the system
- System maintains previously created self-paced plans
- The trainer is able to access the newly added plan

Minimal Guarantee

- The same amount of users is still stored in the system as before
- No corrupted or half filled out data is stored in the system, user must select the save option
- The system should provide an error messaging if anything goes wrong

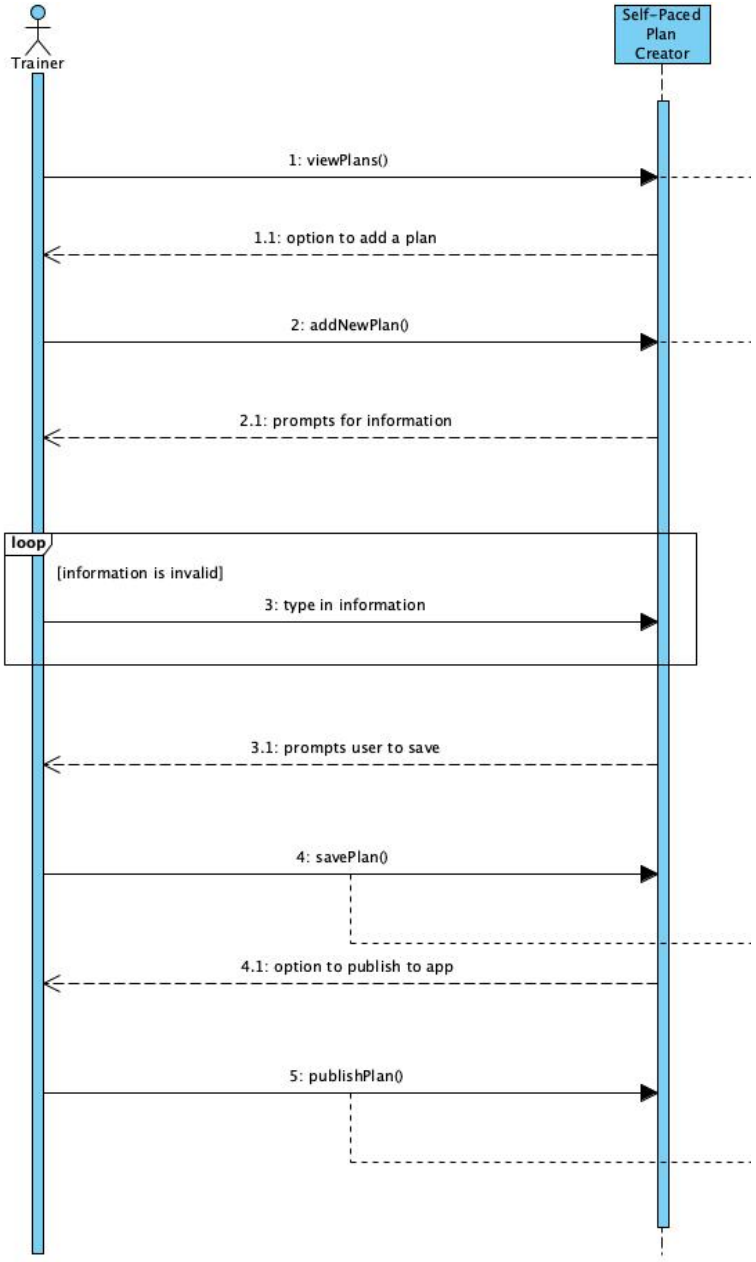
Main success scenario

1. The system presents action options about classes and self-paced plans
2. The trainer wishes to create a new self-paced plan and is able to do so
3. The system prompts trainer for information about class
4. The user inputs desired traits for class
5. The system should evaluate whether the specified inputted value is valid
6. If value is sound, system presents a create and save option to save the draft plan
7. The system saves the information and presents the option to publish the self-paced plan to the public
8. The trainer is able to preview what the plan looks like to users of the app

Alternate paths

5a. There is an invalid value inputted
    i. The system will prompt the trainer to re-enter the information
5b. A field is left blank

i. The system will prompt the trainer to either enter a value or cancel the operation


Trainer
Self-Paced Plan Creator
1: viewPlans()
1.1: option to add a plan
2: addNewPlan()
2.1: prompts for information
loop
[information is invalid]
3: type in information
3.1: prompts user to save
4: savePlan()
4.1: option to publish to app
5: publishPlan()

Operation: viewPlans()

Cross References:

- Use Case - Create Self-Paced Plan

Preconditions:

- App is opened to home page

Postconditions:

- none

Exceptions:

- none

Operation: addNewPlan()

Cross References:

- Use Case - Create Self-Paced Plan

Preconditions:

- App is opened to plans page

Postconditions:

- Plan instance *plan* is created (object creation)

Exceptions:

- none

Operation: savePlan(information: String)

Cross References:

- Use Case - Create Self-Paced Plan

Preconditions:

- Information in fields is valid

Postconditions:

- *plan* attributes were set to values passed in (attribute modification)

Exceptions:

- none

Operation: publishPlan()

Cross References:

- Use Case - Create Self-Paced Plan

Preconditions:

- *plan* is created and instantiated

Postconditions:

- *plan* is added to *publicPlans* list (association formed)

Exceptions:

- none

Author: Carter Lewis
ID: UC-T.CLASS-02
Name: UC Create Class
Scope: App Self-Paced Plan Creation System
Level: User Goal
Primary Actor: Trainer

Stakeholders and interests:

- Trainer
    - Interested in creating an asynchronous class for users to do

Precondition

- Trainer is logged in

Postcondition

- New asynchronous class is created and is stored in the system
- System maintains previously created classes
- The trainer is able to access and start the newly added class

Minimal Guarantee

- The same amount of users is still stored in the system as before
- No corrupted or half filled out data is stored in the system, user must select the save option
- The system should provide an error messaging if anything goes wrong

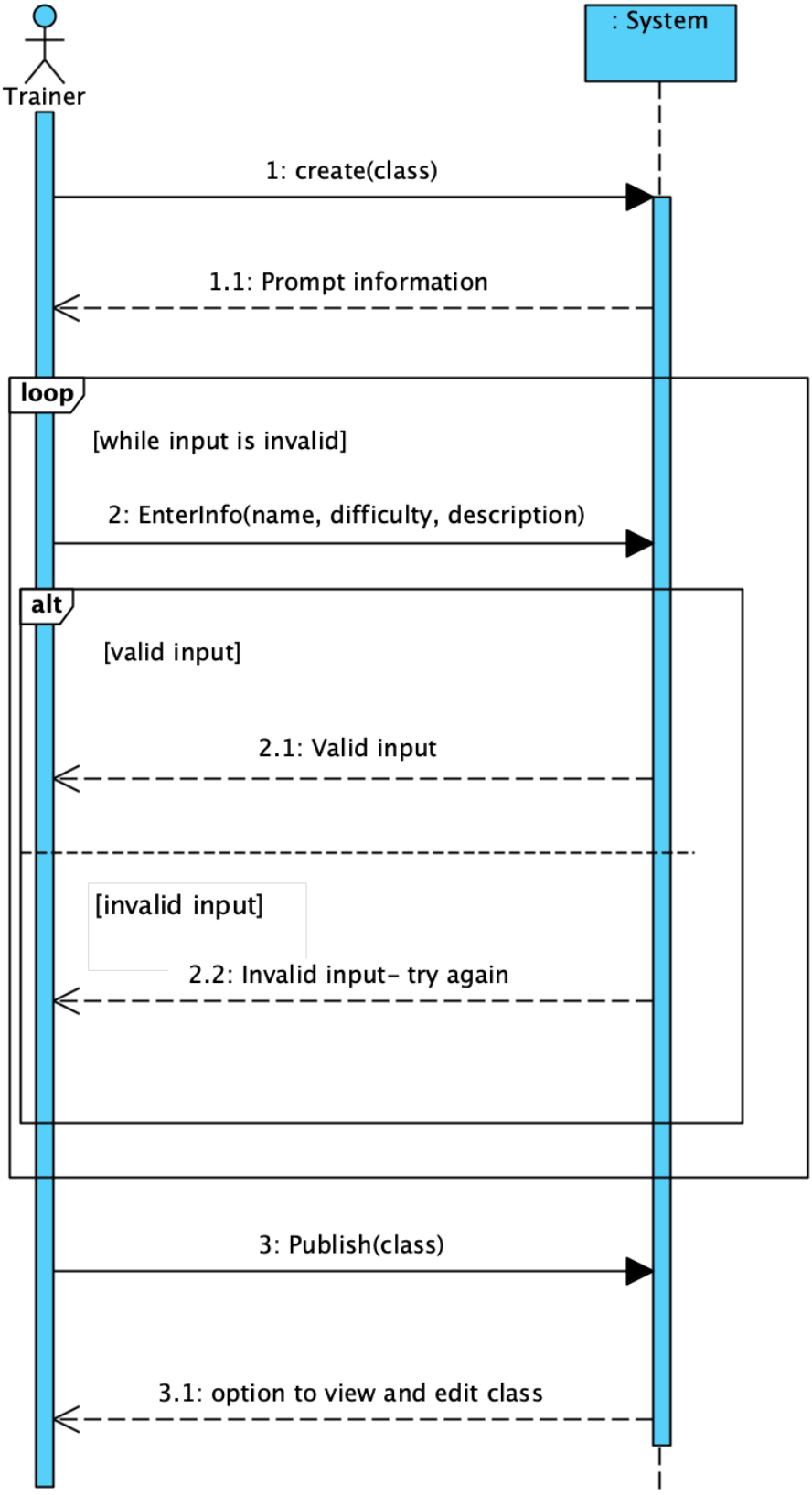
Main success scenario

1. The system presents action options about classes
2. The trainer wishes to create a new asynchronous class and is able to do so
3. The system prompts trainer for information (name, description, difficulty) about class
4. The user inputs desired traits for class
5. The system should evaluate whether the specified inputted value is valid
6. If value is sound, system presents a create and save option to save the draft class
7. The system saves the information and presents the option to publish the class to the public
8. The trainer is able to preview what the class looks like to users of the app

Alternate paths

5a. There is an invalid value inputted
    i. The system will prompt the trainer to re-enter the information
5b. A field is left blank

i. The system will prompt the trainer to either enter a value or cancel the operation


Trainer
: System
1: create(class)
1.1: Prompt information
loop
[while input is invalid]
2: EnterInfo(name, difficulty, description)
alt
[valid input]
2.1: Valid input
[invalid input]
2.2: Invalid input- try again
3: Publish(class)
3.1: option to view and edit class

Operation: create(class: class)

Cross References

- Use Case: Trainer Class Creation

Preconditions:

- *trainer* is logged in

Postconditions

- Fitness course instance *course* was created (Object creation)

Exceptions

- None

Operation: enterInfo(String name, int difficulty, Type: type, String description)

Cross References

- Use Case: Trainer Class Creation

Preconditions:

- *trainer* is logged in
- *course* is created

Postconditions

- *class* attributes were updated with inputted information (Attribute modification)

Exceptions

- None

Operation: submit(class: class)

Cross References

- Use Case: Trainer Class Creation

Preconditions:

- *trainer* is logged in
- *course* has been instantiated

Postconditions

- Association between *trainer* and *class* (Association formed)

Exceptions

- None

Author: Emily Wokoek
ID: UC-T.CLASS-03
Name: UC Modify Plan
Scope: App Self-Paced Plan System
Level: User Goal
Primary Actor: Trainer

Stakeholders and interests:

- Trainer
    - Interested in modified a pre-existing self-paced plan

Precondition

- The trainer is logged in

Postcondition

- Self-paced plan is modified and is stored in the system
- System maintains previously created self-paced plans that are not edited
- The trainer is able to access the newly added or edited plan

Minimal Guarantee

- The same amount of users is still stored in the system as before
- No corrupted or half filled out data is stored in the system, user must select the save option
- The system should provide an error messaging if anything goes wrong

Main success scenario

1. The system presents action options about classes and self-paced plans
2. The trainer selects the option modify a new self-paced plan
3. The trainer selects one of their self-paced plans
4. The system displays the plan information to the trainer
5. The trainer edits desired fields for class
6. The system should evaluate whether the specified inputted value is valid
7. If value is sound, system saves the information and publishes the self-paced plan to the public
8. The trainer is able to preview what the plan looks like to users of the app

Alternate paths

4a. Trainer enters the edit mode and then decides that they no longer wish to edit any of the information

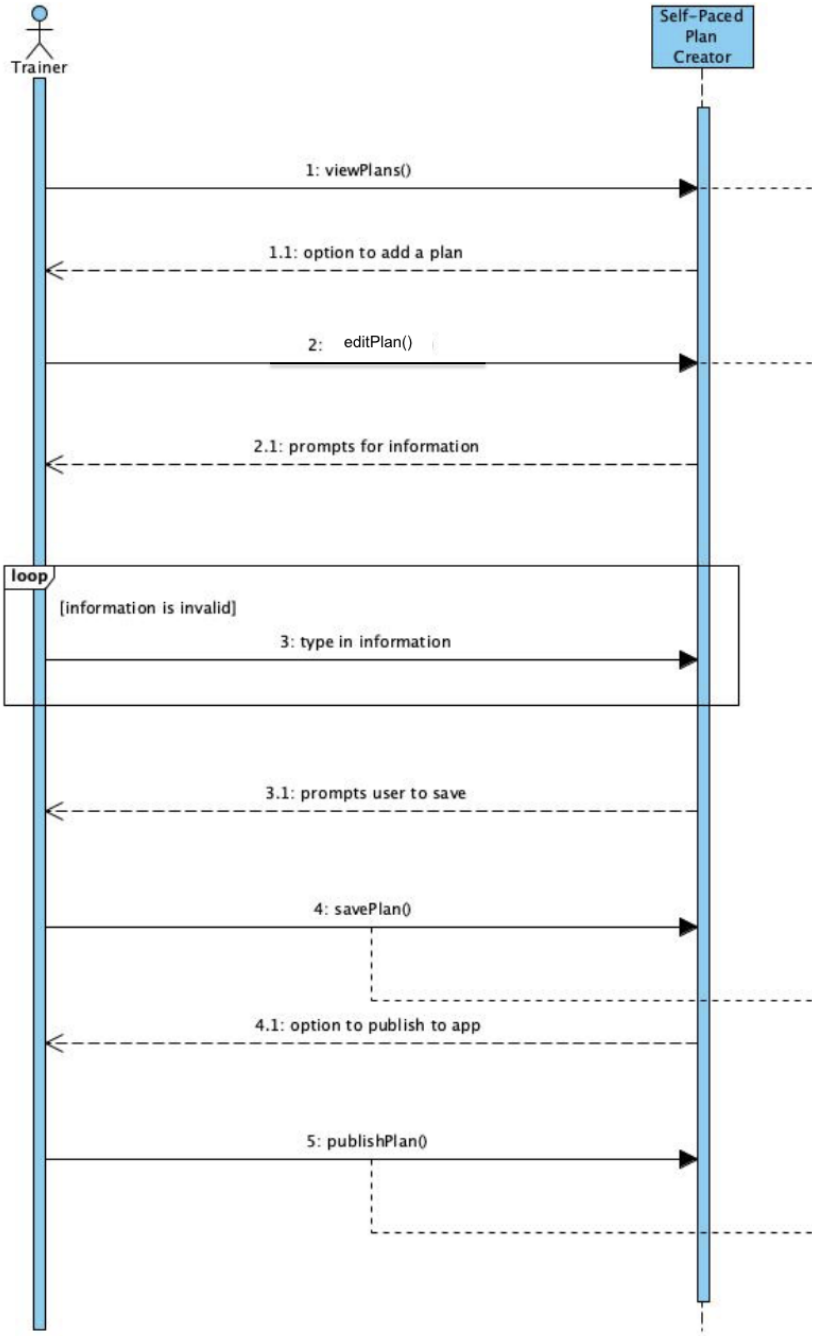
i. The system presents the option to cancel this operation

6a. There is an invalid value inputted

i. The system will prompt the trainer to re-enter the information

6b. A field is left blank

i. The system will prompt the trainer to either enter a value or cancel the operation


Trainer
Self-Paced Plan Creator
1: viewPlans()
1.1: option to add a plan
2: editPlan()
2.1: prompts for information
loop
[information is invalid]
3: type in information
3.1: prompts user to save
4: savePlan()
4.1: option to publish to app
5: publishPlan()

Operation: viewPlans()

Cross References:

- Use Case - Create Self-Paced Plan

Preconditions:

- App is opened to home page

Postconditions:

- none

Exceptions:

- none

Operation: editPlan()

Cross References:

- Use Case - Create Self-Paced Plan

Preconditions:

- *plan* exists

Postconditions:

- none

Exceptions:

- none

Operation: savePlan(information: String)

Cross References:

- Use Case - Create Self-Paced Plan

Preconditions:

- Information in fields is valid

Postconditions:

- *plan* attributes were set to values passed in (attribute modification)

Exceptions:

- none

Operation: publishPlan()

Cross References:

- Use Case - Create Self-Paced Plan

Preconditions:

- *plan* is created and instantiated

Postconditions:

- none

Exceptions:

- none

Author: Carter Lewis
ID: UC-T.CLASS-04
Name: UC Edit existing class
Scope: App Self-Paced Plan Creation System
Level: User Goal
Primary Actor: Trainer

Stakeholders and interests:

- Trainer
    - Interested in editing attributes of existing class

Precondition

- Trainer is logged in

Postcondition

- New attributes of class are saved and visible by users (Object creation)
- System maintains previously created asynchronous classes
- The trainer is able to access the newly edited class, and edit again if desired

Minimal Guarantee

- The same amount of users is still stored in the system as before
- No corrupted or half filled out data is stored in the system, user must select the save option
- The system should provide an error messaging if anything goes wrong

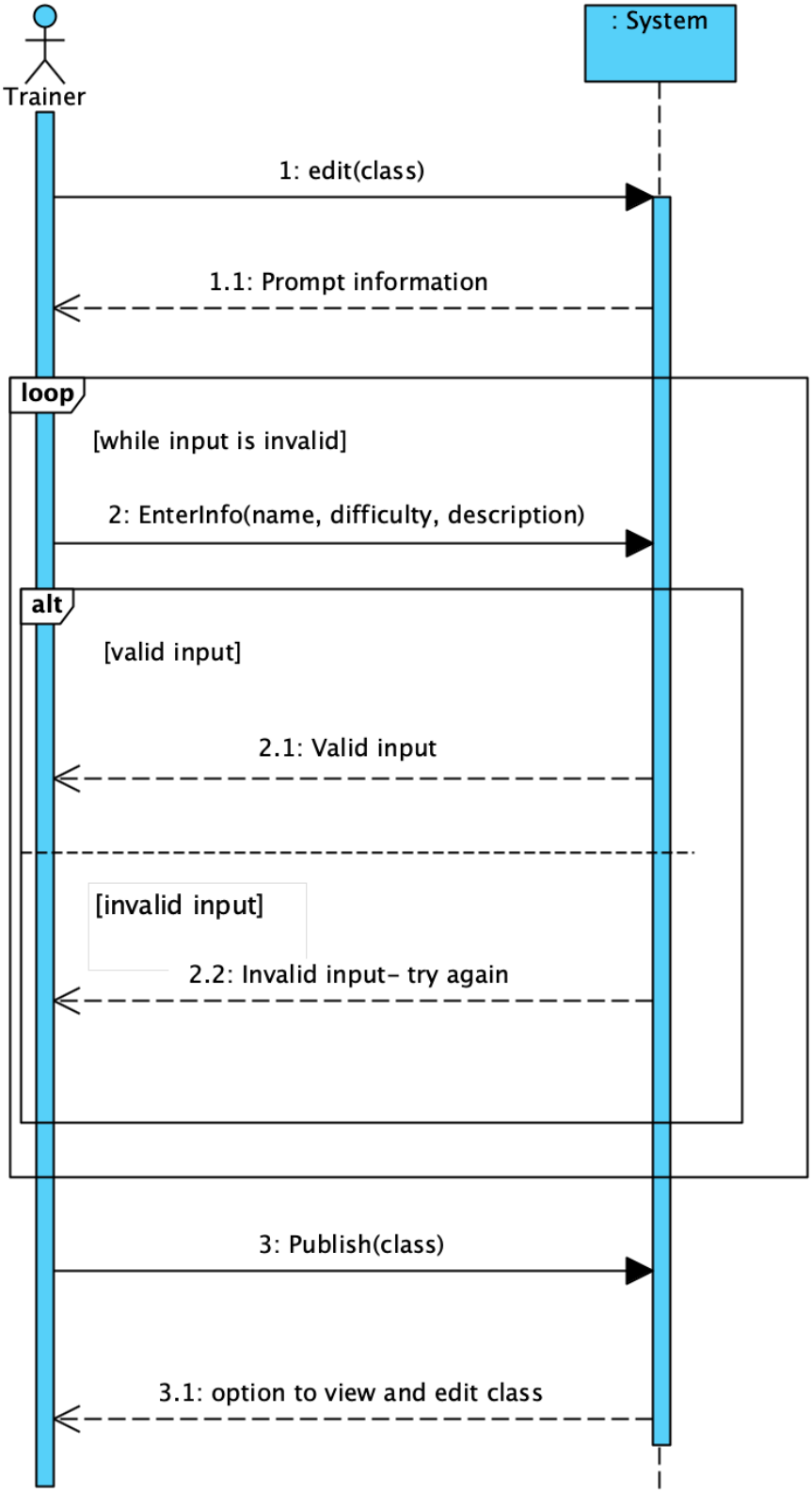
Main success scenario

1. The system presents action options about classes
2. The system prompts trainer for information about class
3. The user inputs desired traits for class
4. The system should evaluate whether the specified inputted value is valid
5. If values are sound, system presents a create and save option to save the draft plan
6. The system saves the information and updates all information for users to see
    a. Notifies members of a class when edits are made via email or push notification?
        i. Not in current design, but could be implemented in future iterations
7. The trainer is able to preview what the class looks like to users of the app

Alternate paths

5a. There is an invalid value inputted
    i. The system will prompt the trainer to re-enter the information
5b. A field is left blank

i. The system will prompt the trainer to either enter a value or cancel the operation


Trainer
: System
1: edit(class)
1.1: Prompt information
loop
[while input is invalid]
2: EnterInfo(name, difficulty, description)
alt
[valid input]
2.1: Valid input
[invalid input]
2.2: Invalid input- try again
3: Publish(class)
3.1: option to view and edit class

Operation: edit(class: class)

Cross References

- Use Case: Trainer Edit Class

Preconditions:

- *trainer* is logged in
- *class* exists

Postconditions

- *class* is removed from the *publicClasses* list (association broken)

Exceptions

- None

Operation: enterInfo(String name, int difficulty, Type: type, String description)

Cross References

- Use Case: Trainer Edit Class

Preconditions:

- *trainer* is logged in
- *class* exists

Postconditions

- *class* attributes were updated with new information (Attribute modification)

Exceptions

- None

Operation: publish(class: class)

Cross References

- Use Case: Trainer Edit Class

Preconditions:

- Trainer is logged in
- Valid class information has been input

Postconditions

- *class* is added to the *publicClasses* list (Association Formed)

Exceptions

- None

Author: Kiera Shepperd
Use Case: UC-T.CLASS-05 Host Class
ID: UC-T.CLASS-05
Scope: Trainer Account
Level: Trainer Goal
Primary Actor: Trainer

Stakeholders and Interests

- Trainer
    - Wants to start a preexisting class

Pre-Conditions

- Trainer is logged into the app
- Class is created

Post-Condition

- Class statistics are recorded
- Class is archived
- Class roster is visible to the trainer

Minimal Guarantee

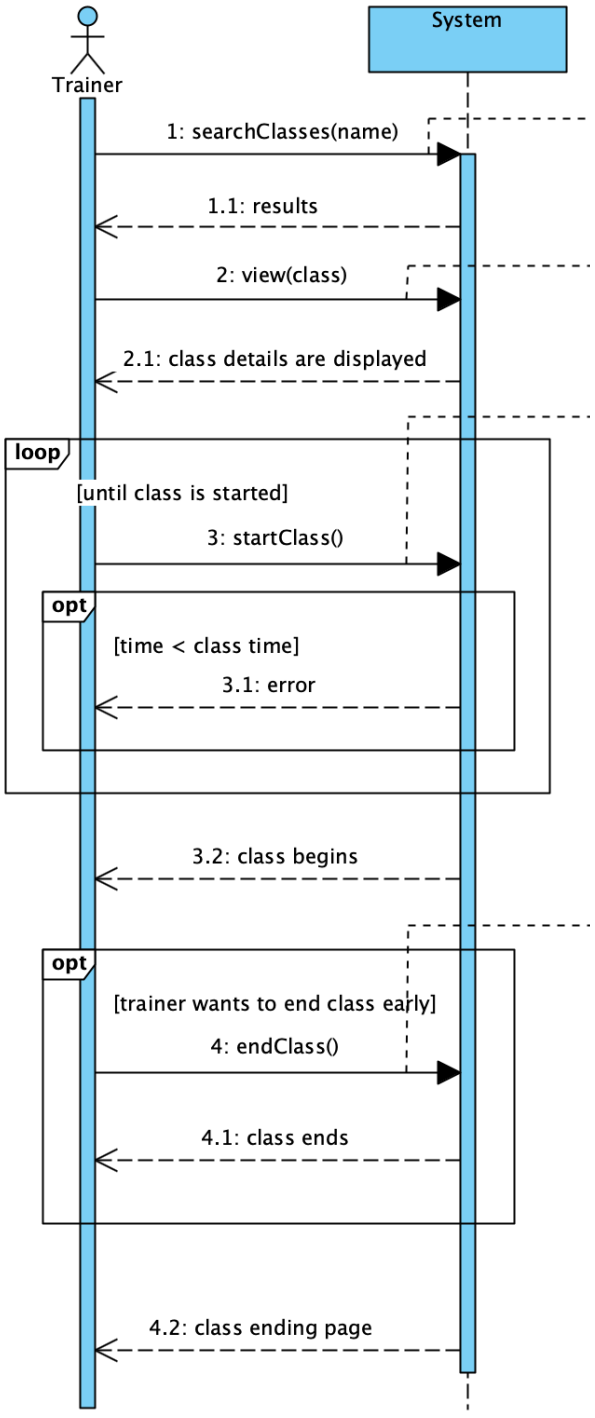
- The system will provide error messaging if something goes wrong
- Class will not be deleted

Main Success Scenario

1. Trainer searches for class
2. Trainer chooses desired class that they've created
3. At the time the class is scheduled to start, the trainer starts the class
4. The system keeps track of who attends the class
5. After the length of the class, the class ends

Alternative Paths

3a. The trainer tries to start the class before it is scheduled to start
    i. An error message appears and tells the trainer to wait until the posted class start time
3b. The class is not started 5 minutes after it's scheduled start time
    i. The class is rescheduled for 15 minutes later than the previously recorded time
4a. The trainer wishes to end the class early
    i. The trainer is selects the end class button


Trainer
System
1: searchClasses(name)
1.1: results
2: view(class)
2.1: class details are displayed
loop
[until class is started]
3: startClass()
opt
[time < class time]
3.1: error
3.2: class begins
opt
[trainer wants to end class early]
4: endClass()
4.1: class ends
4.2: class ending page

Operation: searchClasses(String name)

Cross References

- Use Case: Host Class

Preconditions:

- *trainer* is logged in

Postconditions

- none

Exceptions

- None

Operation: view(class: class)

Cross References

- Use Case: Host Class

Preconditions:

- *trainer* is logged in
- *class* exists

Postconditions

- none

Exceptions

- None

Operation: startClass(class: class)

Cross References

- Use Case: Host Class

Preconditions:

- *trainer* is logged in
- *class* exists

Postconditions

- *class.started* is set to true (attribute modification)

Exceptions

- TooEarly if *class.canStart* is false

Operation: endClass(class: class)

Cross References

- Use Case: Host Class

Preconditions:

- *trainer* is logged in
- *class* exists

Postconditions

- *class.archived* is set to true (attribute modification)

- *class* is removed from *publicClasses* (association broken)

Exceptions

- TooEarly if *class.canStart* is false

Author: Noah Matthew
Use Case: UC-REPORT-01 View User Report
ID: UC-REPORT-01
Scope: User Account
Level: User Goal
Primary Actor: User

Stakeholders and Interests

- User
    - Wants to view a report of their recent activity

Pre-Conditions

- User is logged into the app

Post-Condition

- All information remains stored in the user's profile

Minimal Guarantee

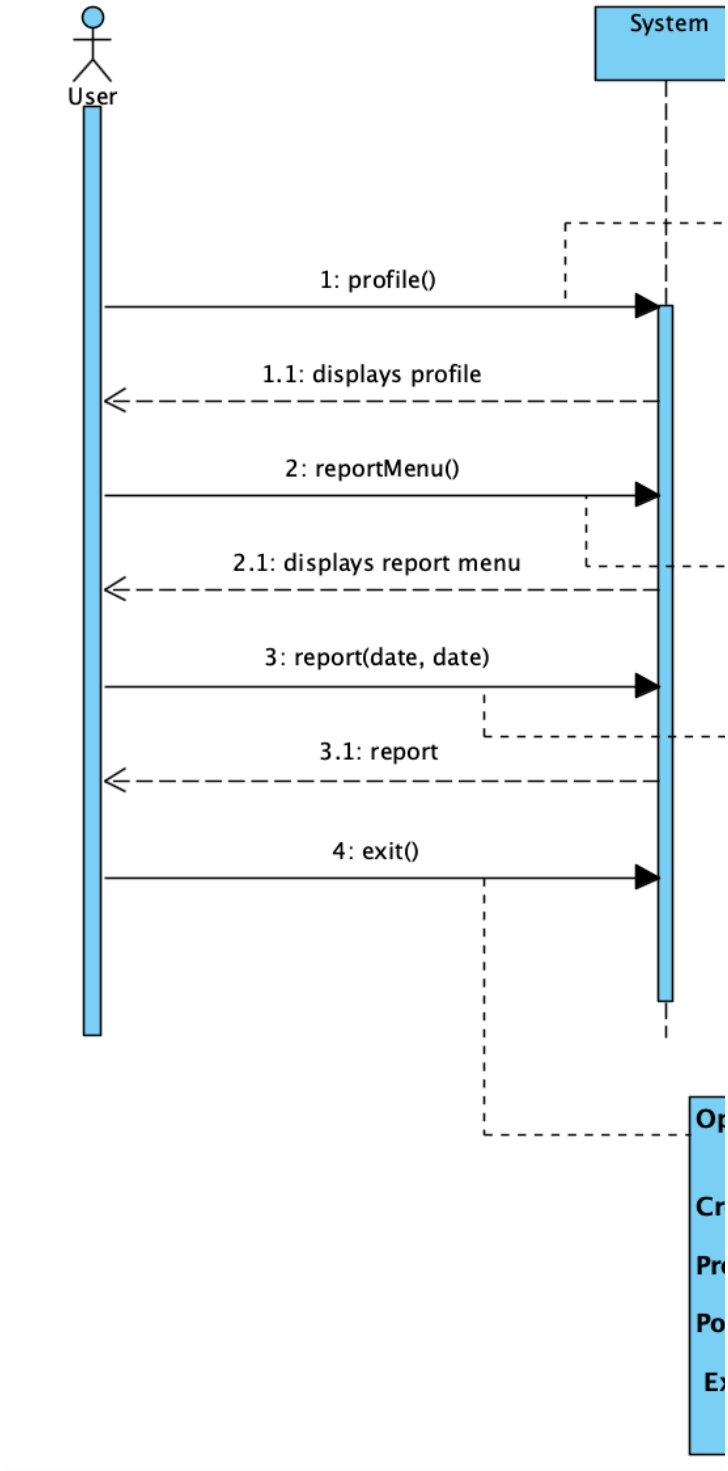
- The system will provide error messaging if something goes wrong
- User's data will not be deleted

Main Success Scenario

1. User navigates to their profile information
2. User clicks report
3. User selects a date range, ranging from when they created their account to the present
4. System averages the information logged during that time
5. System graphs the information logged during that time in comparison to the user's goals
6. System generates a report the information for the user to view
7. User exits the screen

Alternative Paths

3a. User no longer wants a report
    i. User exits the screen
5a. User's goals changed during the time frame selected for the report
    i. System graphs the current goal onto the entire time frame


User
System
1: profile()
1.1: displays profile
2: reportMenu()
2.1: displays report menu
3: report(date, date)
3.1: report
4: exit()

Operation: profile()

Cross References

- Use Case: View User Report
- Use Case: Log Information

Preconditions:

- *user* is logged in

Postconditions

- None

Exceptions

- None

Operation: reportMenu()

Cross References

- Use Case: View User Report

Preconditions:

- *user* is logged in

Postconditions

- None

Exceptions

- The User has no data

Operation: report(Date date, Date date)

Cross References

- Use Case: View User Report

Preconditions:

- *user* is logged in
- Report page is displayed

Postconditions

- Report instance *report* is created (object creation)
- *report* is added to *user.reports* list (association formed)

Exceptions

- NoData is thrown if there are no DailyMetrics between the dates provided

Operation: exit()

Cross References

- Use Case: View User Report

Preconditions:

- *user* is logged in
- Report page is displayed

Postconditions

- None

Exceptions

- None

Author: Kiera Shepperd
Use Case: UC-REPORT-02 View Class Report
ID: UC-REPORT-02
Scope: Trainer Account
Level: Trainer Goal
Primary Actor: Trainer

Stakeholders and Interests

- Trainer
    - Wants to view a report of those who attended the class

Pre-Conditions

- Trainer is logged into the app
- Class has started and ended
- Class statistics are recorded

Post-Condition

- All information remains stored in the archive with the class

Minimal Guarantee

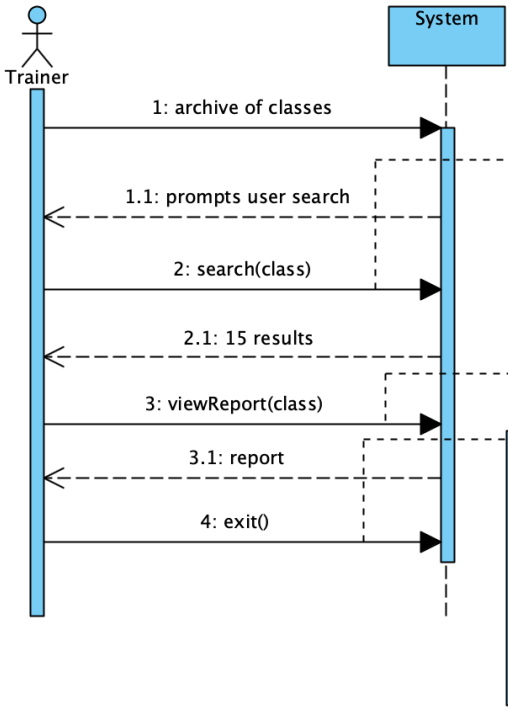
- The system will provide error messaging if something goes wrong
- Class will not be deleted

Main Success Scenario

1. Trainer navigates to report menu
2. Trainer searches for archived class
3. Trainer chooses desired class that they've hosted
4. Trainer clicks report
5. System generates a report of the attendees of the class and the average calories burned across the attendees
6. Trainer exists the screen

Alternative Paths

4a. Trainer no longer wants a report
    i. Trainer exits the screen
3a. Trainer wants to see reports that aggregate multiple classes
    i. Trainer selects multiple classes that they've hosted


Trainer
System
1: archive of classes
1.1: prompts user search
2: search(class)
2.1: 15 results
3: viewReport(class)
3.1: report
4: exit()

Operation: search(class: class)

Cross References

- Use Case: View Class Report

Preconditions:

- *trainer* is logged in
- *class* exist

Postconditions

- None

Exceptions

- NoClasses thrown if *trainer* has no classes

Operation: viewReport(class: class)

Cross References

- Use Case: View Class Report

Preconditions:

- *trainer* is logged in
- Report menu is displayed
- *class* exist

Postconditions

- Report instance created *classReport* (object creation)
- *classReport* added to *trainer.classReports* list (association formed)

Exceptions

- None

Operation: exit()

Cross References

- Use Case: View Class Report

Preconditions:

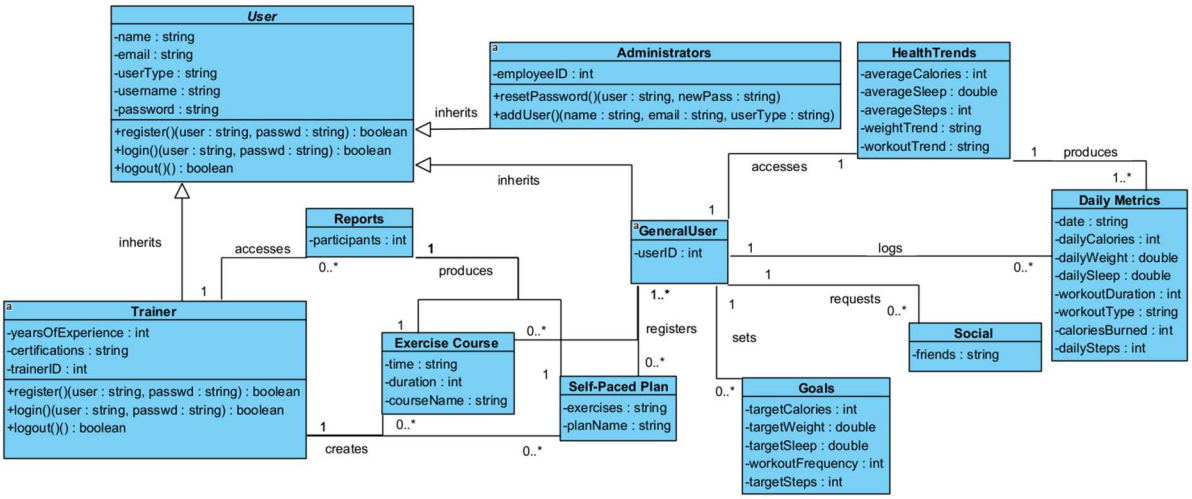
- *trainer* is logged in

Postconditions

- None

Exceptions

- None


User
-name : string
-email : string
-userType : string
-username : string
-password : string
+register()(user : string, passwd : string) : boolean
+login()(user : string, passwd : string) : boolean
+logout()() : boolean
Administrators
-employeeID : int
+resetPassword()(user : string, newPass : string)
+addUser()(name : string, email : string, userType : string)
HealthTrends
-averageCalories : int
-averageSleep : double
-averageSteps : int
-weightTrend : string
-workoutTrend : string
Daily Metrics
-date : string
-dailyCalories : int
-dailyWeight : double
-dailySleep : double
-workoutDuration : int
-workoutType : string
-caloriesBurned : int
-dailySteps : int
Reports
-participants : int
GeneralUser
-userID : int
Trainer
-yearsOfExperience : int
-certifications : string
-trainerID : int
+register()(user : string, passwd : string) : boolean
+login()(user : string, passwd : string) : boolean
+logout()() : boolean
Exercise Course
-time : string
-duration : int
-courseName : string
Self-Paced Plan
-exercises : string
-planName : string
Goals
-targetCalories : int
-targetWeight : double
-targetSleep : double
-workoutFrequency : int
-targetSteps : int
Social
-friends : string
inherits
inherits
inherits
accesses
accesses
produces
produces
logs
requests
registers
sets
creates
1
0..*
1..*

Summary of JUnit Testing

Most of the functions have been covered with JUnit testing. Approximately 75% of functions in the controller (which call all other functions) have been tested. SQLExceptions were difficult to work around because they often created dialogs in the GUI system which we were not testing. JUnit testing was a little difficult because of the amount of functions that we were working with. If we had more time, testing would have been a little easier to manage and hopefully would have produced a more polished result.

Test Cases

TC ID# 1

Scenario/Condition:

User successfully enters exercise and description. Exercise is then visible on the User's "Excercises" page and can be accessed from the remote database.

Test inputs:

String exerciseName, String exerciseDescription

"Squats", "4 sets of 10"

Expected Result:

When the database is queried for the Exercise WHERE NAME = 'Squat', appropriate Exercise Object is returned.

Actual Result (can be left blank if not tested yet)

Correct exercise object is returned

Assigned tester name(s): Carter

TC ID# 3

Scenario/Condition:

User enters exercise without a name

Test inputs:

String exerciseName, String exerciseDescription

"", "4 sets of 10"

Expected Result:

Function call throws NoExerciseNameException, user is notified and given the chance to re-enter a name.

Actual Result (can be left blank if not tested yet)

Correct exception is thrown, user is notified

Assigned tester name(s): Carter

TC ID# 4

Scenario/Condition:

User enters an password that does not meet requirements in registration

Test inputs:

String username, String password

“Tester101”, “badpass”

Expected Result:

Function call throws IllegalArgumentException, user is notified and given the chance to re-enter a password.

Actual Result (can be left blank if not tested yet)

Function call throws IllegalArgumentException, user is notified and given the chance to re-enter a password.

Assigned tester name(s): Carter

TC ID# 4

Scenario/Condition:

User enters an username that does not meet requirements in registration

Test inputs:

String username, String password

“bad”, “StrongPass@123”

Expected Result:

Function call throws IllegalArgumentException, user is notified and given the chance to re-enter a password.

Actual Result (can be left blank if not tested yet)

Function call throws IllegalArgumentException, user is notified and given the chance to re-enter a password.

Assigned tester name(s): Carter

TC ID# 6

Scenario/Condition:

User logs in with an invalid password

Test inputs:

String username, String password

“Carter_Lewis”, “1”

Expected Result:

Function call throws IncorrectPasswordException, user is notified and given the chance to login again.

Actual Result (can be left blank if not tested yet)

Function call throws IncorrectPasswordException, user is notified and given the chance to login again.

Assigned tester name(s): Carter

TC ID# 7

Scenario/Condition:

User logs in with an unregistered username

Test inputs:

String username, String password

“carterlewis”, “1”

Expected Result:

Function call throws UserNotFoundException, user is notified and given the chance to login again.

Actual Result (can be left blank if not tested yet)

Function call throws UserNotFoundException, user is notified and given the chance to login again.

Assigned tester name(s): Carter

TC ID# 8

Scenario/Condition:

User registers correctly and submits empty onboarding form

Test inputs:

String username, String password

“Tester1001”, “Password@123”

Expected Result:

Function throws no exceptions, all inputted information is stored in userInfo in the database.

Actual Result (can be left blank if not tested yet)

Function throws no exceptions, all inputted information is stored in userInfo in the database.

Assigned tester name(s): Carter

TC ID# 9

Scenario/Condition:

User logs in correctly

Test inputs:

String username, String password

“Carter_Lewis”, “Password@123”

Expected Result:

Function throws no exceptions, UserDashboard is displayed

Actual Result (can be left blank if not tested yet)

Function throws no exceptions, UserDashboard is displayed

Assigned tester name(s): Carter

TC ID# 10

Scenario/Condition:

Trainer logs in correctly

Test inputs:

String username, String password

“NewTrainer”, “Password@123”

Expected Result:

Function throws no exceptions, TrainerDashboard is displayed

Actual Result (can be left blank if not tested yet)

Function throws no exceptions, TrainerDashboard is displayed

Assigned tester name(s): Carter

Team Contributions

Total hours spent working on the project, per developer:

Carter: 48/343 = 14%

Emily: 53/343 = 15%

Josh: 65/343 = 19%

Kiera: 51/343 = 15%

Lawson: 52/343 = 15%

Noah: 74/343 = 22%

Main contributions:

- Carter was the first to implement GUI scenes for our project, laying the foundation for the rest of our scenes. In addition, he did a large percentage of the testing for our project. His work was invaluable and he helped us find many errors and bugs in our code early on.
- Emily was the librarian and did her job wonderfully, making sure we had meeting notes and the documentation of the code in order. Additionally, Emily pioneered all of our graphs, doing plenty of research to figure out the best way to approach report graphs.
- Josh as the project manager was in charge of the majority of the project deliverables. He led the team in terms of casting vision and providing motivation and accountability. Additionally, he was the last quality control before we put something out.
- Kiera did a good portion of the refactoring necessary to make the code more readable, better designed, and more easily tested. She was often in charge of merging and made sure ideas didn't conflict. Additionally, she helped with the project deliverables after Iteration 2.
- Lawson developed a large portion of our exercise functionality, pulling together lots of people's ideas to make a simple solution. He was also essential in making sure that our code followed the SOLID, DRY, and KISS principles. Lawson was also the most adept at using GitHub and often helped the team navigate the new software.
- Noah was our database master and spent a large amount of time configuring our multi-accessible database. Additionally, Noah developed the majority of the trainer's functionality and consistently brought new ideas to the team.

Tailscale + SSH Tunnel Setup Guide for FitnessDB

1. Overview

This guide walks you through setting up secure access to the group MySQL database using Tailscale and SSH tunneling.

You will connect to Noah's Ubuntu-hosted database from your local machine, allowing you to both access the Java project and run SQL commands via the shell.

2. Requirements

- Your own GitHub account (used to log into Tailscale)
- A Tailscale client installed on your machine (Windows/macOS)
- A terminal (PowerShell for Windows, Terminal for macOS)

3. Step-by-Step Setup

STEP 1: Install and Set Up Tailscale

1. Go to https://tailscale.com/download
2. Download and install Tailscale for your operating system.
3. https://login.tailscale.com/uinv/i06c781e2d13593ae
3. Click link and sign in using your GitHub in app.
4. Wait for Noah to approve your device (he'll get a notification). [shouldn't be necessary anymore].

STEP 2: Pick Your Local Port

Choose a unique local port to avoid conflicts:

- Noah: 3307
- Emily: 3308
- Kiera: 3309
- Josh: 3310
- Lawson: 3311
- Carter: 3312

STEP 3: Create the SSH Tunnel

Open your terminal and run the following command, replacing <port> with your port:

Tailscale + SSH Tunnel Setup Guide for FitnessDB

ssh -L <port>:localhost:3306 projecttunnel@100.68.92.66

Example (Noah): ssh -L 3307:localhost:3306

projecttunnel@100.68.92.66

Enter the SSH password when prompted:

Password: maninthemirror

Keep this terminal open, it establishes the tunnel to the database. Now run the program, select y to use the mysql database, and type in your port 33XX to use the app.

4. Connecting to SQL Shell (Optional) [FOR DEVS]

To interact with the MySQL database

directly:

1. Ensure your SSH tunnel is running.

2. In the active shell run:

```
mysql -u fitnessuser -p
```

3. Enter the MySQL password:

Password: strongpassword123

You will now be in the MySQL shell and can run SQL commands like:

USE fitnessdb

SHOW TABLES;

SELECT * FROM users;

5. Tips

- If the SSH tunnel closes, either run again or add the parameters -N and -f

- Don't use port 3306 locally - it's reserved by your own system.

Tailscale + SSH Tunnel Setup Guide for FitnessDB

- Never share the SSH or SQL password outside of the team.
- Don't close the terminal window running the tunnel while you're using the DB.

6. Help

Problem: SSH connection fails

Solution: Make sure you're connected to Tailscale and your device was approved.

Problem: MySQL won't connect

Solution: Ensure you used the correct port in the mysql command and that the tunnel is active.

Problem: SSH tunnel closes immediately

Solution: You may have hit Ctrl+C or closed the terminal. Re-run the SSH tunnel command.

CardioB User Manual

By: Trello Fellows

Introduction

What is CardioB?

CardioB is a health app that tracks health and lifestyle activities. Our app provides user friendly functionality for fitness trainers to create and share custom self-paced workout plans and host live classes! As for users hoping to partake in these revolutionary classes, general users have easy access to classes and are able to log each workout and track their progress. Ready to get in your CardioB?

Features:

Features for the Users:

Users can view a record of past data recorded and document health data.

Class Registration

- ➢ Users can sign up for classes created by trainers
- ➢ Users can report data to their trainer
- ➢ The user can leave a class

Goal Tracking

- ➢ Users can view if they are on track towards goals
- ➢ Users can set custom goals for themselves

Reminders

- ➢ Users can create and set reminders for daily tasks

Features for the Trainers:

Exercise Plans

- ➢ Trainers can create exercise plans for users

Scheduled Classes

- ➢ Trainers can create scheduled classes that include date, time, and days of the week the class occurs, length of class, recommended fitness level, and required equipment
- ➢ Trainers can limit the number of participants in the class

Class Data

- ➢ Trainers can view information about their plans and classes

Install Instructions:

- Go to https://josh-ful.github.io/cardioB_3471/ and click “Download CardioB” button
- Navigate to zip file on your computer
- Decompress zip file in desired location
- Open the project in an IDE, and follow the UserGuideSetupForKirk pdf.

How to use as a User:

Login/Register Page

- As a new user, click the “Register” button and input your desired login according to the constraints. Follow the pop-up instructions and login to the app.

User Dashboard

- **Once successfully logged in**, you are brought to the dashboard, which displays all current metrics.
- **Click on your username in the top lefthand side** and select from the menu items which course of action you would like to take.
- **Clicking on “Daily Metrics”** takes you to a new page of your health information. Click “Add Daily Metrics” to take you to a pop up screen where you can log health information for any specific date. Pressing save will update your information and each of the graphs.
- **Clicking on “Classes”** to take you to a page that has a list of classes that you have registered for. Click the “Register for New Class Button” to register for another class.
    - This will take you to a list of all the available classes. Filters include type of class and a search bar.
    - Register for a class by clicking the “Register” button to the right of the name. Your registered class should show up on the “Classes” page.
    - Once you are registered for a class, press continue or join to participate in the class.
- **Click “Exercise Log”** to take you to a page that has a list of workouts that you have done and recorded. Click “Add Workout!” to log another workout and fill in the information and click “Submit” to add it to the list.
- **Click “Profile”** to take you to a page that displays all profile information. Click “Edit Information” to add new information or modify existing information. Click “Set Goals” to input goals for your weekly metrics. These goals will show up on the “Current vs. Goal” table. There is also a “Reset Password” button to reset your password.
- **To log out,** click the menu bar in the top lefthand corner and click “log out”.

How to use as a Trainer:

Login/Register Page

- ➢ As a new trainer, click the “Register” button and input your desired login according to the constraints. Follow the pop-up instructions and login to the app.

Trainer Dashboard

- ➢ **Once successfully logged in**, you’re brought to the dashboard, which displays all current metrics.
- ➢ **Click on your username in the top lefthand side** and select from the menu items which course of action you would like to take.
- ➢ **Click “View Classes”** to take you to a new page displaying a list of your classes.
    - ○ **Click “Create New Class”** to display a pop up to enter information about your new class.
- ➢ **Once you’ve successfully created a group class**, click “Host Class” to display a list of your group classes.
    - ○ Click the Host button next to the class desired to start.
    - ○ To end the class, click “End Class”
- ➢ **Click “View Reports”** to take you to a new page displaying all your classes. Click the “View Report” button next to the desired class.
    - ○ This will display a graph of registrations over time and session join counts on specific dates
- ➢ **To log out,** click the menu bar in the top lefthand corner and click “log out”.

Developers

Josh Fulton - Project Manager
Noah Mathew - Requirements Engineer
Carter Lewis - Design Engineer
Kiera Shepperd - Quality Assurance
Lawson Hale - Quality Assurance
Emily Wokoek - Project Librarian

Meeting 1

1/28/25
11:00am
Cashion 308

Objectives: work on lab so we can work on group portion in lab

Notes for next time:

Meeting 2

1/30/25
2:00pm
Cashion 308

Objectives:

- Work on and finish group portion of lab 2
    - learn to use a CASE tool, Visual Paradigm, to draw a use case diagram
- Work on and finish project deliverable 2
    - Use case diagram with 6-7 use cases in **brief form**

Notes:

- Using Noah's display to make Use Case diagram
- Made shared folder for project with space to brainstorm use cases
- Establish a project vision(what are we making)
- Made a doc for use case briefs
- Research brief format
- Establish requirements
- Josh created and shared GitHub project
- Consolidated all docs into one

Notes for next time:

- Ask Prof Kirk if admin/user/"coach" actors scenes will connect or if they are separate apps

Meeting 3

2/4/25
11:00am
Cashion 308

Objectives: work on lab so we can work on group portion in lab

Notes for next time:

Meeting 4

2/6/25
2:00pm
Cashion 308

Objectives:
Finish Use Cases and Business Vision Doc.
Have everyone join the GitHub and get used to logging hours and issues in GitHub

Notes for next time:

Meeting 5

2/11/25
11:00am
Cashion 308

Objectives: work on lab so we can work on group portion in lab, review what needs to be done for the week

Notes for next time:
Establish expectations for the group in terms of equal workload, personal deadlines, and logging hours

Meeting 6

2/13/25
2:00pm
Cashion 308

Objectives:

- Create fully dressed use cases (2 per person)
    - Create SSDs for each (2 per person)
- Create wireframe(initial draft)
- Create operation contracts
- Create domain model
- Be mostly done with iteration 1(start basic planning for presentation)

Notes for next time:

Meeting 7

2/18/25
11:00am

Cashion 308

Objectives:

- Work on getting use cases done for Iteration 1
- Make sure everyone is on the same page for formatting ssds and operation contracts

Notes for next time:

- Find out if there other requirements for the wireframe before turning in iteration 1

Meeting 8

2/20/25
Cashion 308

Objectives:

- Making plan for presentation
- Finalizing iteration 1
- Implementing project deliverable 4 feedback

Notes for next time:

-

Meeting 9

2/25/25
11:00am
Cashion 308

Objectives:

- Final adjustments on presentation
- Finalized roles for presentation
- Discuss project deliverable and iteration 2

Notes for next time:

- Edit ssd models
    - Remove login function as it will be a precondition to every one of our use cases

Meeting 10

2/27/25
1:00pm
Cashion 308

Objectives:

- Project deliverable 6
    - Discuss how we are going to code and separate roles
    - Fix errors from previous deliverable
    - Research GRASP and what is needed for the next deliverable
- Set Goal for iteration 2:
    - Get all UI done
    - Basic level implementation
    - Implementing user fully
    - Save database interaction for iteration 3

Notes for next time:

Meeting 11

3/4/25
11:00AM
Cashion 308

Objectives:

- Discuss Project Deliverable 7
- Work on individual lab

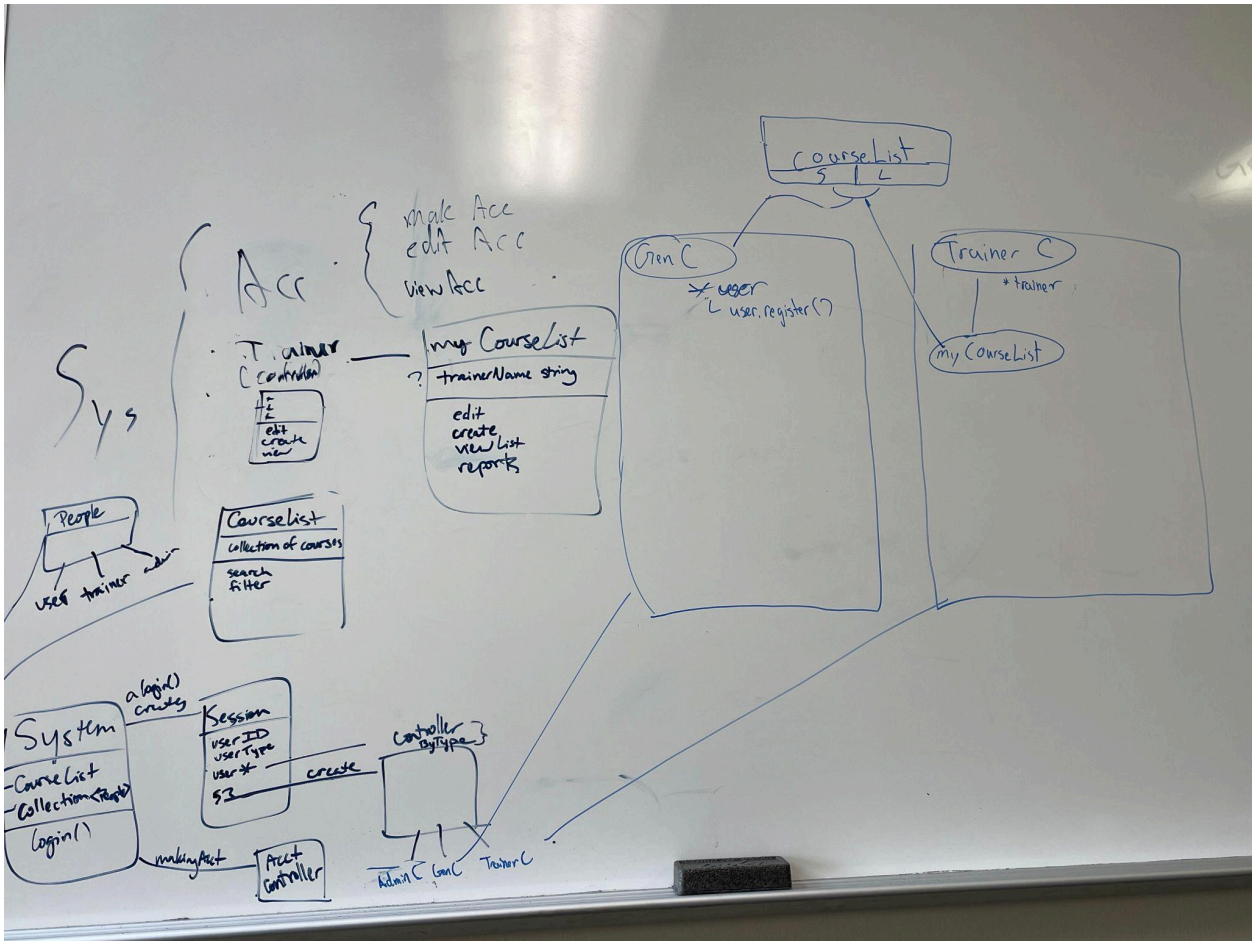
Notes for next time:

Meeting 12

3/6/25
2:00PM
Cashion 308
Objectives:

- Discuss program structure

- Mystical courseList GLOBAL



- **System** contains
  - courseList
    - Collection of courses
    - Search
    - filter
  - collection of people
    - People is class(user,trainer,admin)
  - login()
    - Creates **session** contains:
      - userID
      - userType
      - user*
      - constructor
        - Creates controller by type
          - adminC
          - GenC
            - *user
              - user.register()
          - trainerC

- Edit create new
- *trainer
- **myCourseList**
  - Edit
  - Create
  - Viewlist
  - Reports

Notes for next time:

Meeting 13

3/18/25
11:00AM
Cashion 308

Objectives:
- Discuss Project Deliverable 8
- Splitting up work
- Working on individual labs

Notes for next time:

Meeting 14

3/20/25
11:00AM
Cashion 308

Objectives:
- Write a basic framework in Swing for Iteration 2(progress 1)

Notes for next time:

Meeting 15

3/21/25
2:00PM
Cashion 308

Objectives:
- Created base for project
- Basic scenes

Notes for next time:

Meeting 17

3/25/25
11:00AM
Cashion 308

Objectives:

- Start documentation
- Junittests
- Merged all members commits
- Discussed how we should implement GRASP concepts in the future
-

Notes for next time:

- Ask how we should implement GRASP concepts in the future
- Start serious documentation
- Organize

Meeting 17

3/27/25
2:00PM
Cashion 308

Objectives:

- Discuss project deliverable
- Merge all branches
- Begin serious documentation(JDOC)

Notes for next time:

-

Meeting 18

4/10/25
2:00PM
Cashion 308

Objectives:

- Video
- Discuss project deliverable 11 and how to implement Trainer and Admin

Notes for next time:

Meeting 19

4/15/25
11:00AM
Cashion 308

Objectives:

- Presentation
- Ensured javadoc complete

Notes for next time:

- Have to do final assigning of tasks by sunday

Meeting 20

4/17/25
2:00PM
Cashion 308

Objectives:

- Split up assignments for rest of semester
- Project deliverable 12
- GRASP(ensuring low coupling, high cohesion)

Notes for next time:

Meeting 21

4/22/25
11:00AM
Cashion 308

Objectives:

- Update on what has been done
- Merge all branches
- Prepare for project deliverable

Notes for next time:

Meeting 22

4/29/25
11:00AM
Cashion 308

Objectives:

- preparing for iteration 3
- Assigning members with features
-

Notes for next time:

Meeting 23

5/1/25
2:00PM
Cashion 308

Objectives:

- Working on iteration 3
-

Notes for next time: