itucsdb Documentation

Release 1.0

postITU

Dec 30, 2016

CONTENTS

Team postITU

Members

- Sıddık Açıl
- Rumeysa Bulut
- Alim Özdemir
- Halit Uyanık
- Ömer Faruk İNCİ

This project is a social networking website that enables its users to share photos momentarily and interact with each other.

- Users can register and sign in to this website.
- Users can post their photos.
- Users can apply image filtering to their photos to create effects.
- Users can share custom filters with other users.
- Users can comment on photos.
- Each photo has a comments section underneath itself.
- A comment can be edited or deleted by commenter.
- An image poster can tag location information to images.
- An image may have more than one location tags.
- A poster can tag other users to specific locations on images.
- A tag can be deleted or edited by tagged user.
- Users can create new groups with the people they followed.
- Users can organize events.

CHAPTER

ONE

INSTALLIATION GUIDE

Following guide explains how to install and ready the postITU project.

After installing all the requirements user/developer can execute the website from the main directory with:

```
python server.py
```

Requirements:

- *Python*
- *Flask*
- *Psycopg2*
- *PostgreSQL*
- *flask_wtf*
- *googlemaps*
- *pillow*
- *passlib*

1.1 Python

Python can be found from the link:

https://www.python.org/downloads/

1.2 Flask

Flask can be installed from command line via:

pip install Flask

You need a working linux, or bash integrated windows command line to use pip.

1.3 Psycopg2

Psycopg2 can be installed from command line via:

pip install psycopg2

1.4 PostgreSQL

PostgreSQL is used to work on project locally without having to change the bluemix db all the time.

Can be downloaded from:

https://www.postgresql.org/download/

1.5 flask_wtf

Another dependency which is used in the website.

pip install Flask-WTF

1.6 googlemaps

Googlemaps needs to be integrated to website in order to see locations etc. Details about how to get an api key and use it can be found at the following link:

https://developers.google.com/maps/

1.7 pillow

Another dependency which is used in the website.

pip install pillow

1.8 passlib

Another dependency which is used in the website.

pip install passlib

CHAPTER

TWO

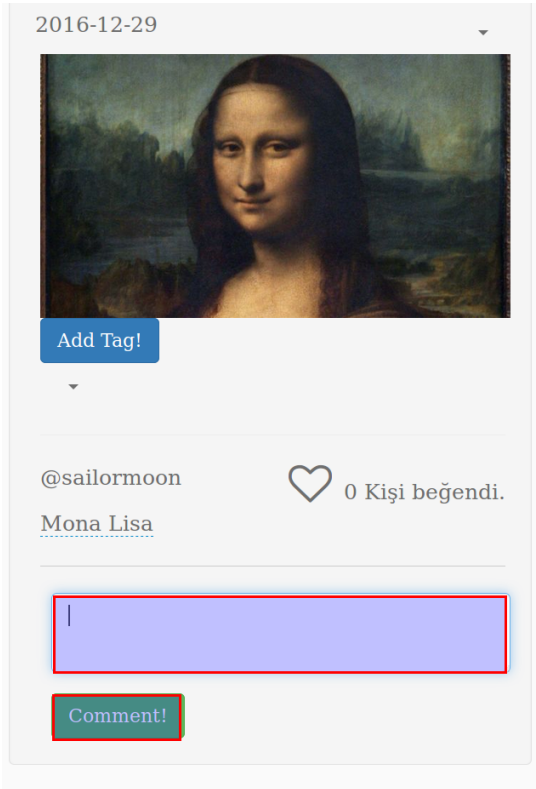
USER GUIDE

2.1 Parts Implemented by Sıddık Açıl

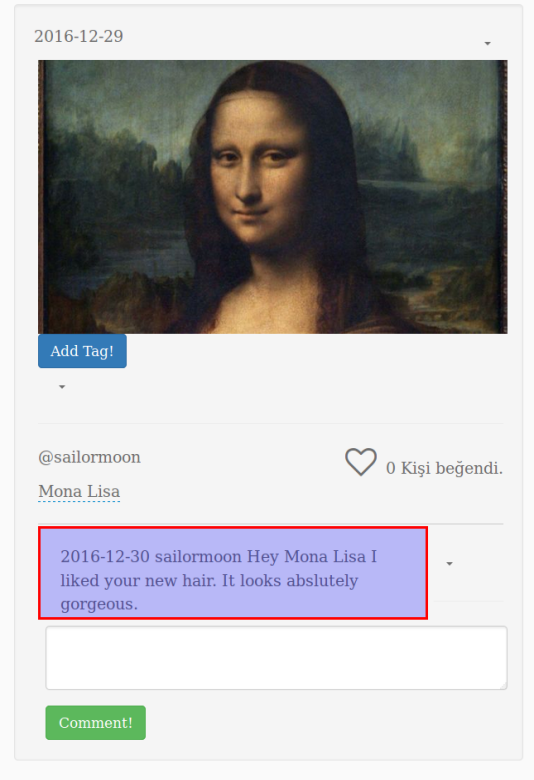
2.1.1 Comments

How to comment?

To comment click and fill the area below and press “Comment!” button.

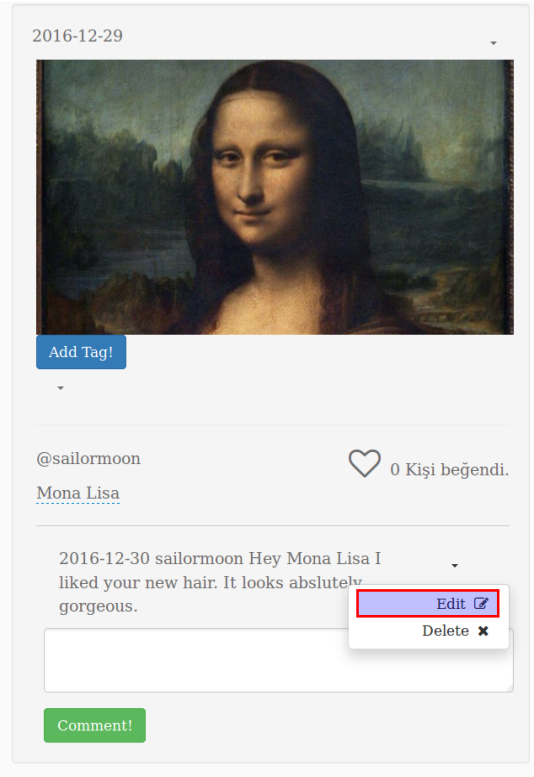


Your comment can be viewed under the image.

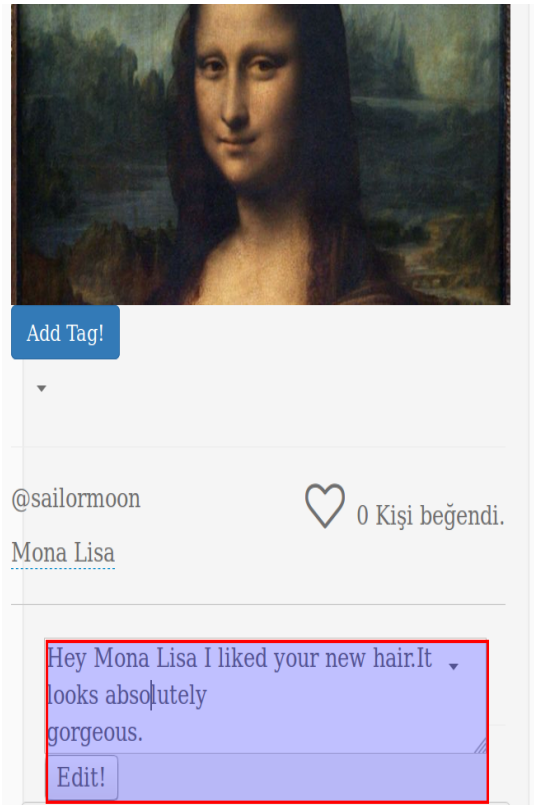


How to edit comments?

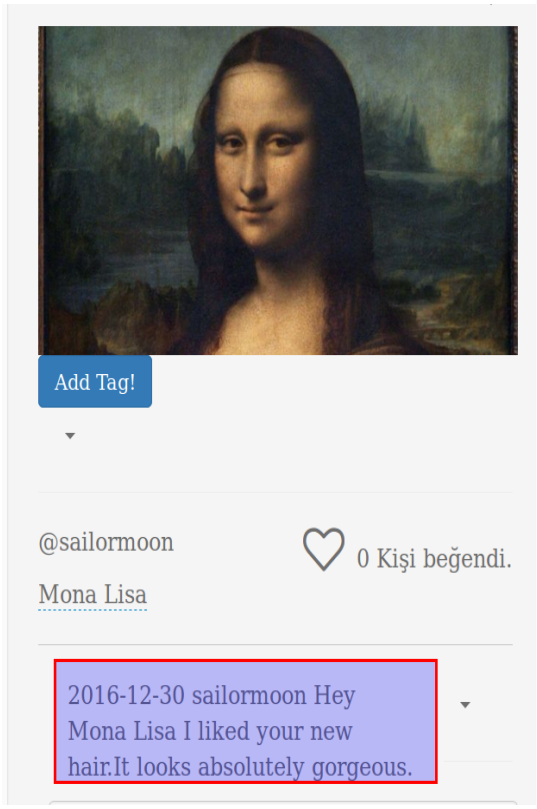
If you have made a mistake while commenting or if you want to add something more you can click the caret next to your comment and click “Edit”



You can edit your comment in the input field and press “Edit!” button to save changes.

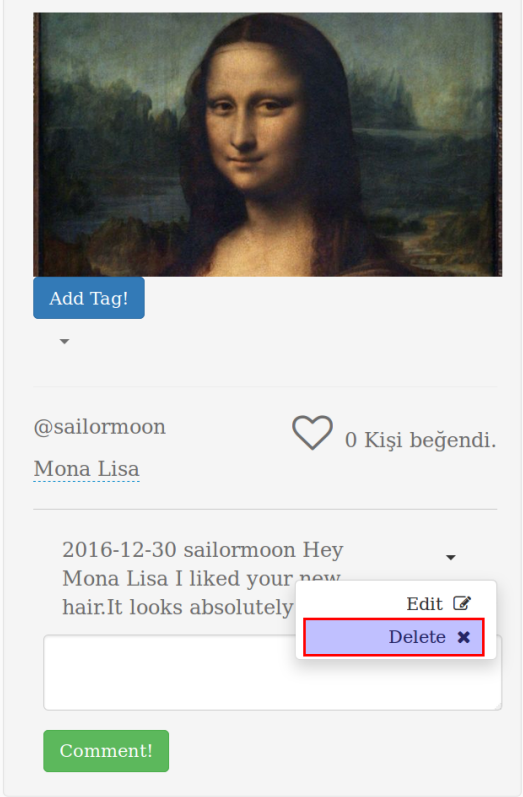


Your comment after the update.

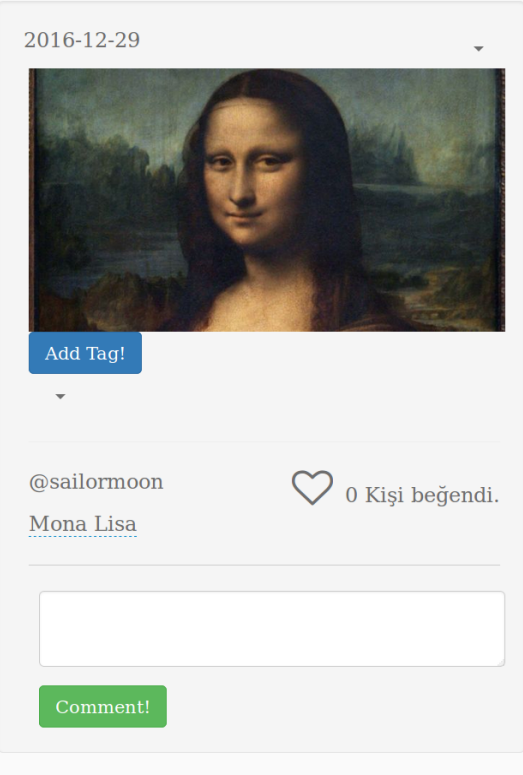


How to delete comments?

If you want to delete a comment you have made click the caret next to your comment and click “Delete”.



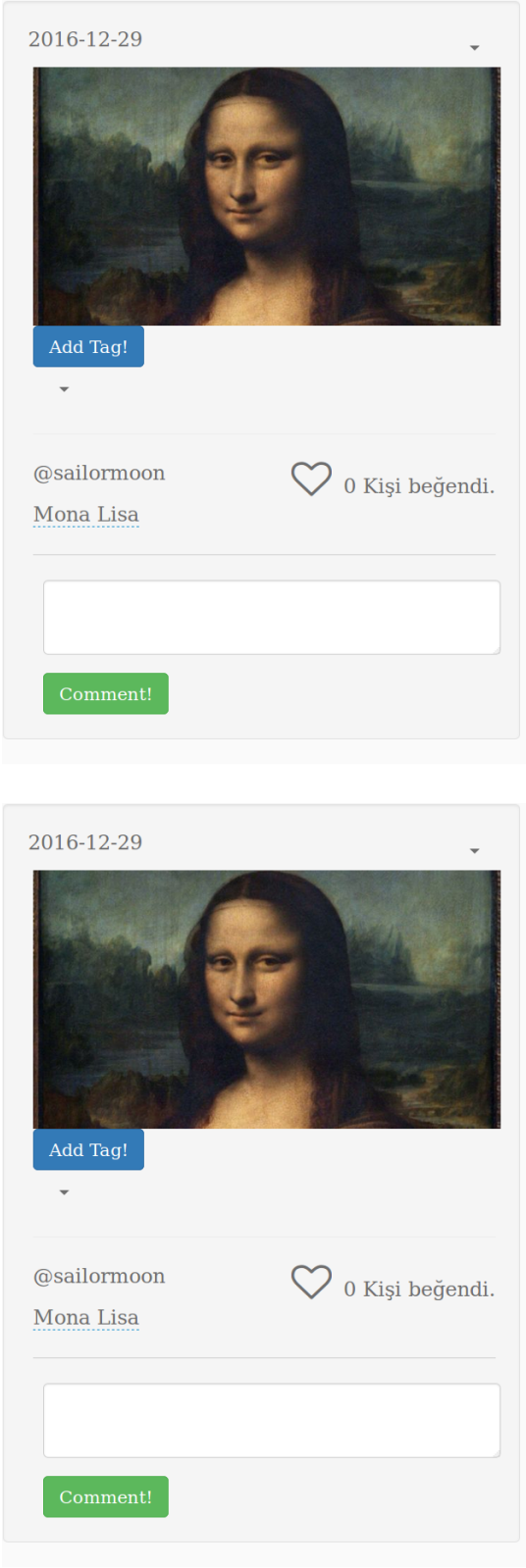
Viola! Your comment is gone.



2.1.2 Reporting Content

How to report a content?

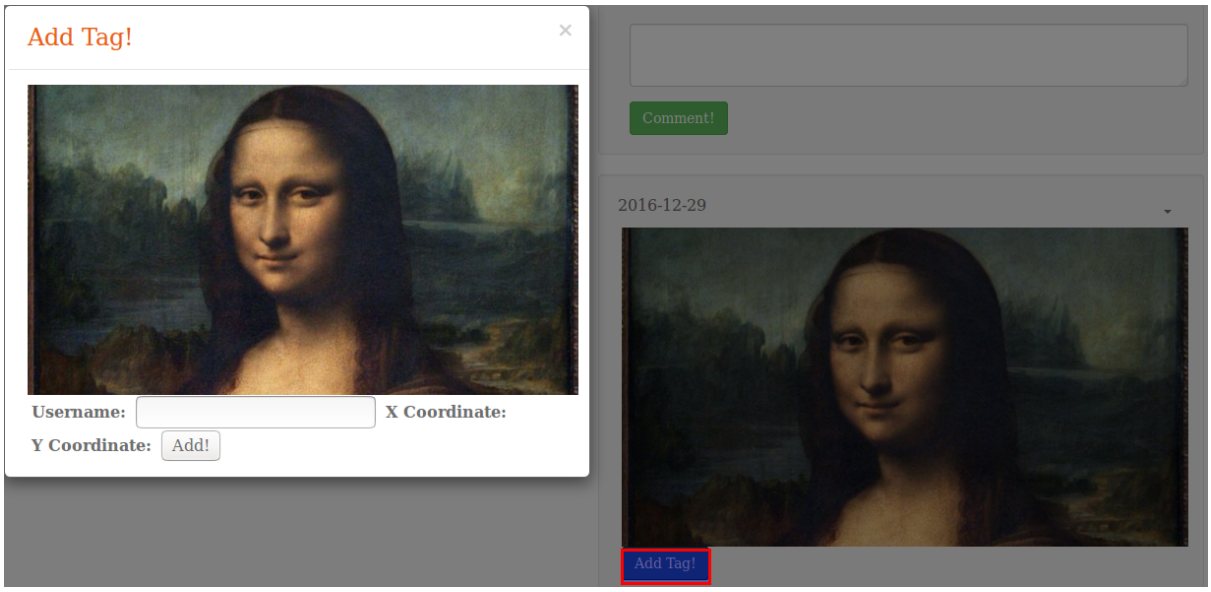
If you find a content unsuitable, you can issue a proposal for deletion by clicking the caret on upper right side of an image then click “Report”.



2.1.3 Image Tagging

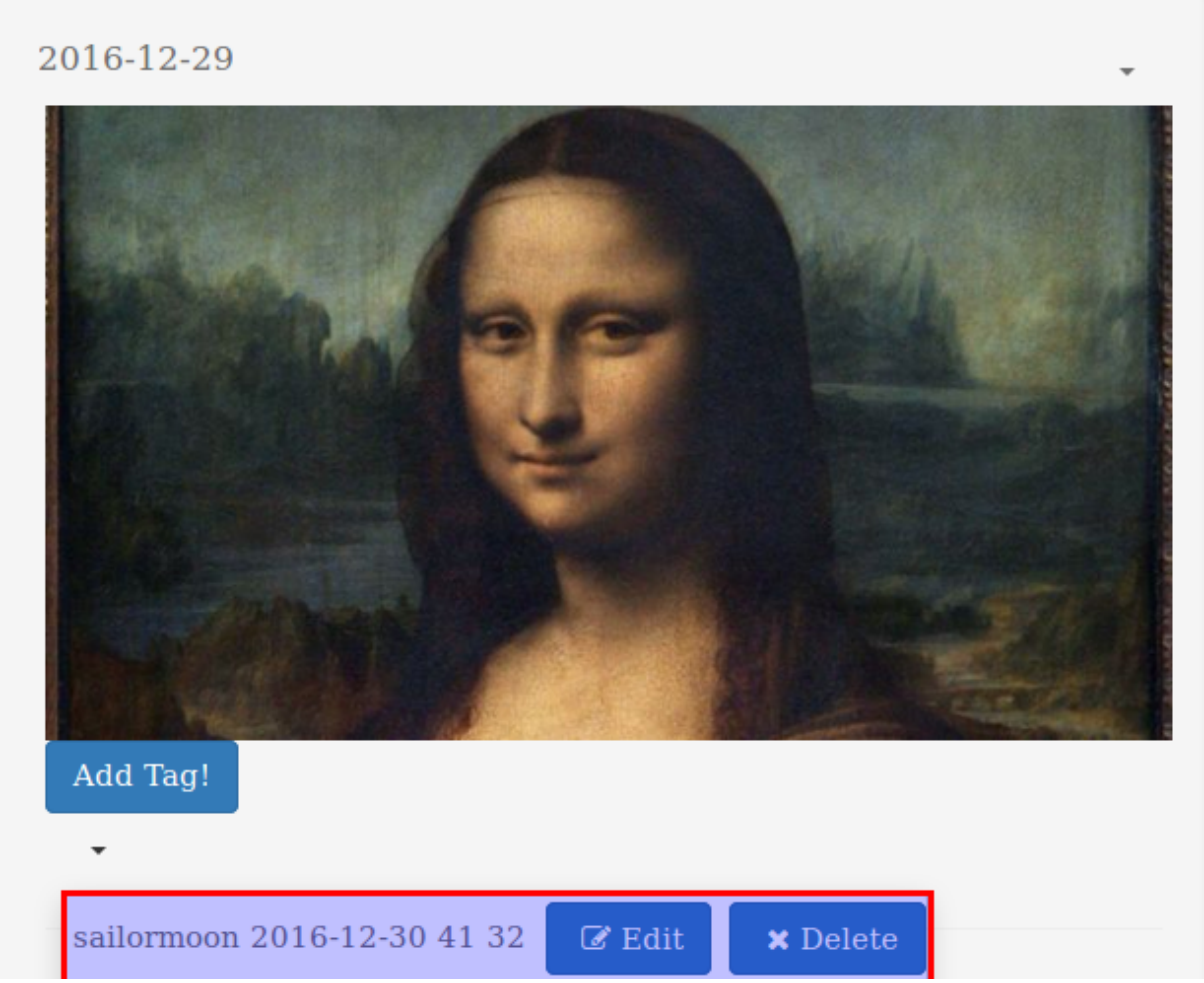
How to tag someone to an image?

If you want to tag someone (including yourself) to an image, click the "Add Tag!" button beneath image.



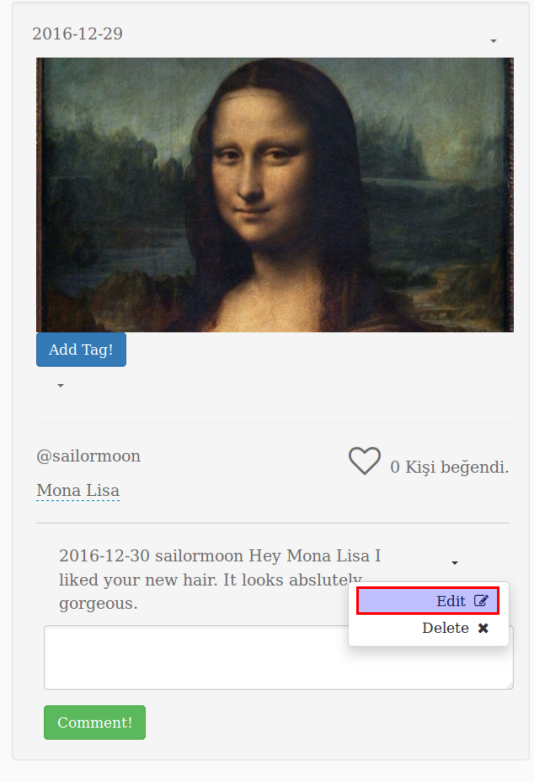
Fill in the name you want to tag and click the position you want to tag the name.

You can view tags of an image by clicking the caret under "Add Tag!" button.

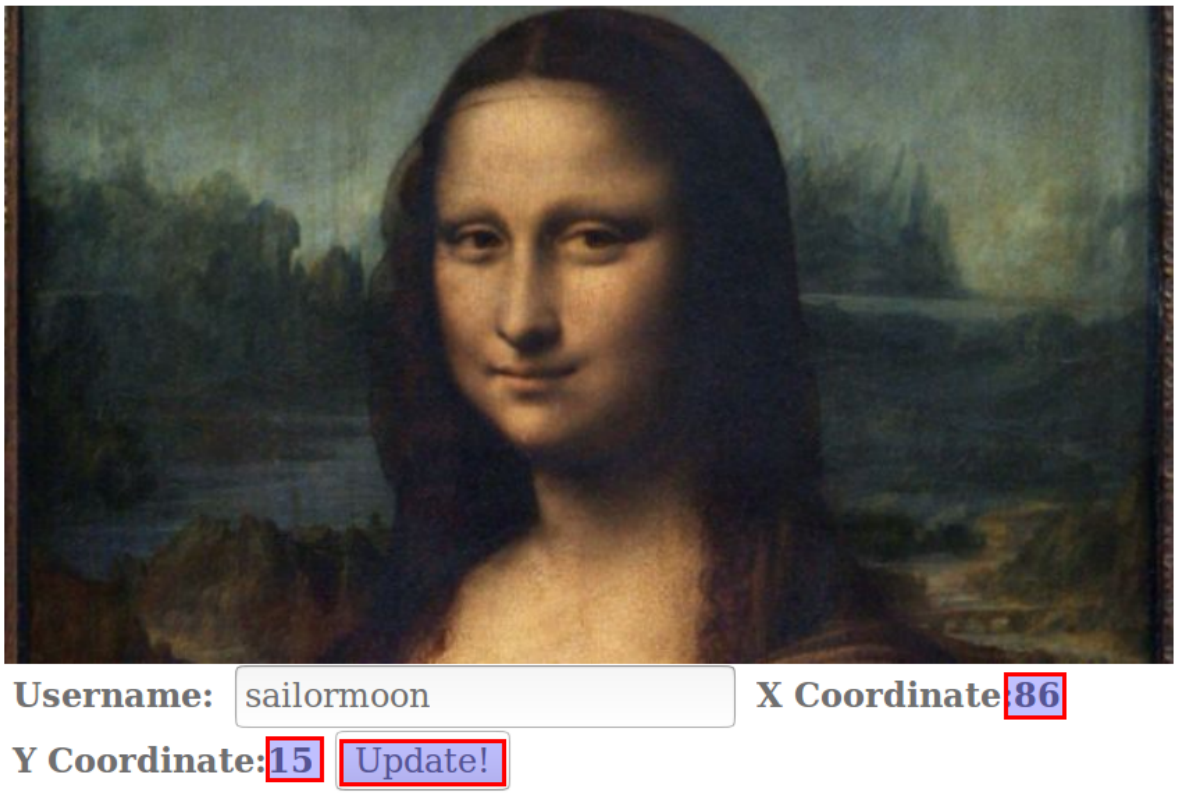


I am tagged to a photo. How do I update this tag?

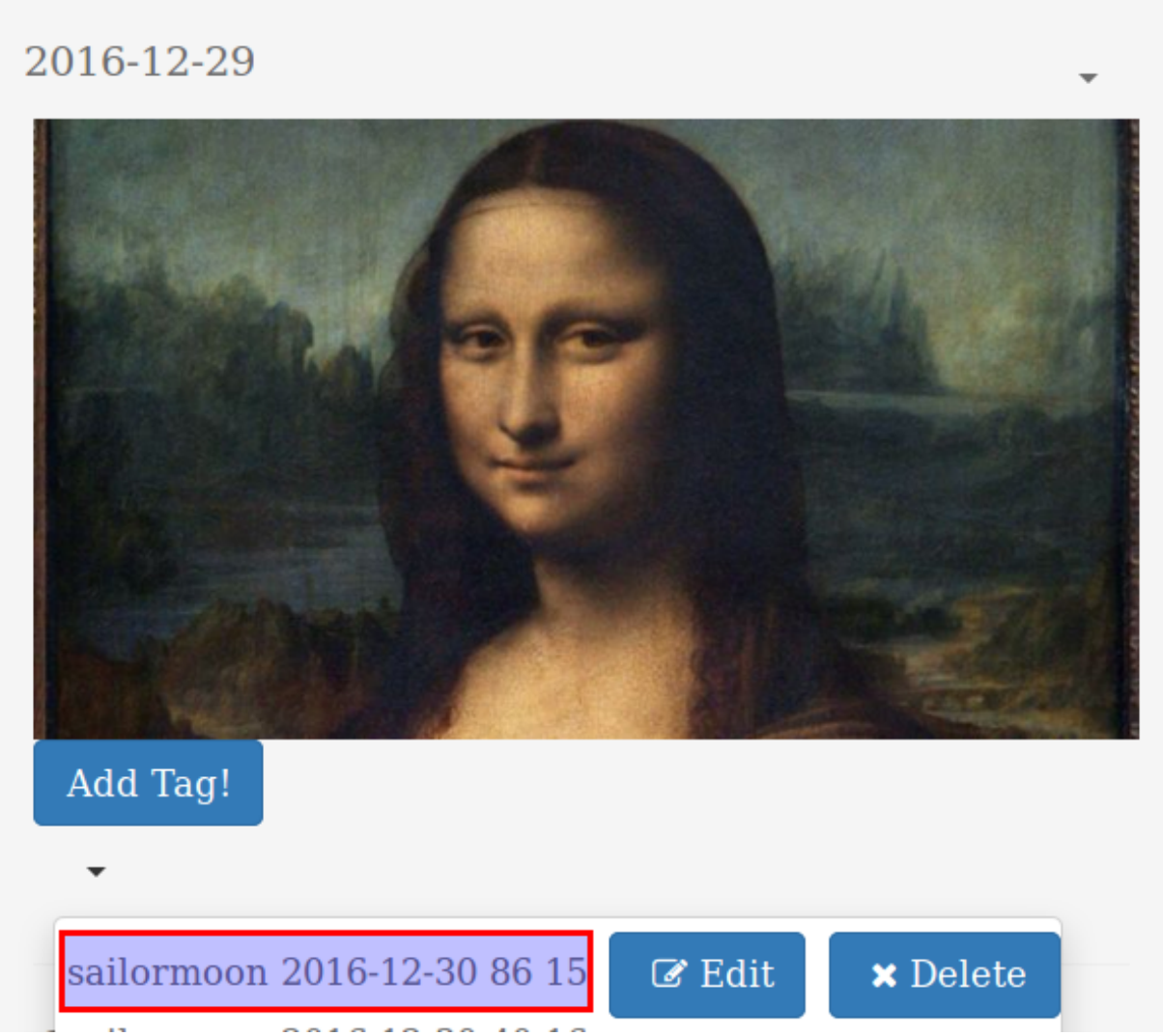
When you click the caret to show the tags of and image, you will find “Edit” button next to tags. If you want to update tag from photo click “Edit”.



Fill the form on the pop-up and click a new place on image if you want to, then press update.

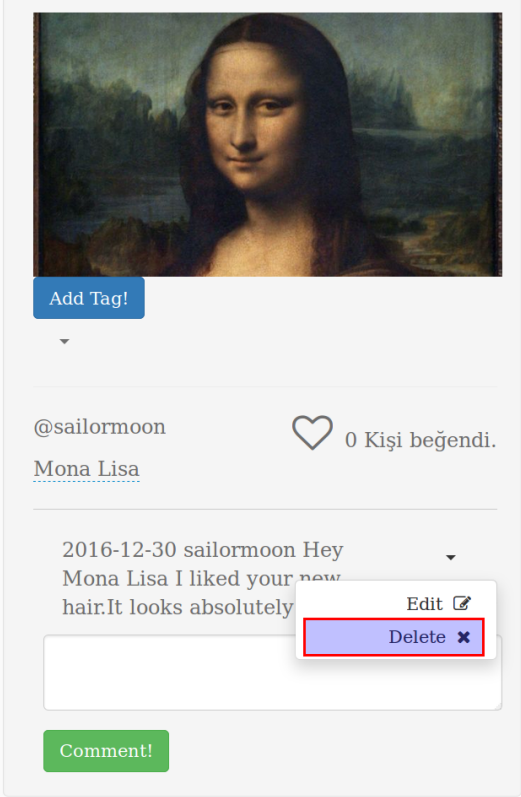


Your changes has been saved.

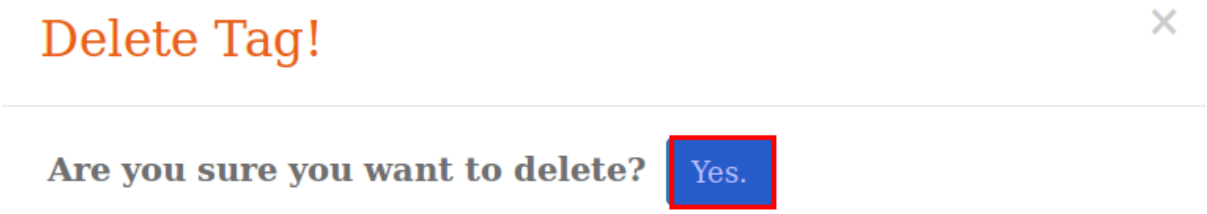


I am tagged to a photo. How do I delete this tag?

When you click the caret to show the tags of and image, you will find “Delete” button next to tags. If you want to remove tag from photo click “Delete”.



A pop-up will show to make sure you want to remove the tag.



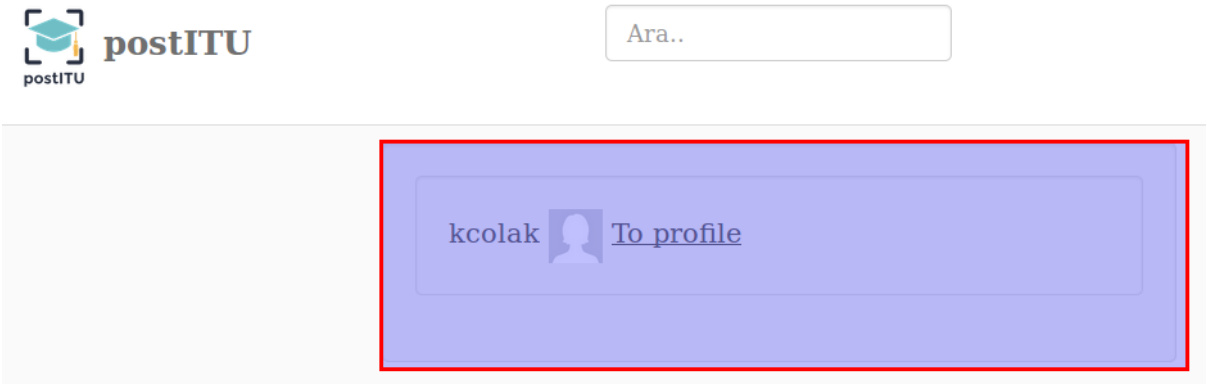
Note: You can only delete/edit tags if you are the user who is tagged.

2.1.4 User Listing

How to search someone?

On the top of the page fill in the "Search" field and press enter.

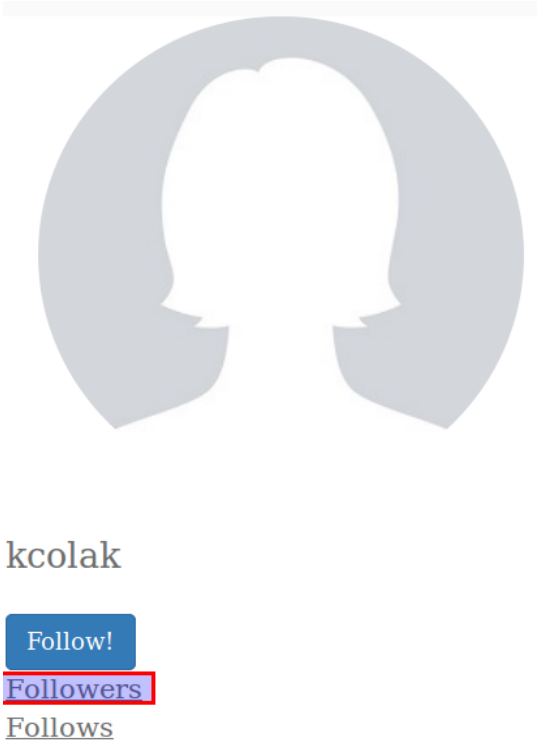
Your search result is.



2.1.5 Following Users

How to view followers?

Click on “Followers” link to view the list of your followers.



How to view people I follow?

Click on “Follows” link to view the list of the people you follow.

kcolak

Follow!
Followers
Follows

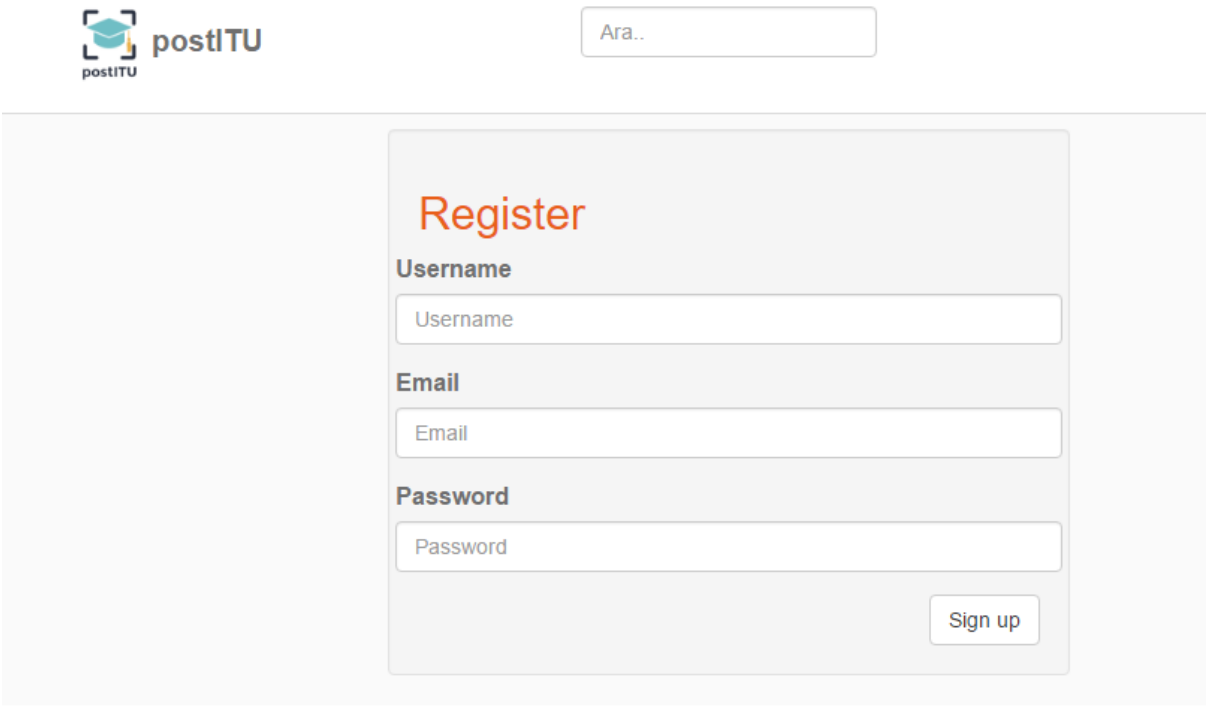
2.2 Parts Implemented by Rumeysa Bulut

2.2.1 User Operations

How can I register?

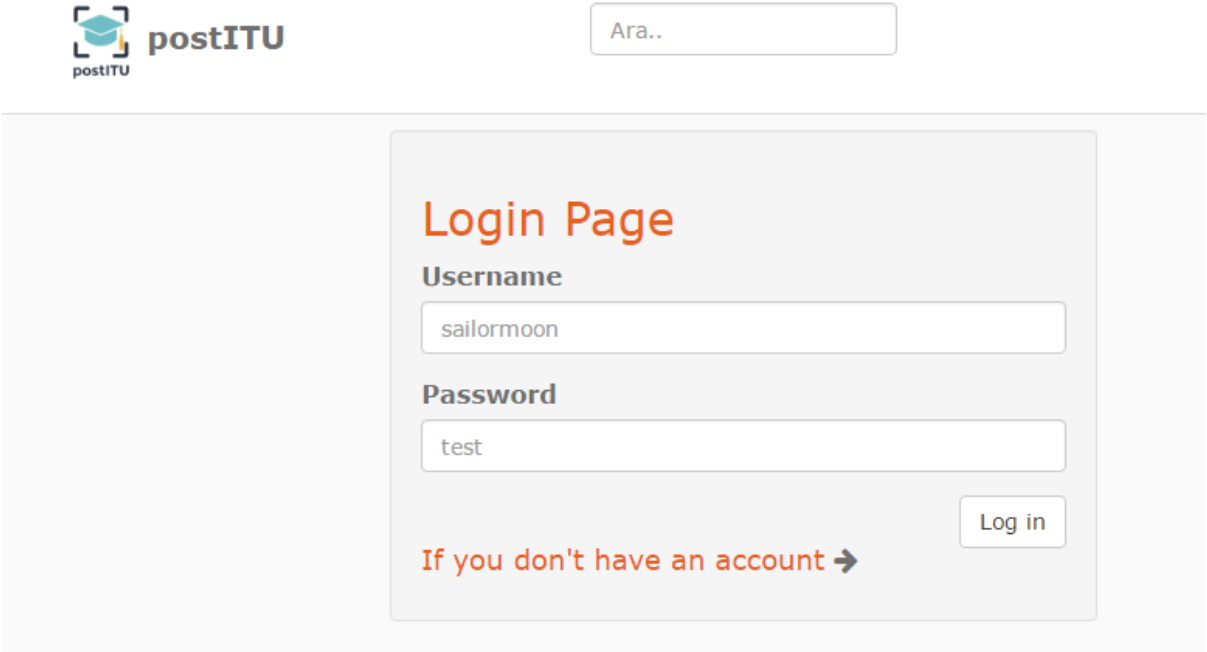
When you open the site, you will see the login page first. There is an option for registering under the login section. If you do not have an account, you can click that arrow sign to reach the sing up page. Also, you can click the dropdown menu, and you will see an icon to direct you to the register page.

Here is a view of register page:



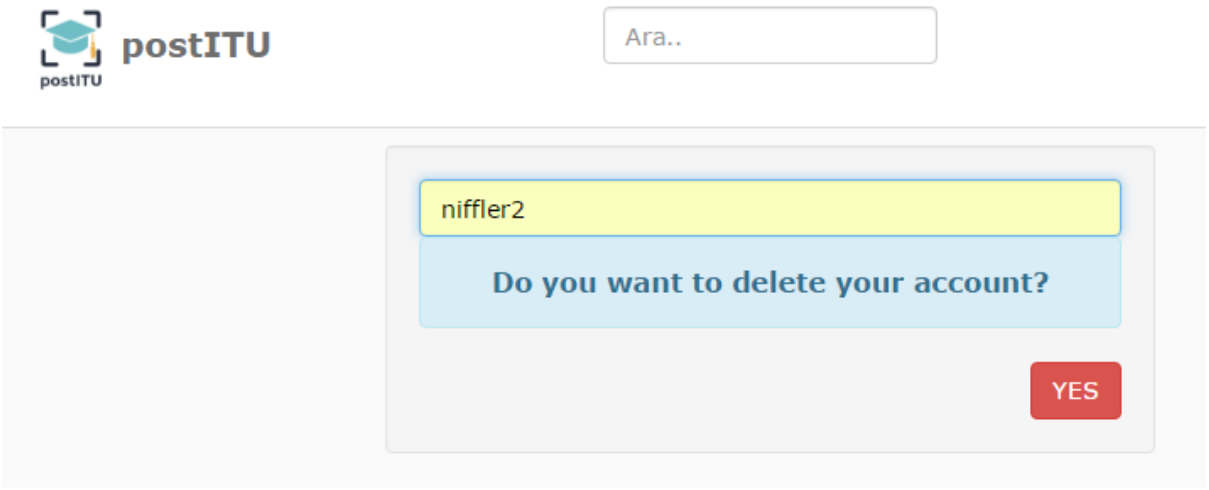
How can I login?

When you open the site, you will see the login page first. You can enter to the site easily by typing your user name and password. Here is a view of login page:



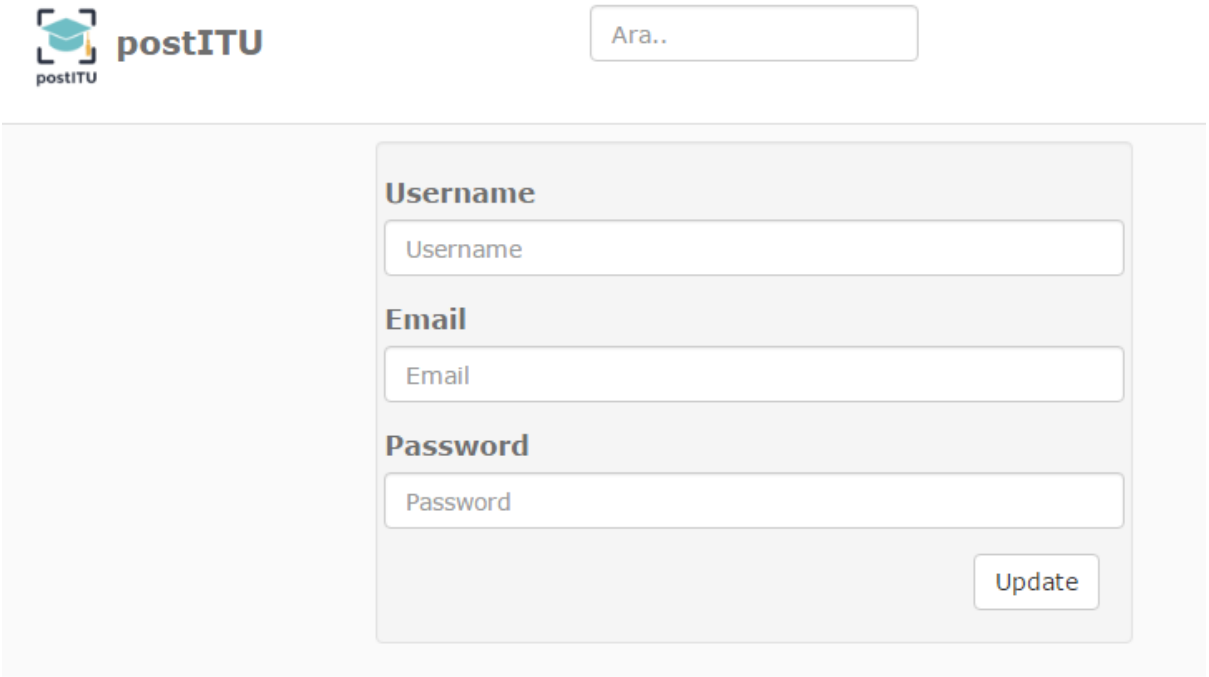
How can I remove my account?

You will see an option which allows you to remove your account on the dropdown menu after you logged in. if you click the cross sign, you will be directed to the remove account page. You can remove your account simply by typing your user name and pressing “yes”. Here is a view of remove account page:



How can I update my information?

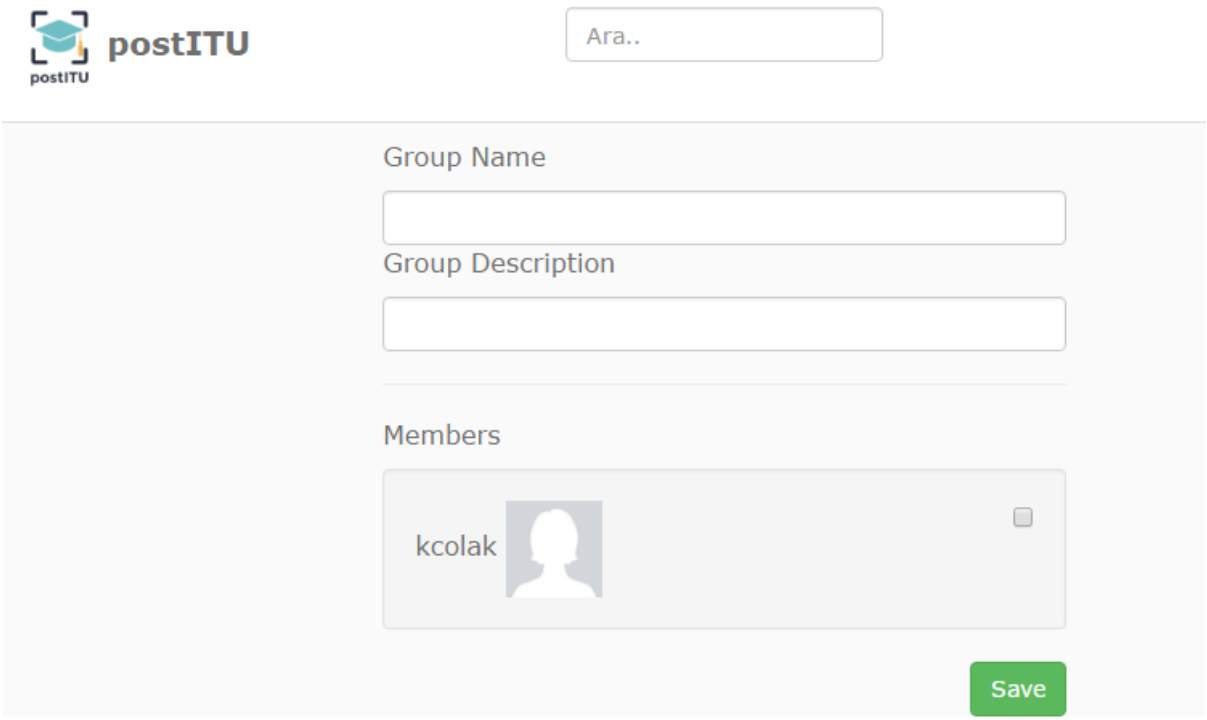
Update account section follows similar steps to removing account section. You will see an option which allows you to update your information on the dropdown menu after you logged in. if you click the refresh sign, you will be directed to the update account page. You can enter your current information to the form in this page. Here is a view of update account page:



2.2.2 User Groups

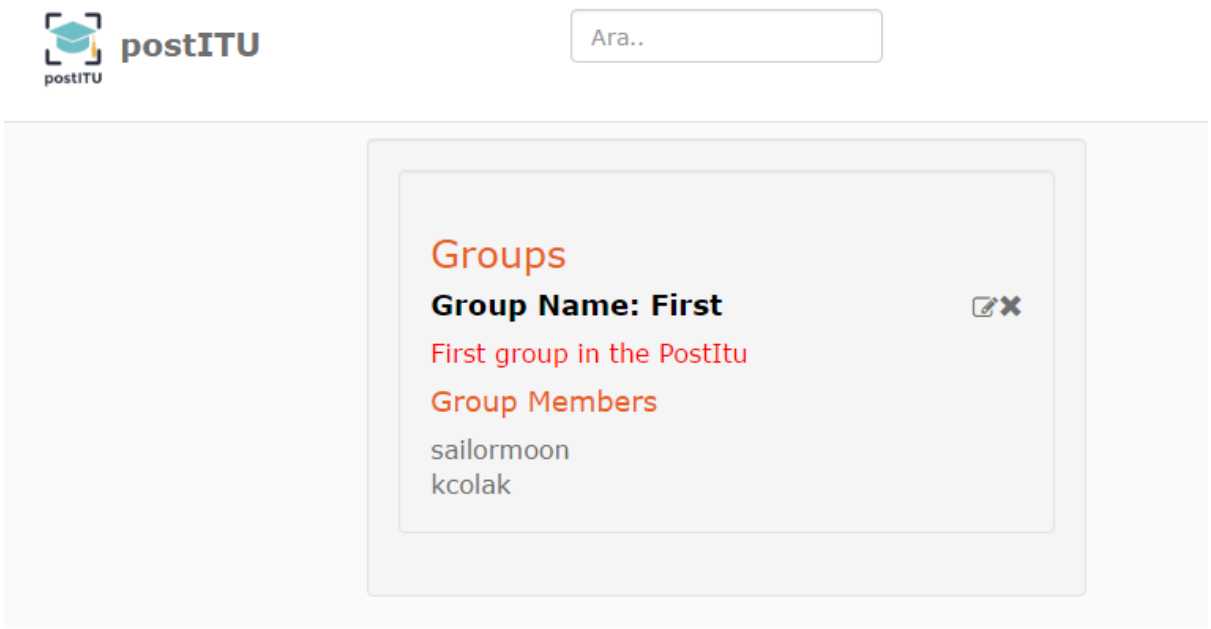
How can I create a group and add participants to it?

Creating group feature is activated after you logged in. The new group link will be appear on the dropdown menu when you logged in. By clicking that button, you will be directed to the create group page. You can create groups with the people you followed. You can give a name and a description to your group. When you press "save" button, your group will be created. Here is a view of creating group page:



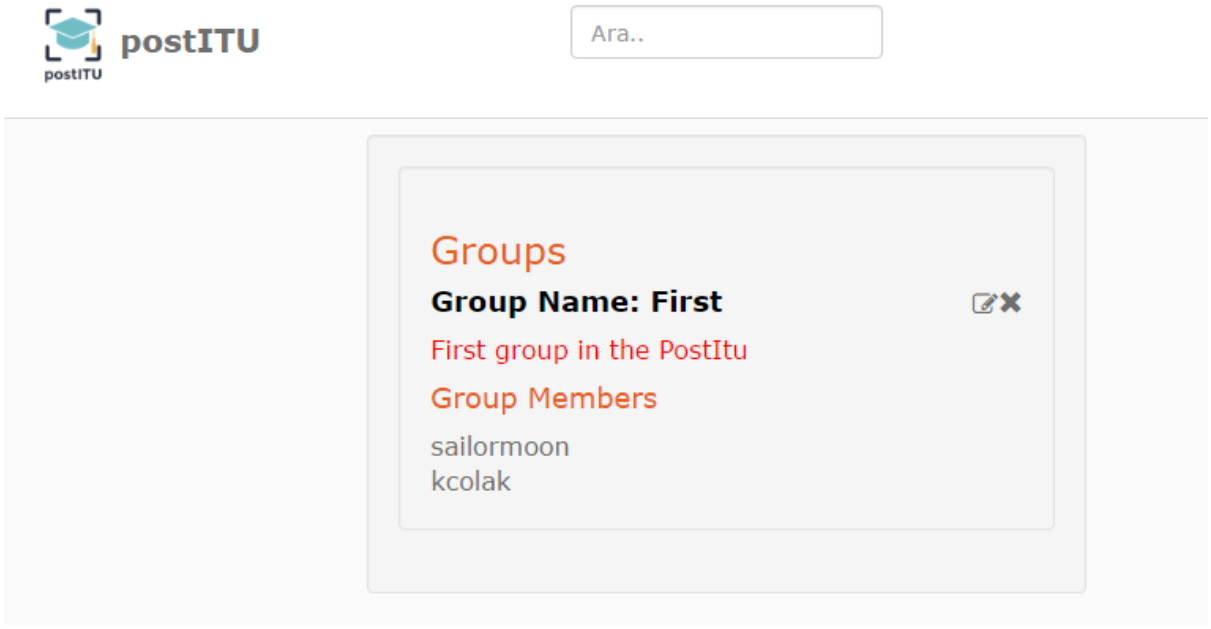
How can I update the group information?

There will be a green group icon on the dropdown menu that is activated after you logged in. When you press it, you can list the all groups that you created. There are two signs near the group name. The pencil icon represents the update function. You can update your group name and description by clicking that icon. Here is a view of the page that lists all groups:



How can I delete a group?

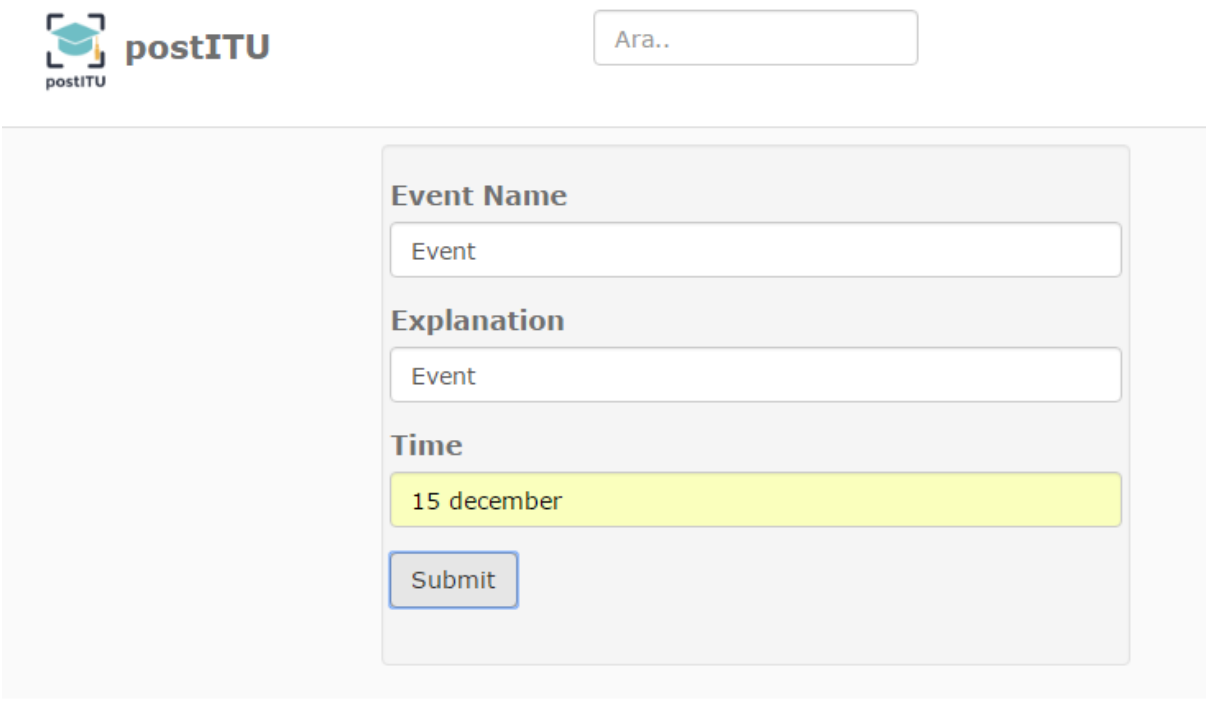
Almost the same process with updating are valid in the removing section. Firstly, you should list all the groups, and click the cross sign near the name. Your group will be deleted. Here is a view to show you to deletion:



2.2.3 Events

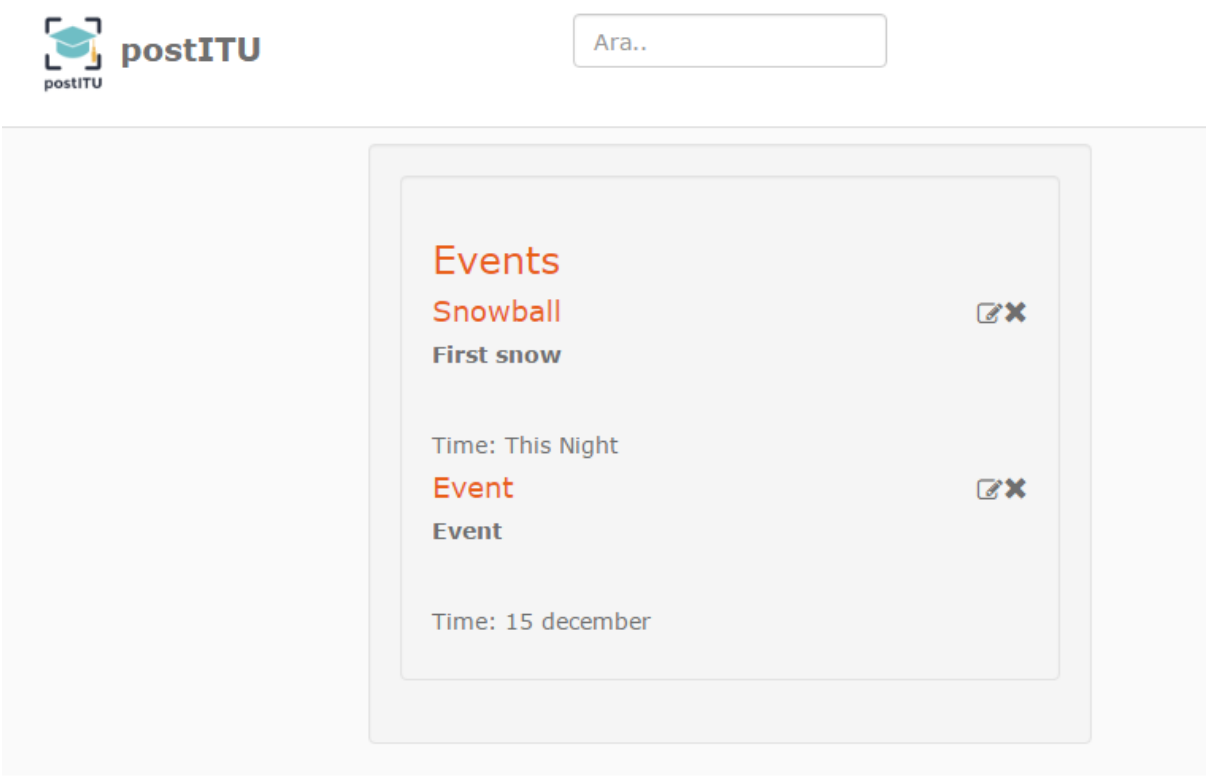
How can I create an event?

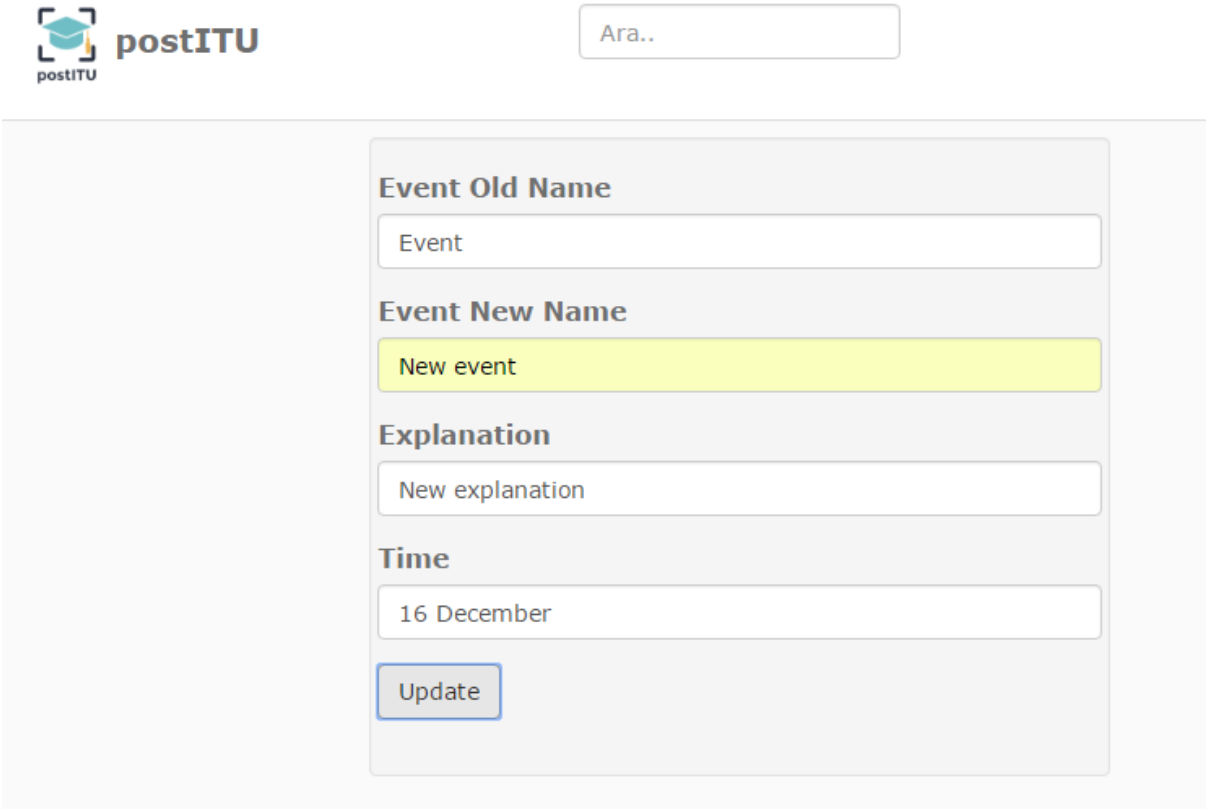
This feature is also activated after logging in. There is an icon for creating events. If you click that icon, you will go to the page where you can create your event easily. Here is a view of the creating event page:



How can I update an event's information?

You should list all events to update them as in the update group section. You can list all events by clicking the green calendar icon on the dropdown menu. Then, you can update the event's information by pressing the pencil icon. Here is some screen pictures of the update event page:

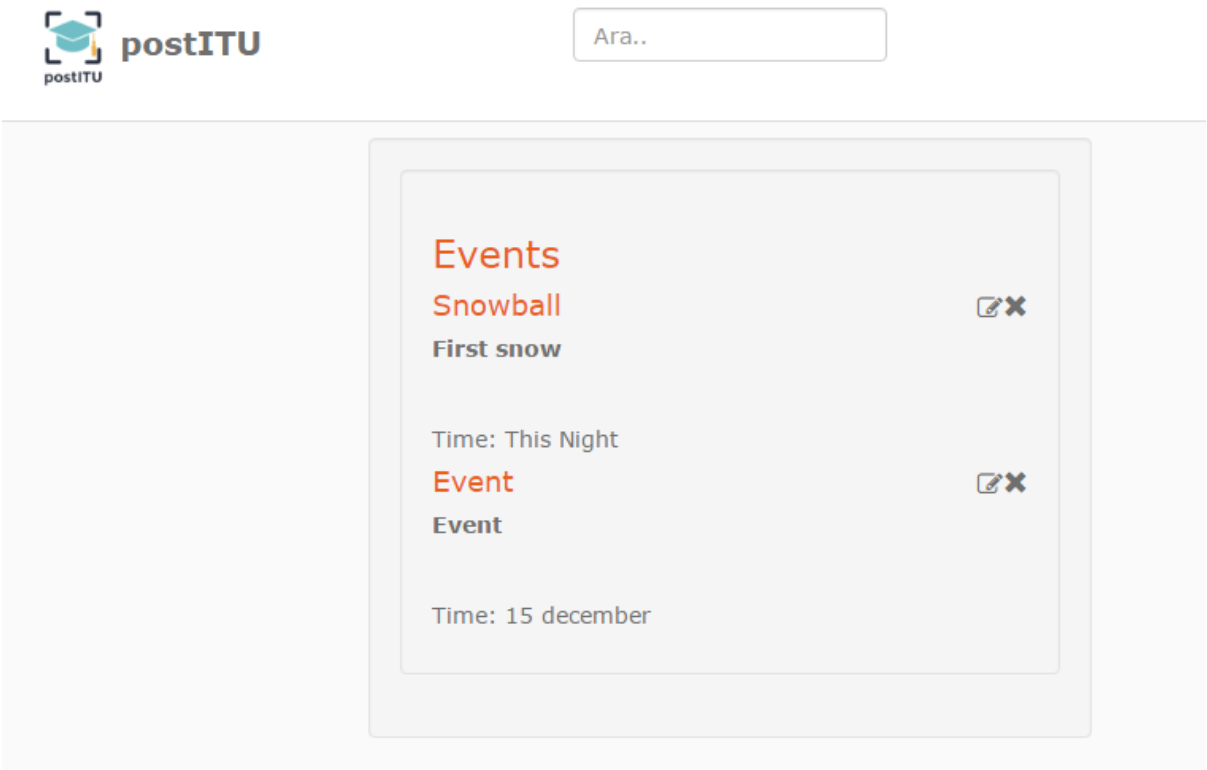




How can I delete an event?

As in the update event section, you should follow almost the same steps. After you listed all events, you can delete them by clicking the cross icon.

Here is the page that lists all groups and shows the cross sign:



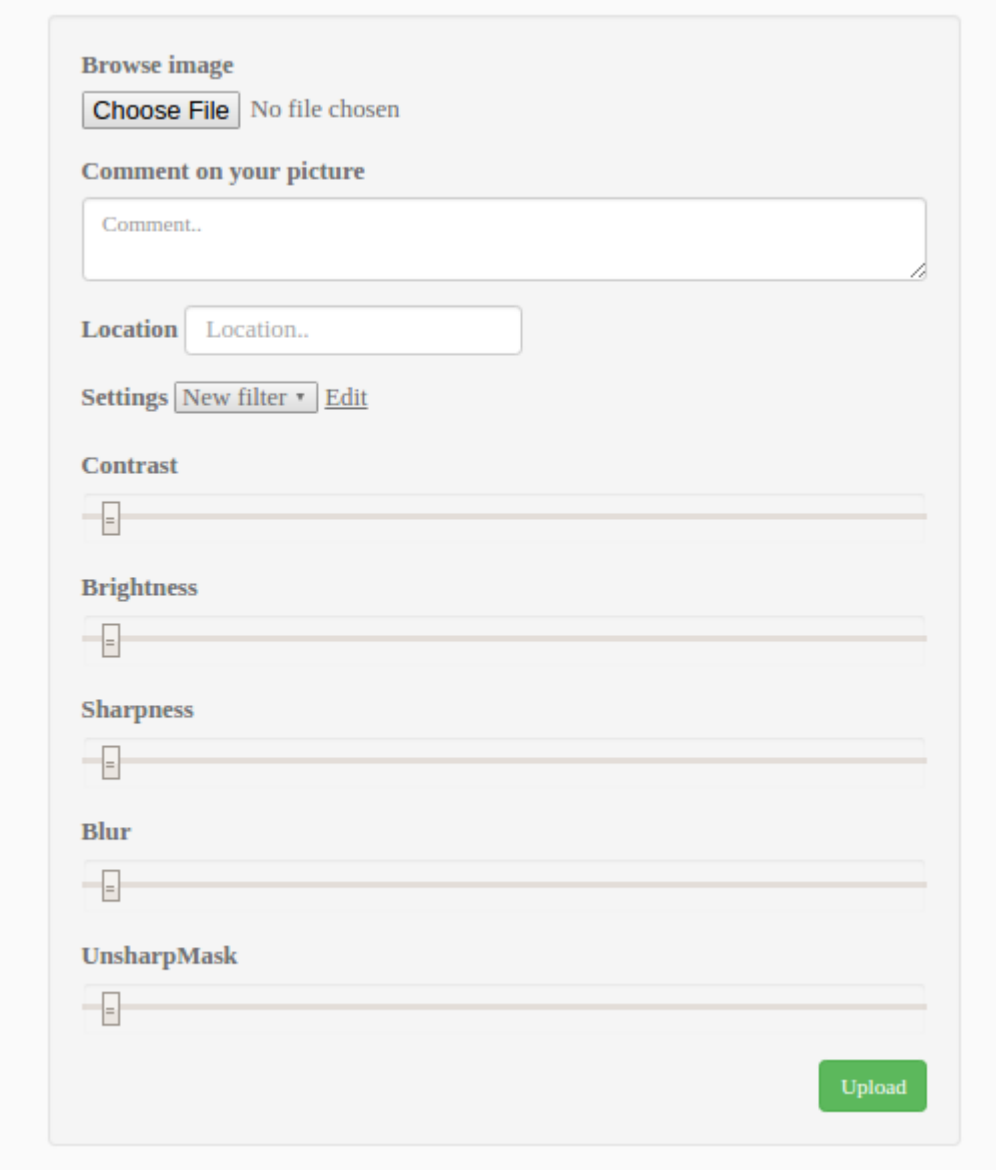
2.3 Parts Implemented by Alim Özdemir

All url relative to bluemix link (http://itucsdb1621.mybluemix.net/)

2.3.1 Image Operations

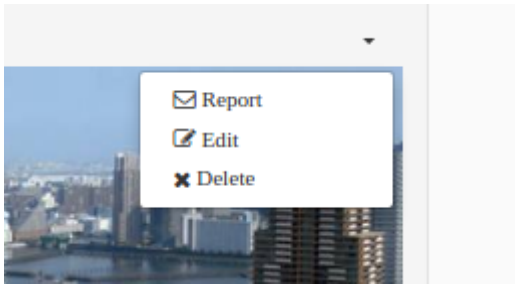
How can I upload ?

Users can upload images via */upload*. This is the main part of the site, almost all thing based on this feature.



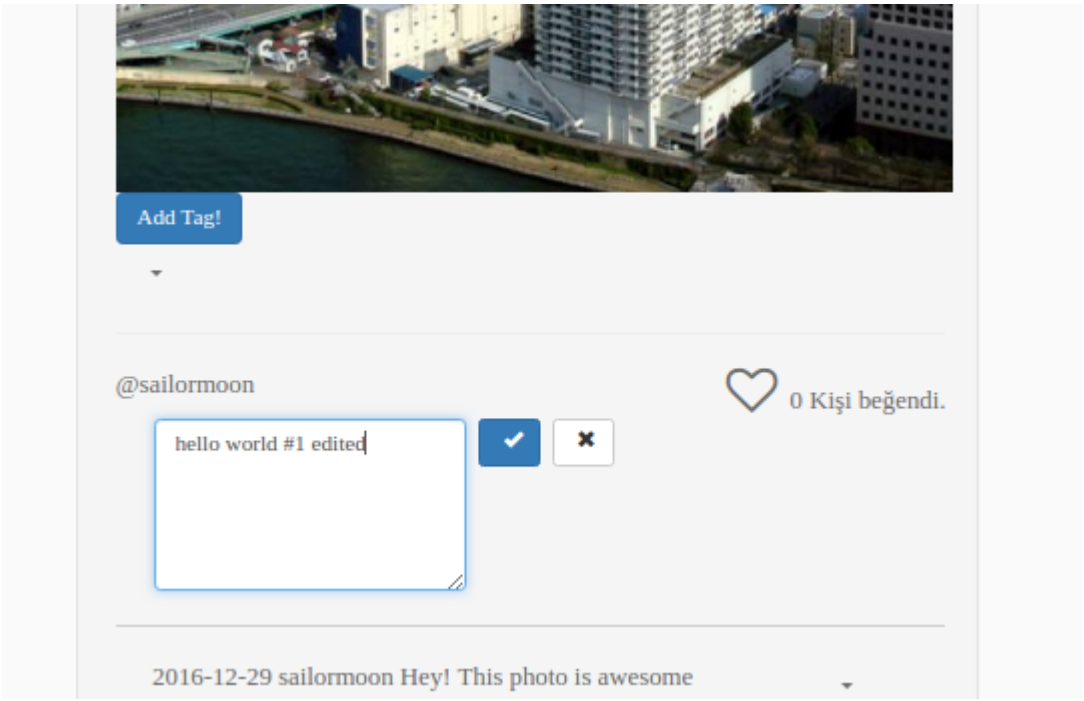
How can I delete ?

The dropdown menu at an image box. Click “x Delete” button.



How can I update ?

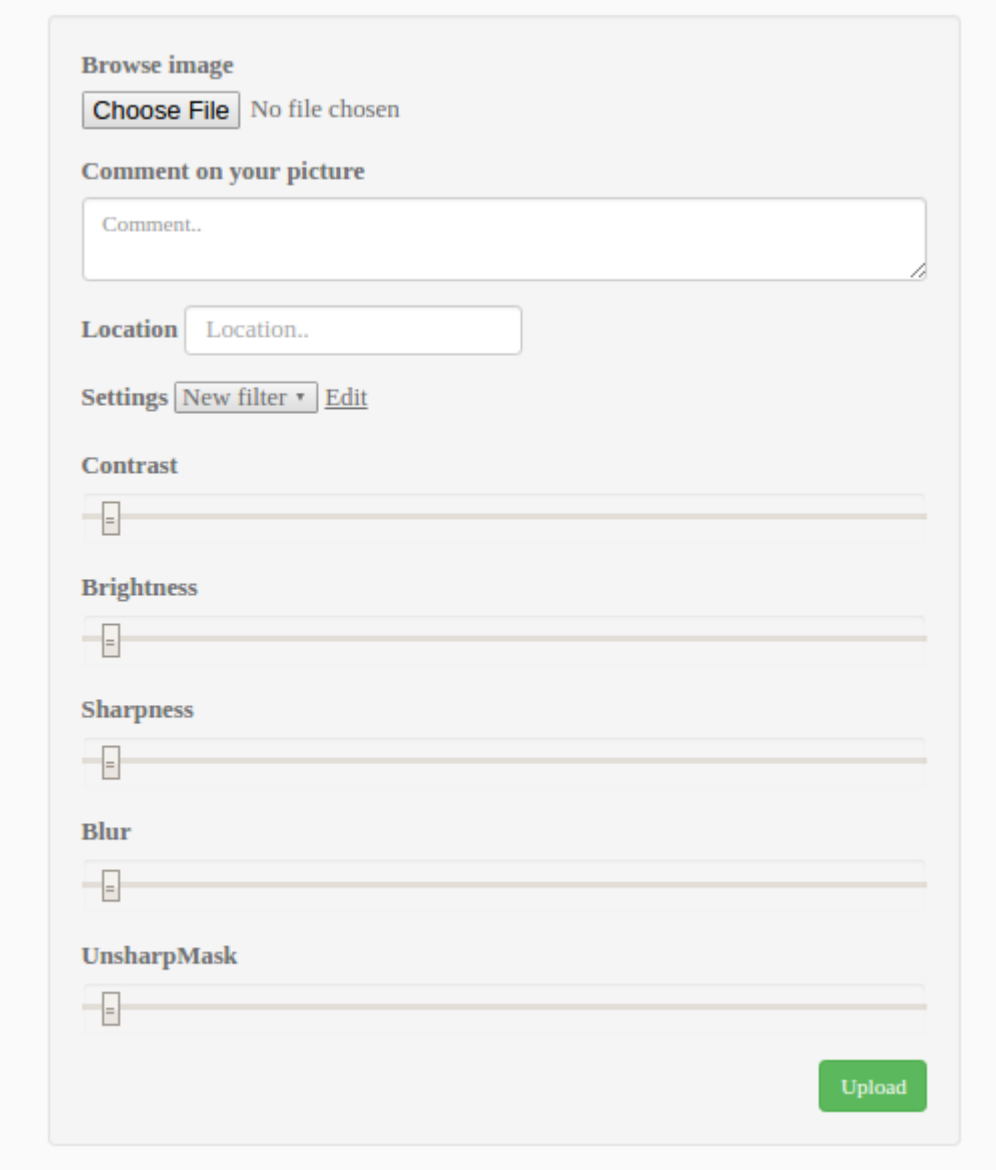
If you are the owner of that image, you can click the description of the image and edit it.



2.3.2 Location Operations

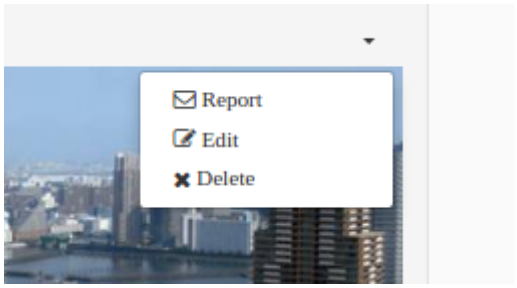
How can I add locations into a image ?

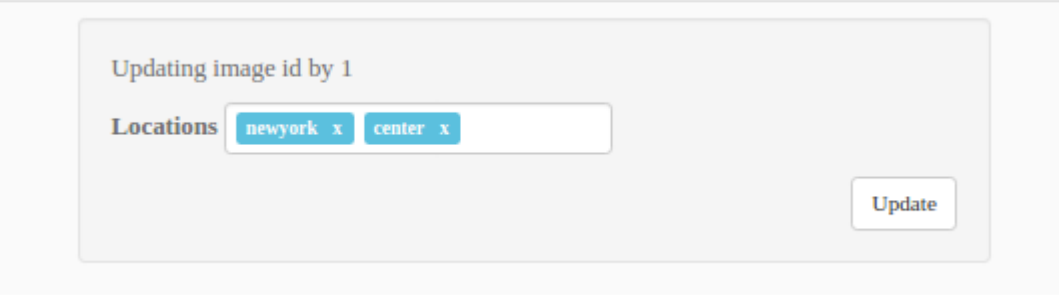
At image uploading screen, there exists location input that you can insert multiple locations.



How can I edit and delete locations ?

The dropdown menu at an image box. Click ” Edit” button. Same functionality as adding locations.





List of All locations

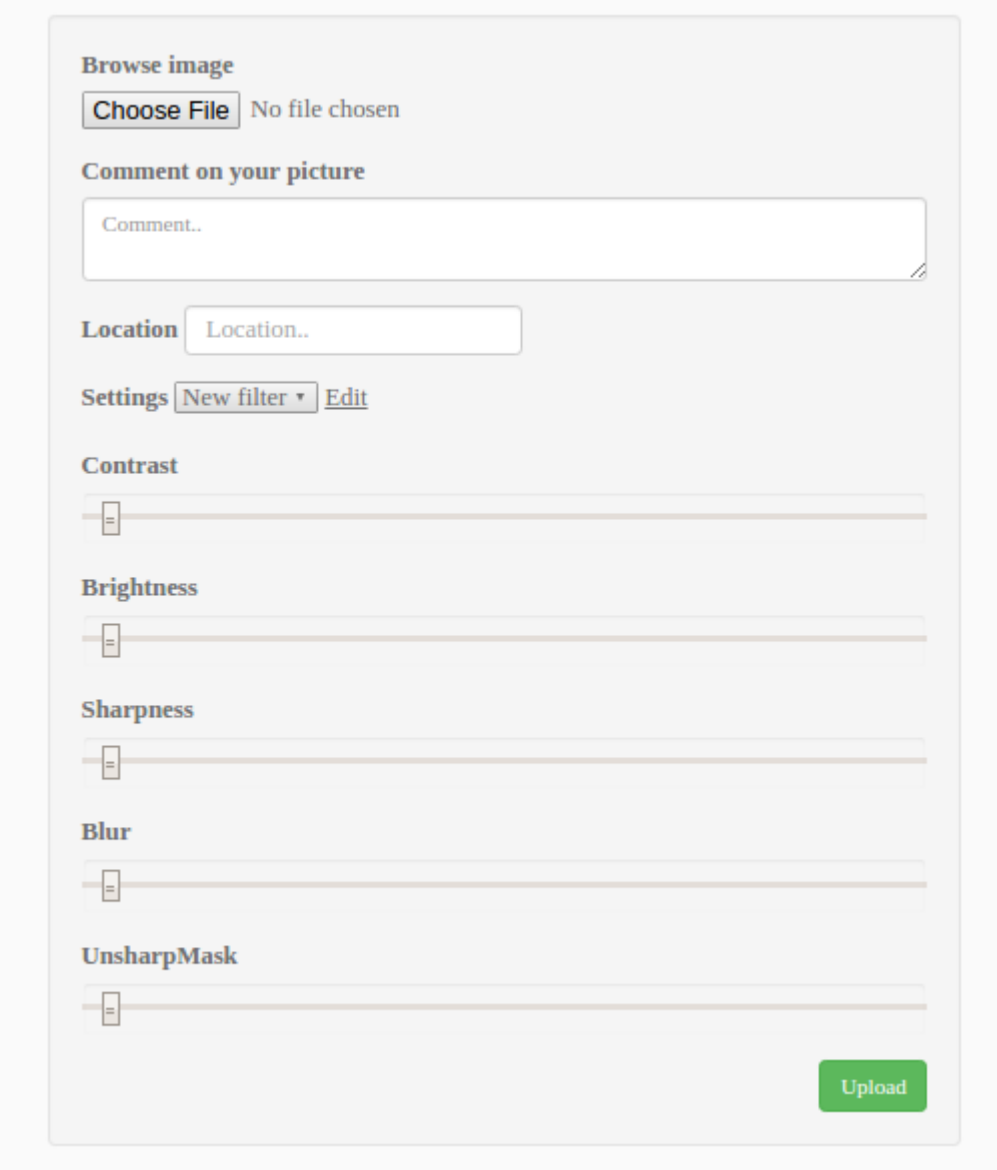
You can list all locations via */locations*.

Location Name	Location Rating	Operation
new york	1.0	✖ Remove it

2.3.3 Filter Operations

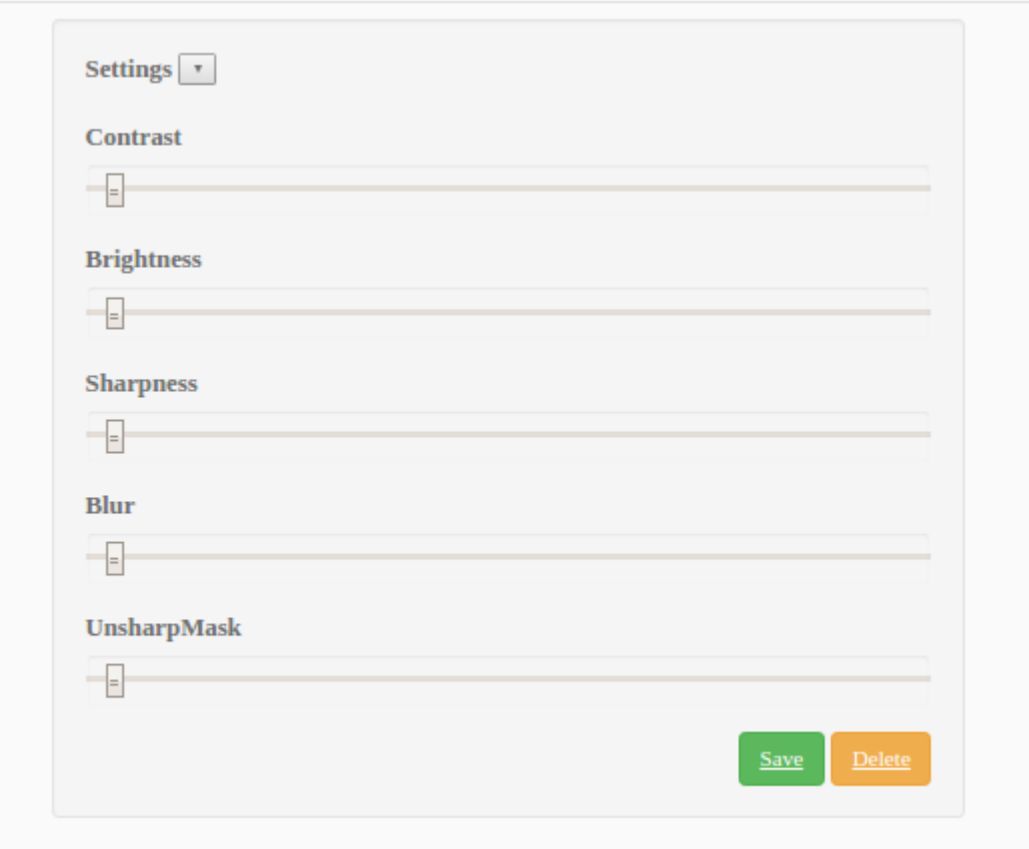
How can I apply a filter to image ?

At image uploading screen, there exists filter settings that you can apply a filter to image.



How can I manage my filter settings ?

You can manage your filters via */filter/index*



2.4 Parts Implemented by Halit Uyanık

2.4.1 Bidding

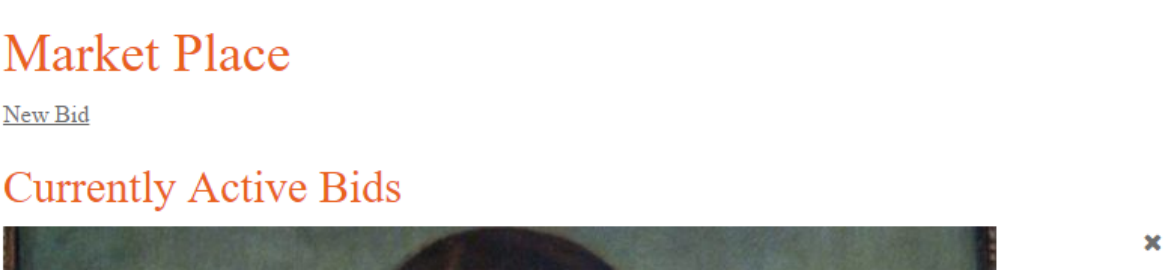
Bidding page provides users to show their own works to gather interest of others and earn some money from it. This is a nice place for talented people to show their powerful works.

Bid page can be accessed via:

http://itucsdb1621.mybluemix.net/bidPage

How to Create a New Bidding?

In the bid page you will see a link on top of the page called “New Bid” just under the “Market Place” header.



After clicking the link you will be directed to bid form page, with header “Place a new item on Market”

After filling the Item Name, Description, choosing an image, and setting a new price with a limitation that you cant go beyond 2 zeros after .0 (0.00), click the submit button.

You successfully added your new bid to market!

Resim Adı	Resort
Fiyat	1123.32 $
Detay	
A beautiful resort!	

Bid price Submit

How to Bid to an existing bid?

At the bid page you may see one or more bids which you can bid more money from the current price.

You will be able to see an area under the image information, where there is one input area, and a button.

Detay
A beautiful resort!

Bid price Submit

There you need to enter a number higher then the current price (or you will recieve an error message!) and click submit button.

After successfully submitting you will see that you new price is active on the image.

Before change:

Fiyat	1123.32 $

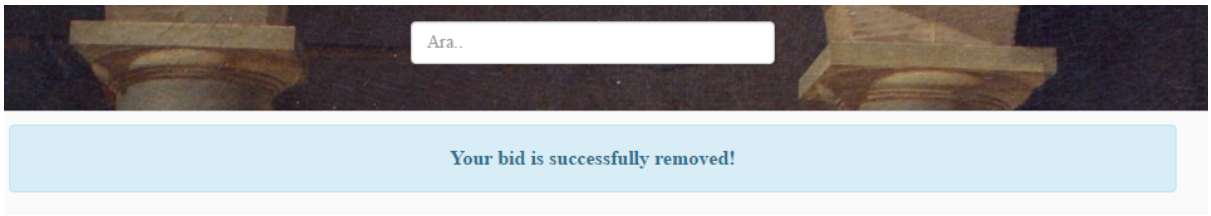
After change:

Fiyat	1499 $

How to remove a bid from market?

At the bid page near an existing bid you will be able to see a 'x' button as below.

If you click the button the bid will be removed from the market.



Warning: Since it is logical to sell an item which exists on the website. If you delete an image from your main page, the associated bid for that image will also be removed!!

2.4.2 Layout Change

Layout changing is focused on users session, and it is an open system for all users to share their created layouts directly.

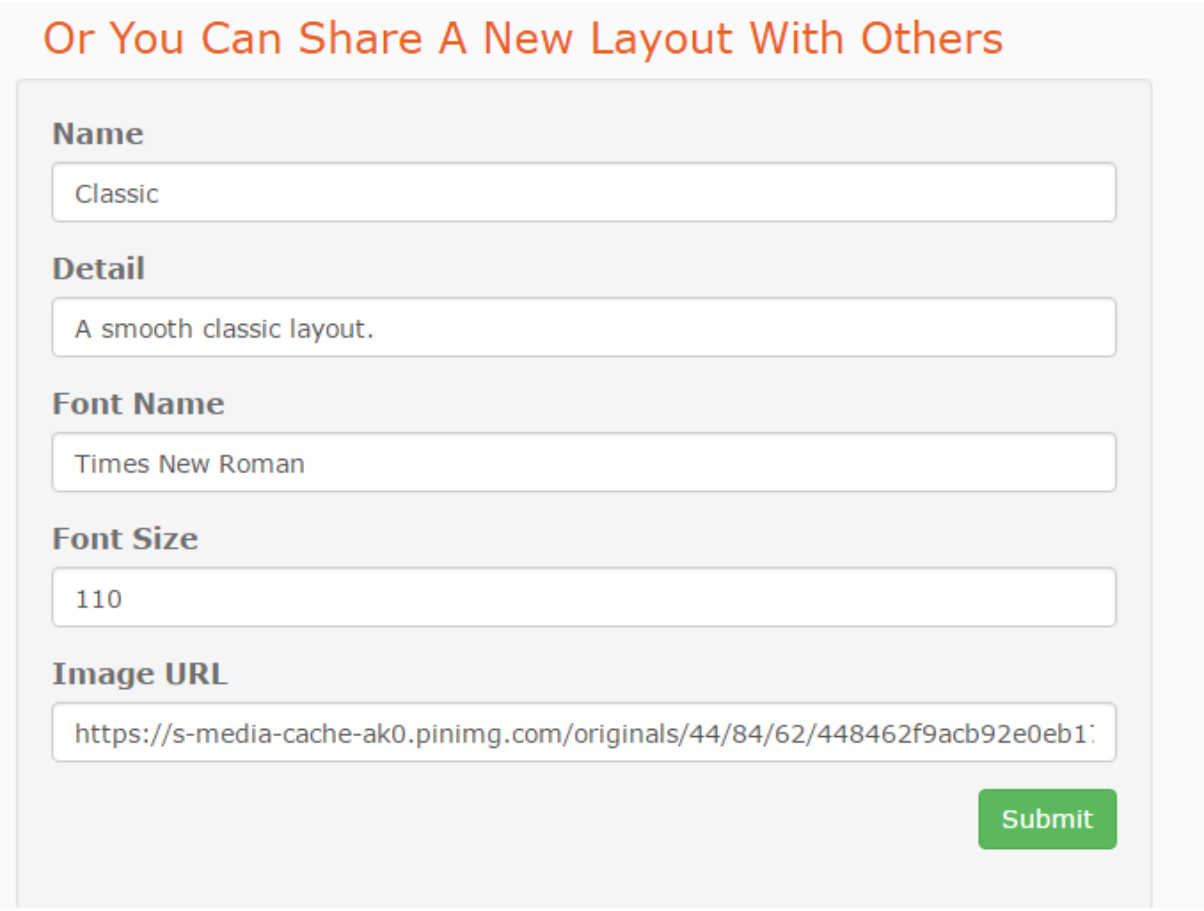
Layout changing is focused on header background image change, font of the website (as defined in layout.html), and the font-size of the website in percentage unit.

The link for the layout operations is:

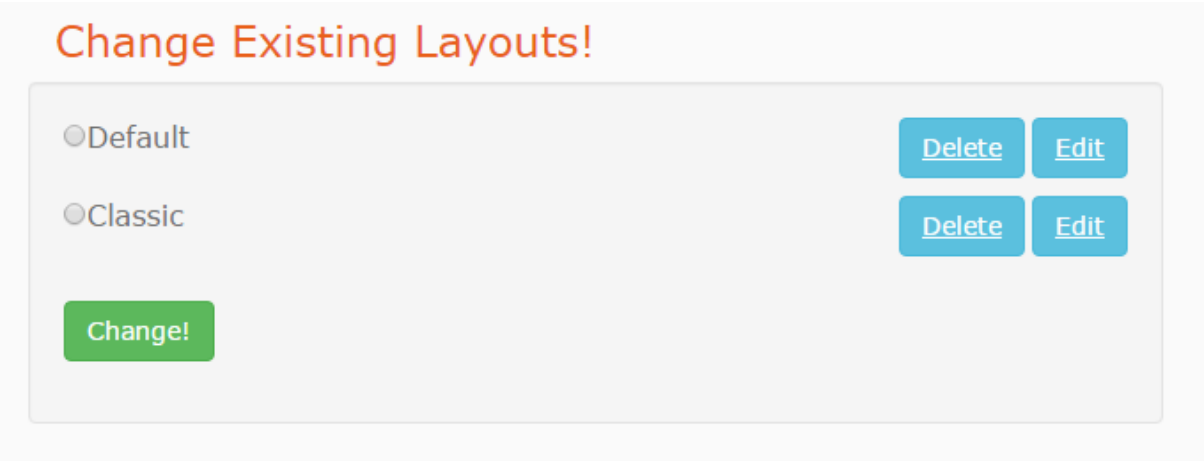
http://itucsdb1621.mybluemix.net/changeLayout

How to Create a New Layout?

At the layout page. Below the existing layouts section there exists a second part called "Or You Can Share A New Layout With Others". Here you can enter the information about your new layout, all fields are required, then submit it!

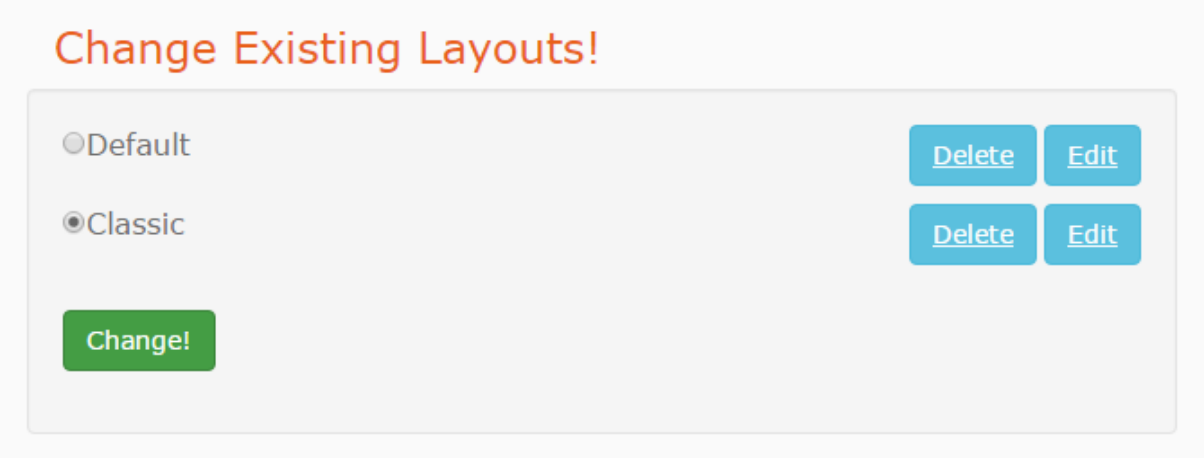


After that you will see that your new layout is added to list.

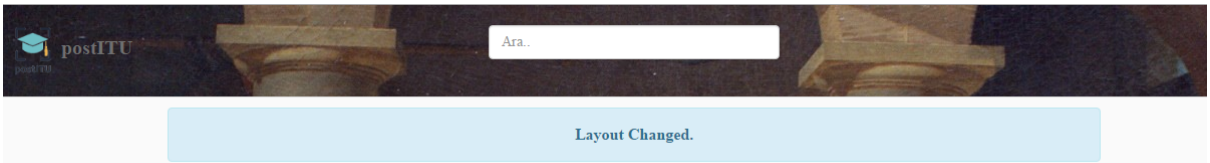


How to Change The Default Layout?

After going to the layout change page, at top you will see a header “Change Existing Layouts!”. There you can choose one of the layouts with radio buttons then click the green ‘Change!’ button at bottom left.

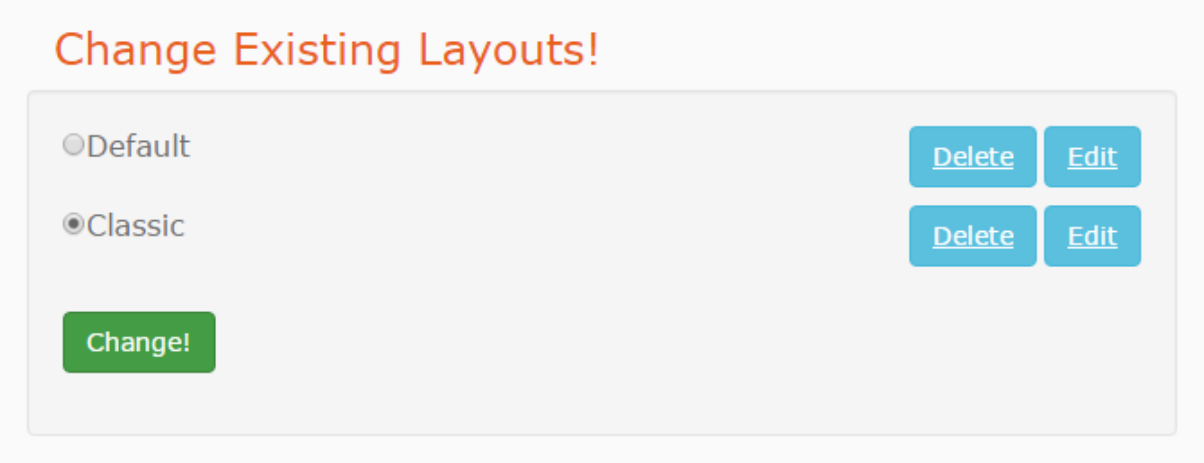


You will notice that you layout is changed at the success message page.

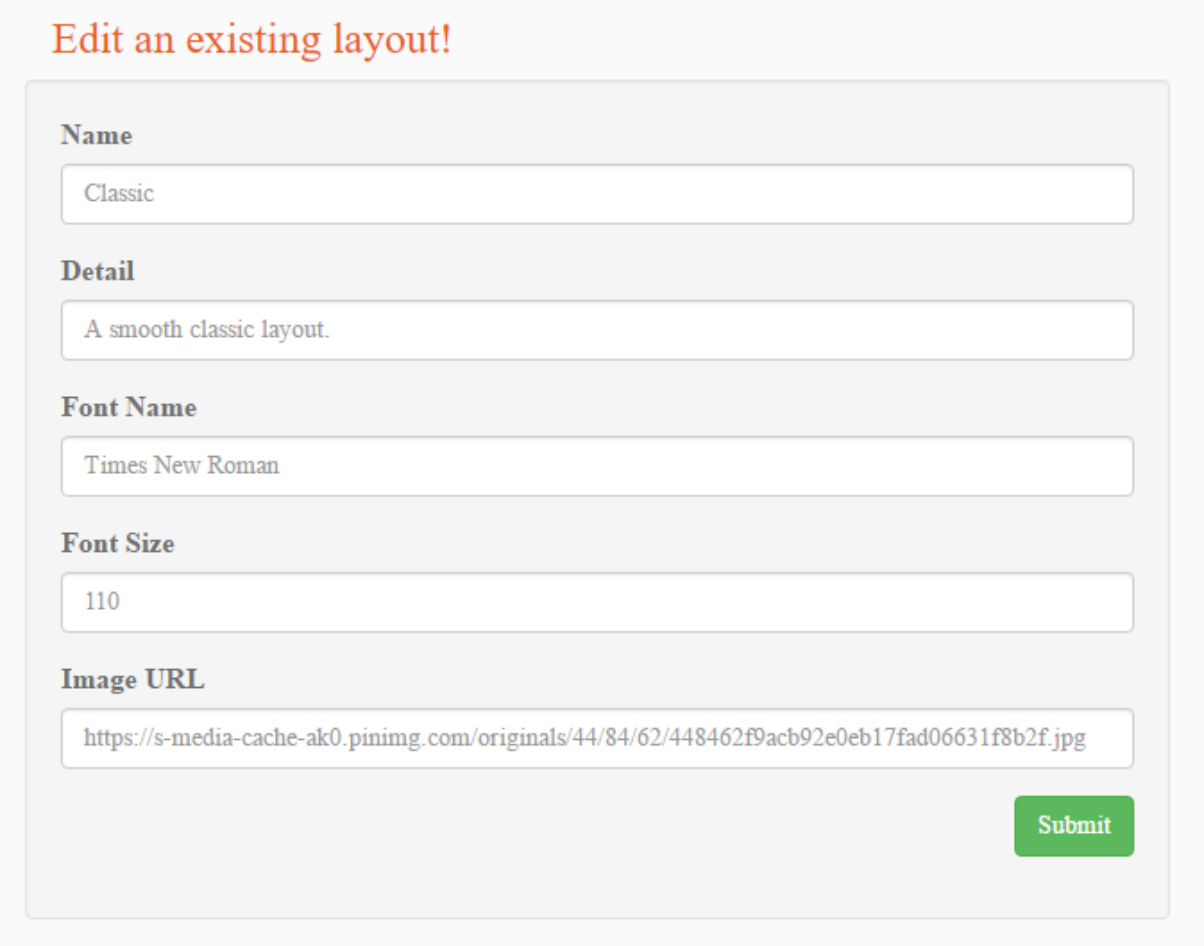


How to Edit Existing Layouts?

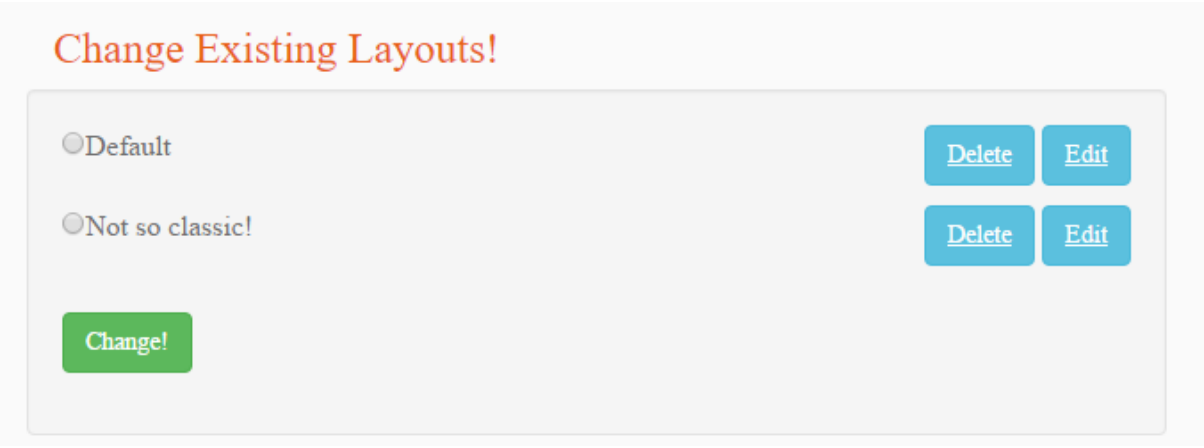
At the layout page. In "Change Existing Layouts!" section, there are two buttons near a layout option. One of them is 'Edit'. Click the button.



After clicking you will be directed to the layout editing page. The default values are the ones for that layout, and you are "*required*" to fill all the areas.

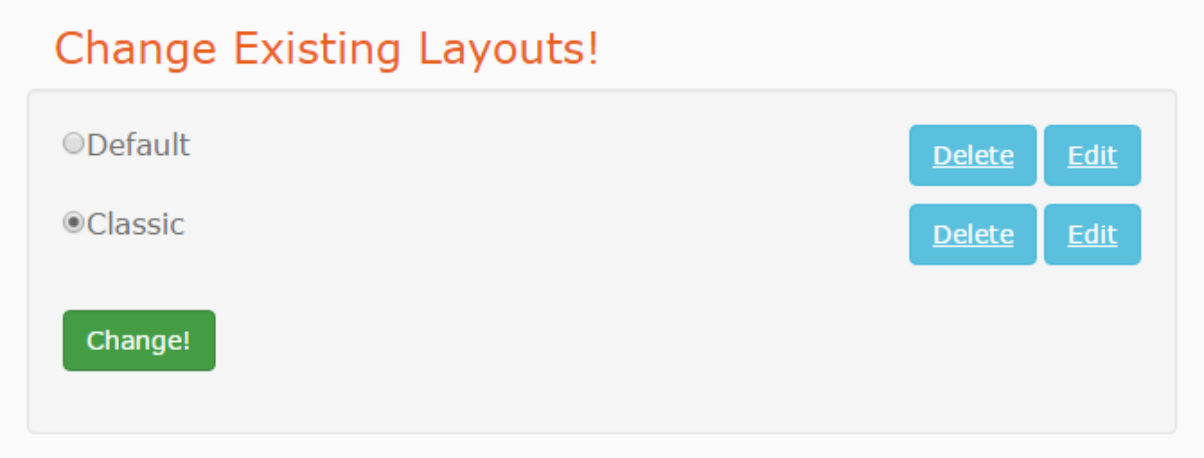


After filling all the information and sending the form info, you have successfully changed the layout!



How to Delete Existing Layouts?

At the layout page in “Change Existing Layouts!” section, you can see the “Delete” button near an option.



After clicking delete the layout will be deleted from the page.

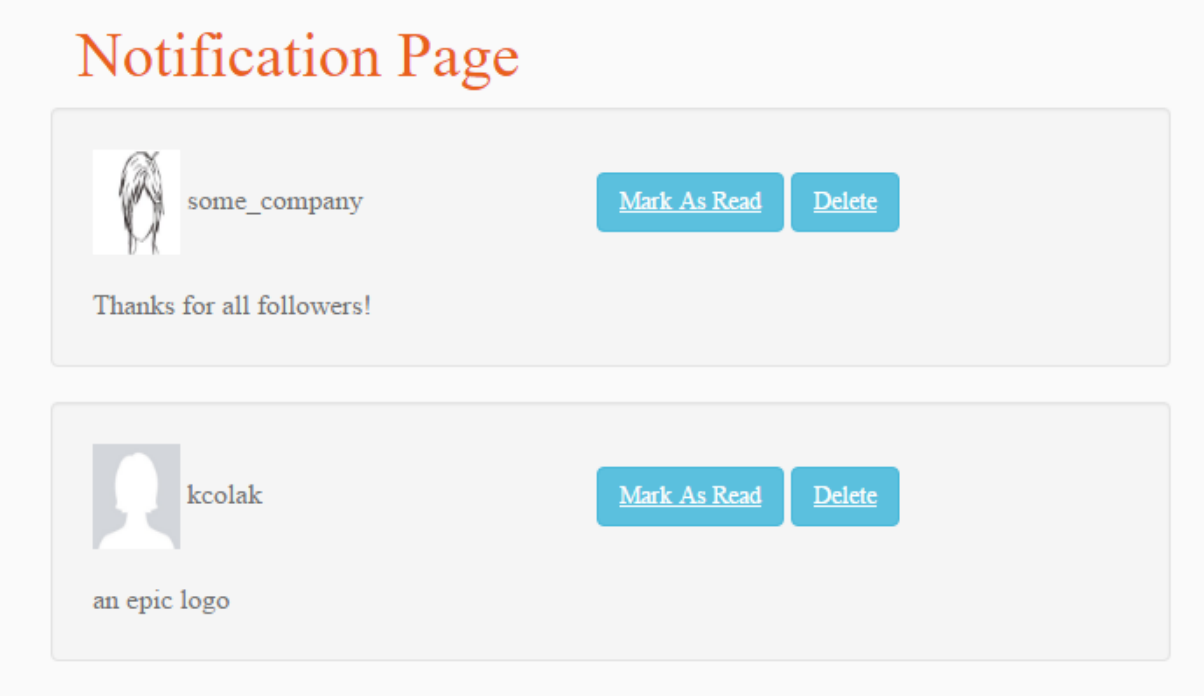
2.4.3 Notifications

Notifications are implemented as a way for user to see new notifications when a user uploads an image to the server.

To see how an image is uploaded see the guide for image upload by team member Alim.

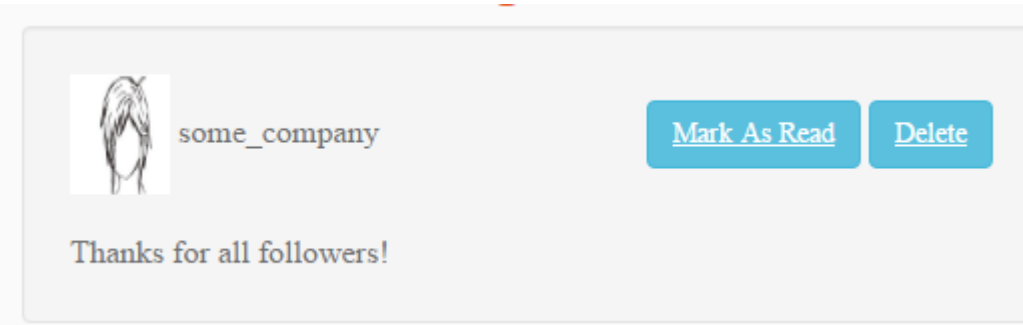
This is the main page for notifications:

http://itucsdb1621.mybluemix.net/notification

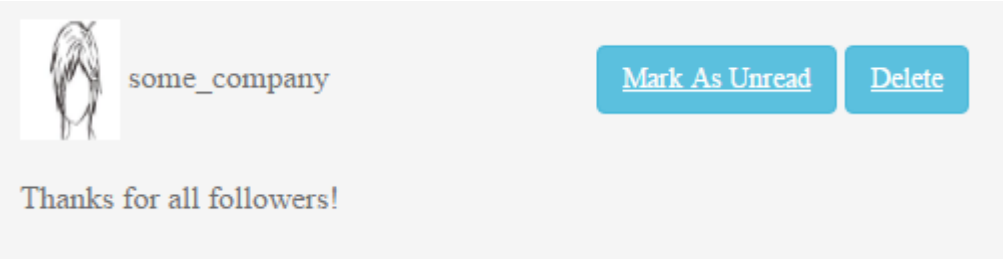


How to update a Notification Status?

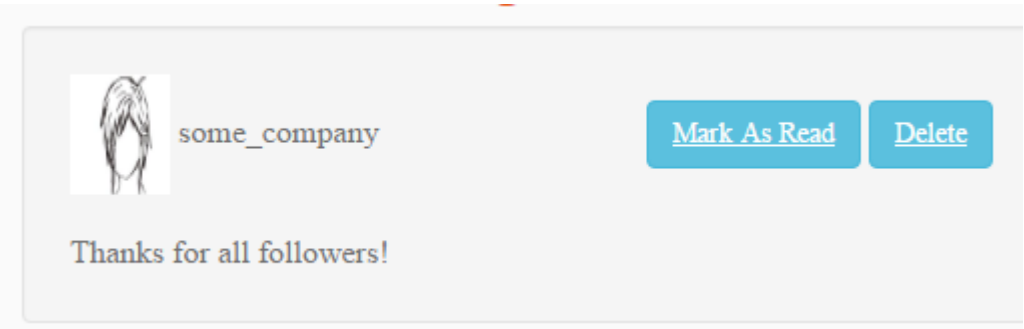
It is possible to update a notification status by marking it as “Mark as Read” and “Mark as Unread”. The default value is of course “Mark as Read” as all newly notifications should be unread.



If you click the button “Mark as Read” the status will be updated.



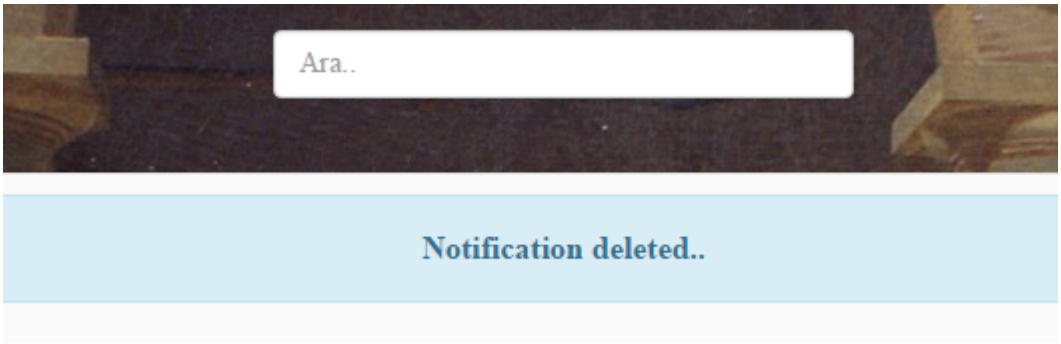
Likewise, there may come a time when a user wishes to mark the notification as unread, as to not forgot it or for another reason. If you click the button “Mark as Unread” then the status will return to its default value.



How to remove a Notification?

If a user wishes to remove a notification from the notification page since after some time the amount may distrub the user, it is possible to remove a notification from the button right to "Mark as —".

After clicking it, the notification will be removed.



2.5 Parts Implemented by Ömer Faruk İNCİ

2.5.1 Sending Message To a Specific User

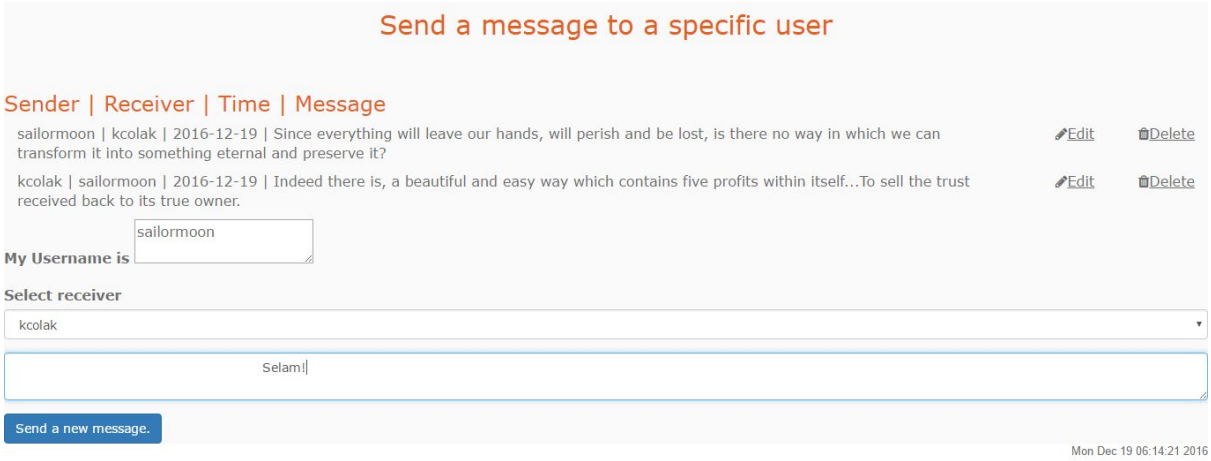
Send Message page enables users to send a message to any of the registered users to the website. All the usernames are shown in a clickable list.

Send Message page can be accessed via:

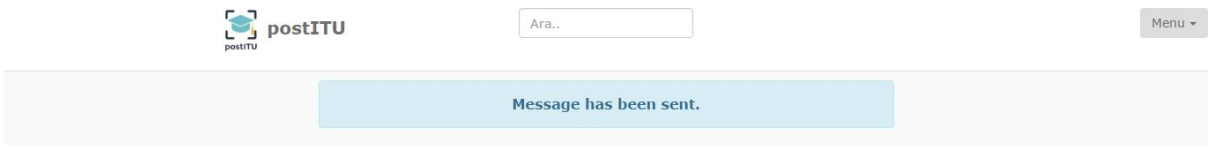
http://itucsdb1621.mybluemix.net/gmessage

How to Send a New Message?

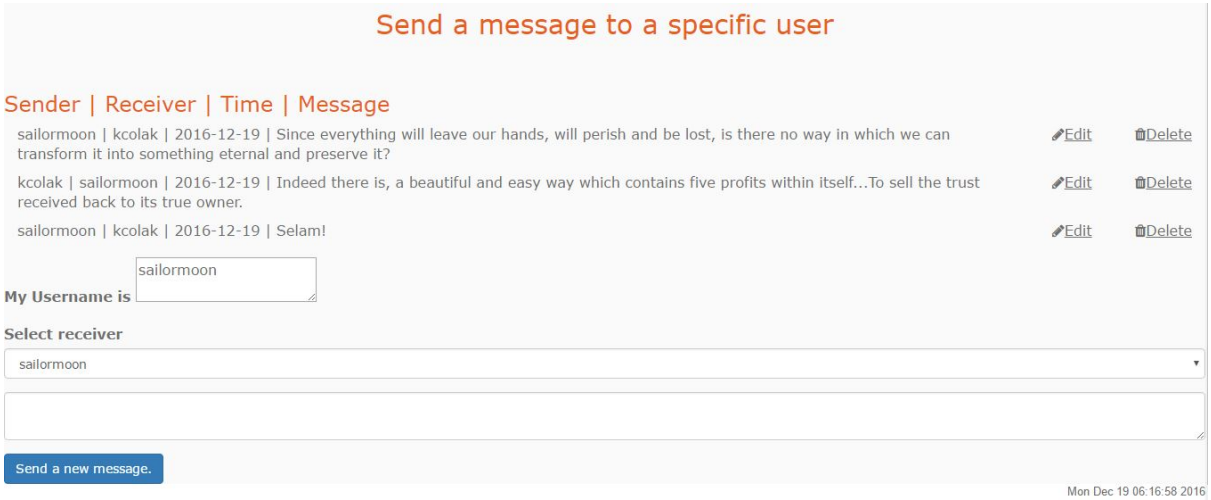
In the Send Message page you can see the text box on the top of the 'Send a new message' button. And also, there is a user list on the top of the text box. After selecting the user from the user list as a reciever and typing your message in the textbox, you can send your message by clicking on the 'Send a new message.' button.



After clicking the 'Send a new message' button. If the sending message operation is succeeded, a page will be appear shows that sending message completed successfully.

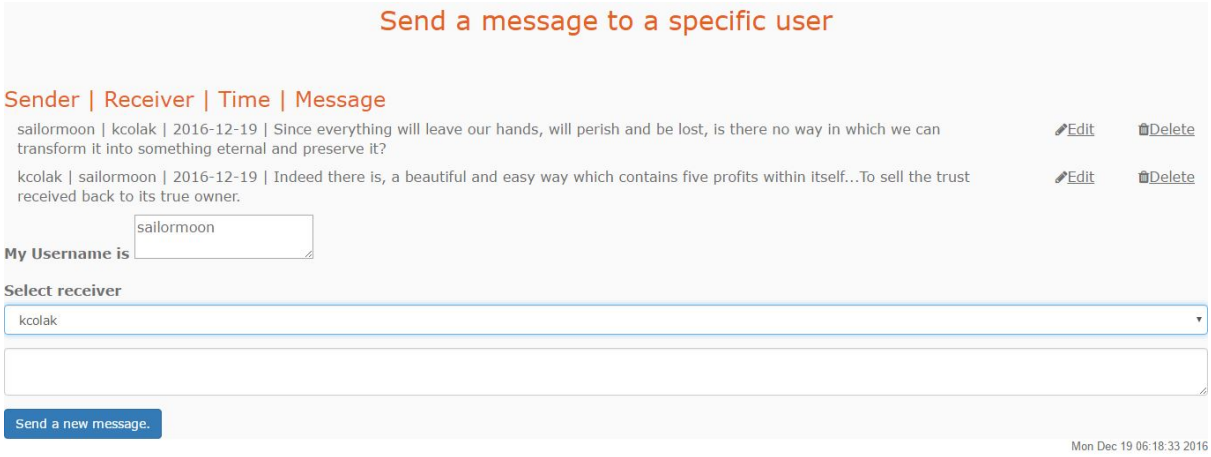


You can see that your message is added successfully by accesing the Send Message page again.

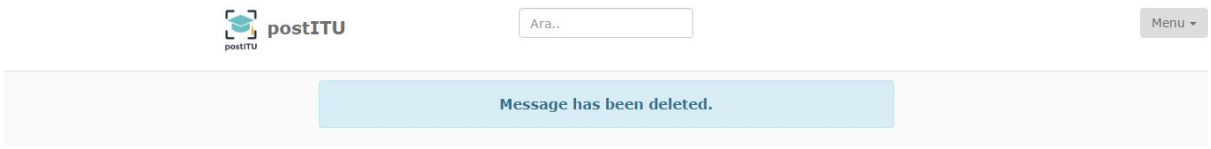


How to Delete an Existing Message?

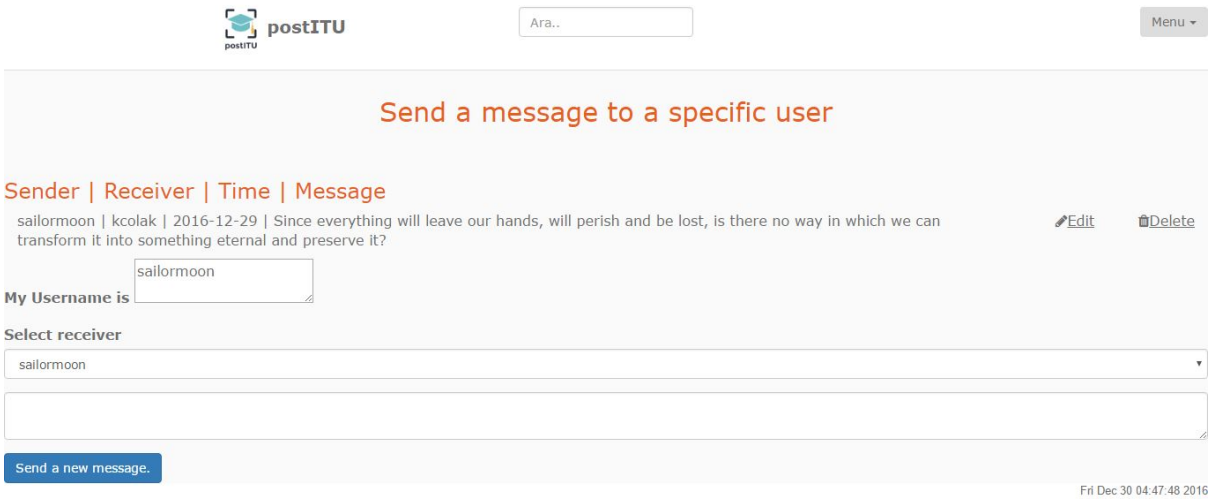
In the Send Message page, you can see that there is a cross icon at the right side of the each message.



If you click to the cross button, the corresponding message will be deleted and a page will be appear shows that deleting message completed successfully.

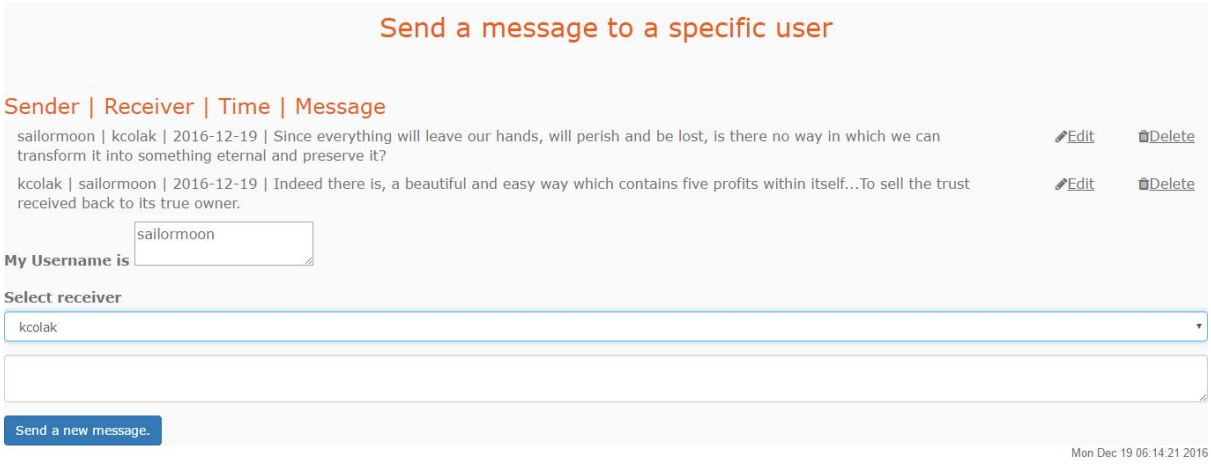


You can see that your message is deleted successfully by accesing the Send Message page again.

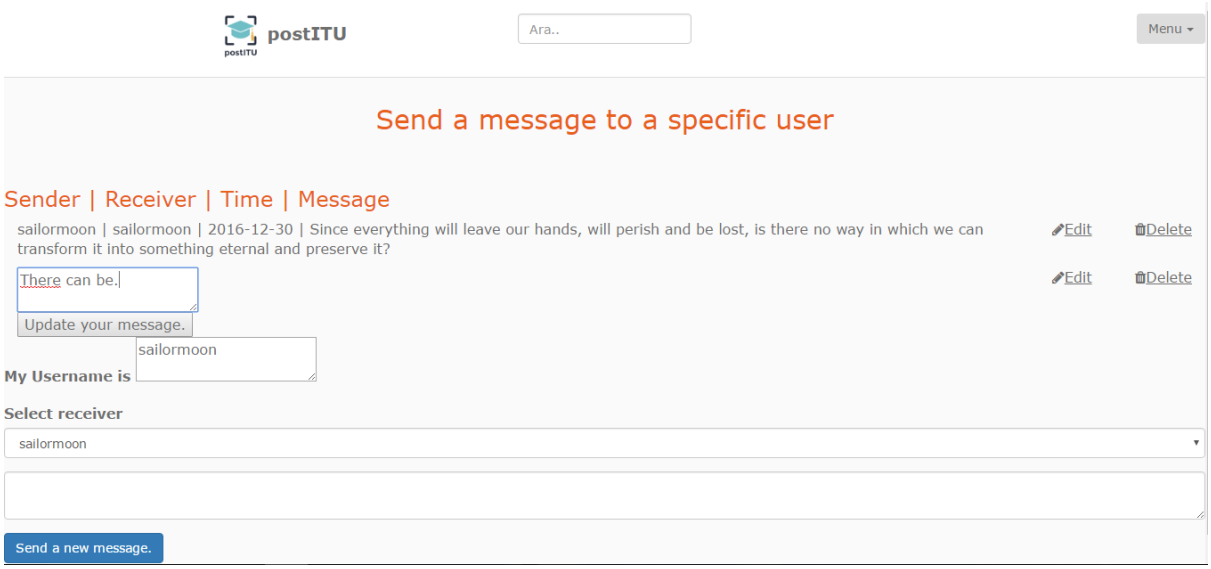


How to Update an Existing Message?

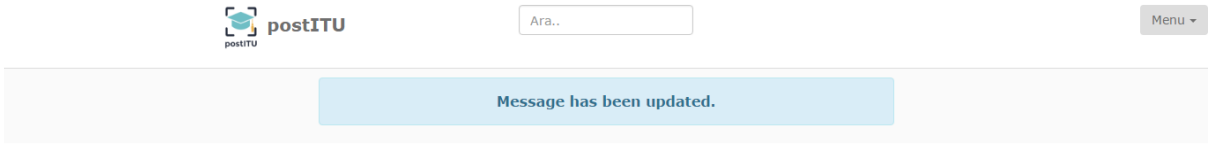
In the Send Message page, you can see that there is a pencil icon at the right side of the each message.



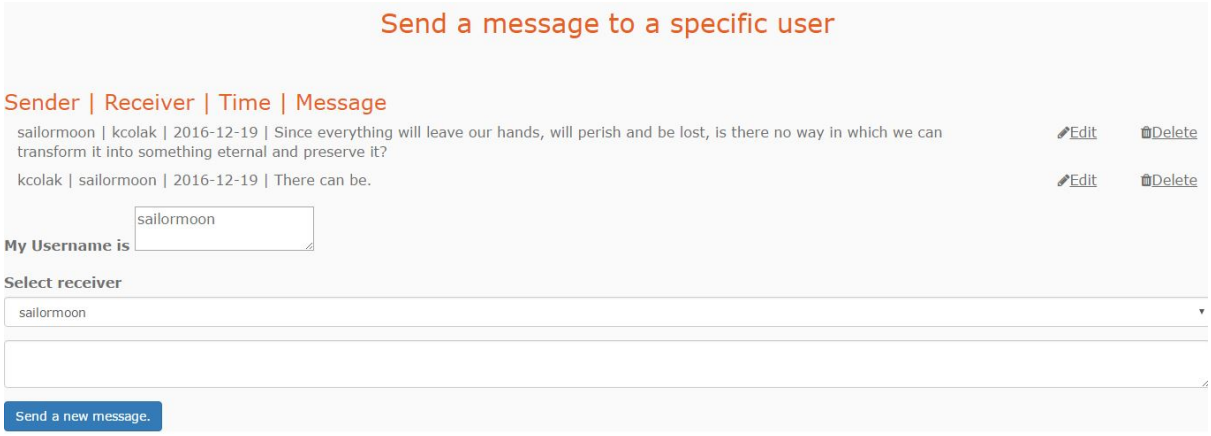
If you click to the pencil button, a textarea appears for entering the new message. And there is a button named 'Update your message.' for updating the old message with the new one.



If you click to the 'Update your message.' button, the corresponding message will be updated and a page will be appear shows that updating message completed successfully.



You can see that your message is updated successfully by accesing the Send Message page again.



2.5.2 Sending Direct Message

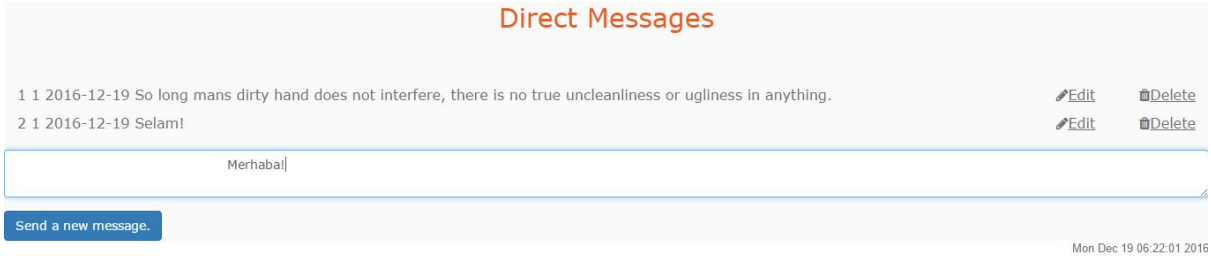
Direct Message page enables users to send a message, delete a message or update an existing message.

Direct Message page can be accessed via:

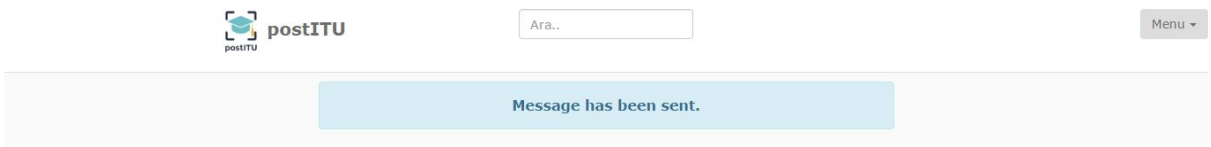
http://itucsdb1621.mybluemix.net/dmessage

How to Send a New Message?

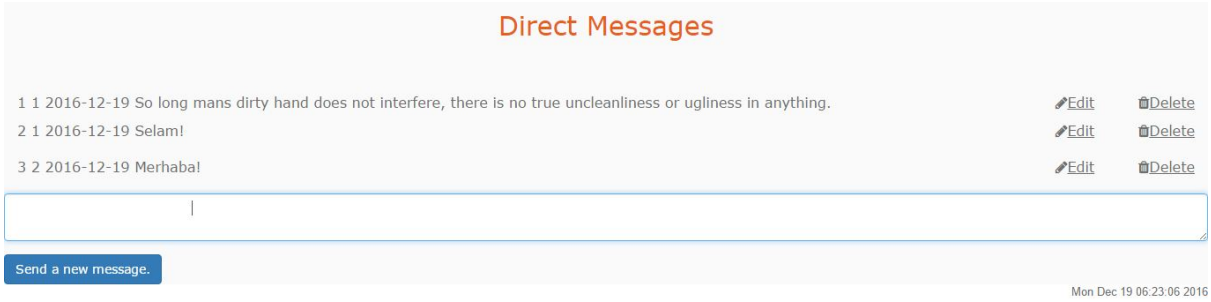
In the Direct Message page you can see the text box on the top of the 'Send a new message' button. After typing your message in the textbox and clicking on the 'Send a new message.' button, you can send your message.



After clicking the 'Send a new message' button. If the sending message operation is succeeded, a page will be appear shows that sending message completed successfully.



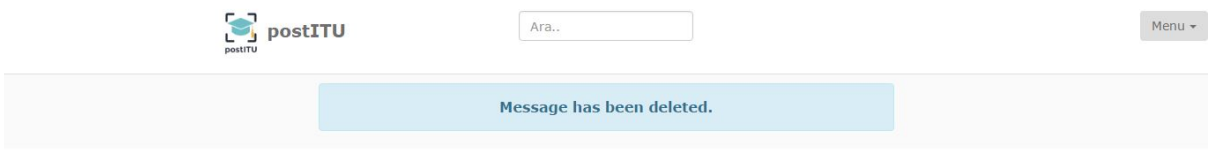
You can see that your message is added successfully by accesing the Direct Message page again.



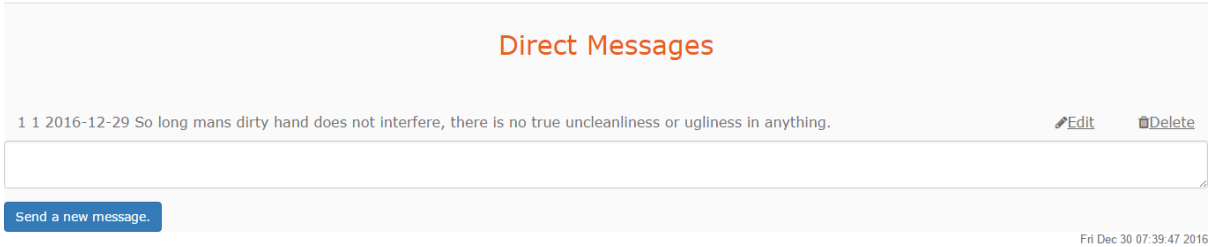
How to Delete an Existing Message?

In the Direct Message page, you can see that there is a cross icon at the right side of the each message.

If you click to the cross button, the corresponding message will be deleted and a page will be appear shows that deleting message completed successfully.

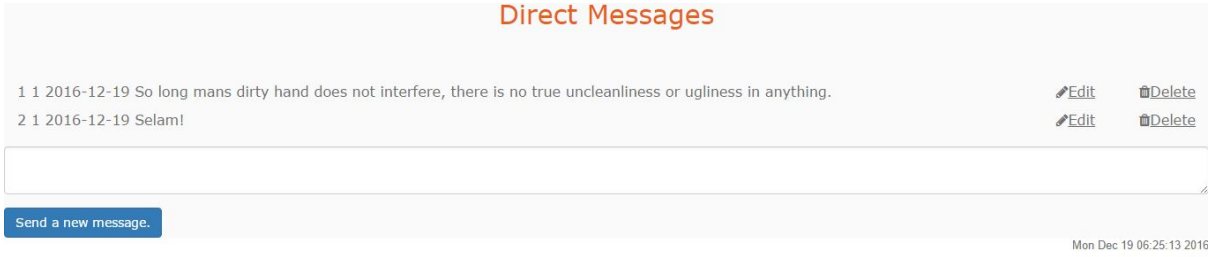


You can see that your message is deleted successfully by accesing the Send Message page again.

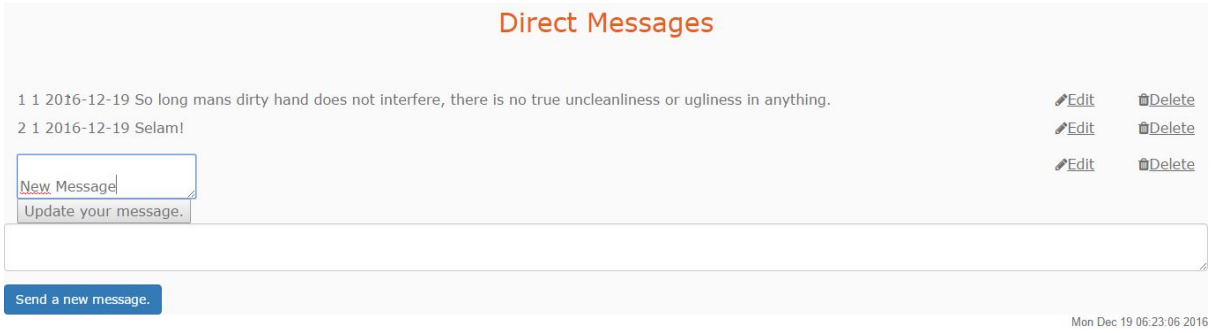


How to Update an Existing Message?

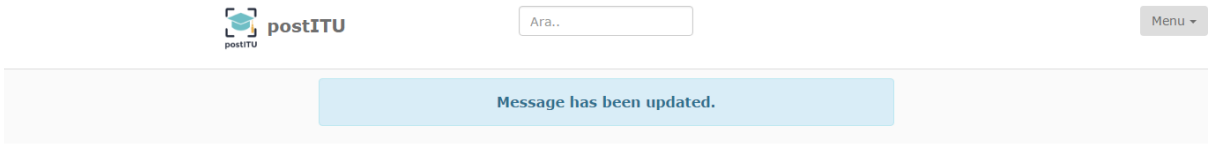
In the DİRect Message page, you can see that there is a pencil icon at the right side of the each message.



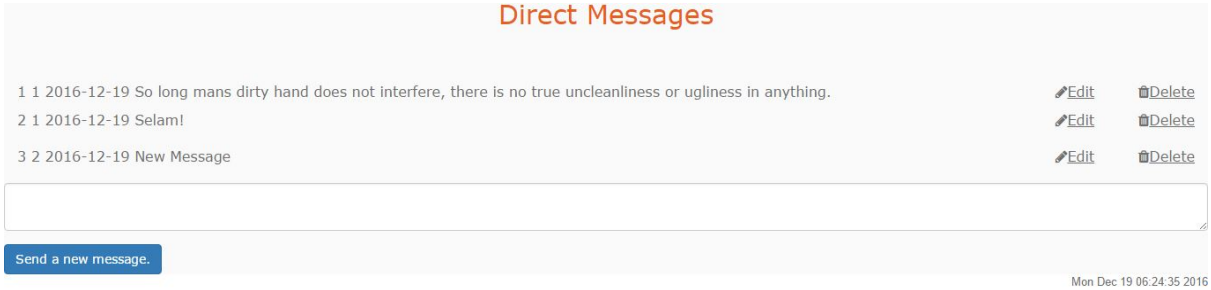
If you click to the pencil button, a textarea appears for entering the new message. And there is a button named 'Update your message.' for updating the old message with the new one.



If you click to the 'Update your message.' button, the corresponding message will be updated and a page will be appear shows that updating message completed successfully.



You can see that your message is updated successfully by accesing the Direct Message page again.



CHAPTER

THREE

DEVELOPER GUIDE

3.1 Parts Implemented by Sıddık Açıl

Comments, content reporting, and image tagging is implemented by me.

3.1.1 General Database Design

3.1.2 Comments

Database Design

```
CREATE TABLE IF NOT EXISTS comments (
        comment_id serial primary key,
        user_id int REFERENCES users(ID) ON DELETE CASCADE,
        image_id int REFERENCES images(image_id) ON DELETE CASCADE,
        time date,
        comment text
);
```

Comments table has a serial primary key and two foreign keys, one to users table, other to images table. Users and images both has 1-n relation with comments, meaning that a user can make many comments and a images may have many comments made on.

Controller Code

```
from flask import Flask
from flask import render_template, request, session
from flask import Blueprint, current_app

#declaring sub app with blueprint
comment_app = Blueprint('comment_app', __name__)
```

As with other components this one uses Flask's Blueprint interface to modulate project into seperate files and therefore isolates workspace, improves readability and most importantly eliminates conflicts.

```
@comment_app.route('/comment/<image_id>',methods=["POST"])
def comment(image_id):
        print("Hey")
        ## insert
        comment = request.form['comment']
        with psycopg2.connect(current_app.config['dsn']) as conn:
                crs=conn.cursor()
                crs.execute("insert into comments (user_id, image_id, time,␣
↪comment) values (%s, %s, now(), %s)", (session.get("user_id"), image_id,␣
↪comment))
```

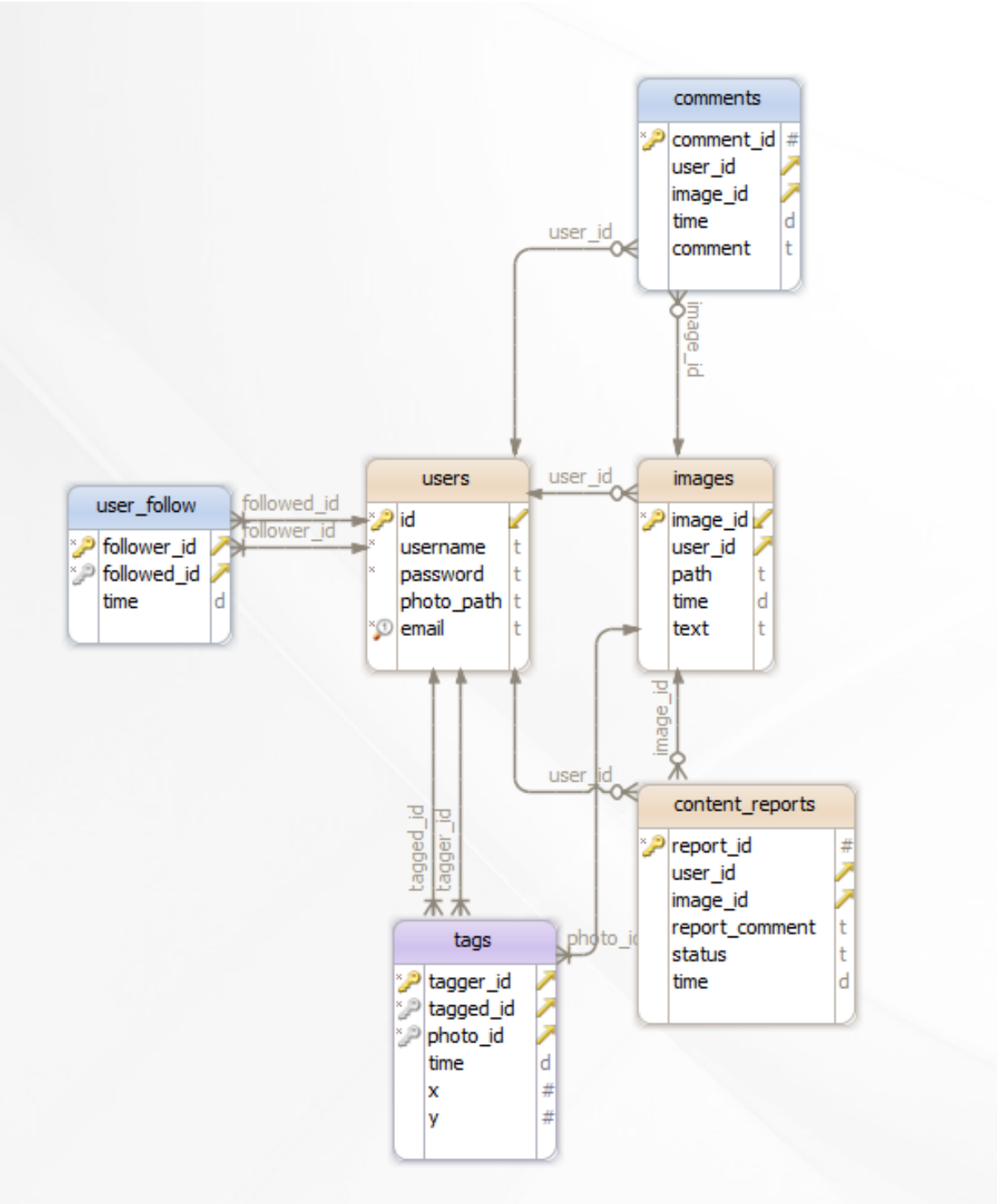


Fig. 3.1: ENTITY-RELATIONSHIP DIAGRAM OF MY PART

```
                data = conn.commit()

        return render_template('message.html', message = "Successfully commented..
↪")
```

This controller gets data from specific forms in image divs in home.html. Thus image_id parameter corresponds to the image_id parameter of the form action. Comment text is acquired via request api which parses request parameters and creates a dictionary named form.

- A disposable connection to database server is created via 'with' command which gets configuration from main application(current_app) settings.
- Create a cursor.
- Execute an SQL insertion.
- Commit the changes and save the result of query.

Function then returns a rendered template messages.html which is used to 'flash' results of actions throughout the entire project.

```
@comment_app.route('/comment_delete/<id>')
def comment_delete(id):
        ## delete
        #id = request.args.get('id')
        with psycopg2.connect(current_app.config['dsn']) as conn:
                crs=conn.cursor()
                crs.execute("delete from comments where comment_id = %s", (id))
                data = conn.commit()

        return render_template('message.html', message = "Comment deleted..")
```

When a user presses delete icon near a comment of his/her own it is routed to this route which gets id of the comment to be deleted from routing argument '<id>'. Then the function connects to database driver, instantiates a cursor, executes delete SQL query with id and commits to the database. Return a "message.html" template denoting that the message has been deleted.

```
@comment_app.route("/comment_update/<id>",methods=["POST"])
def comment_update(id):
        new_comment = request.form["new_comment"]
        with psycopg2.connect(current_app.config["dsn"]) as conn:
            crs = conn.cursor()
            crs.execute('update comments set time=now(),comment=%s where comment_
↪id=%s ',(new_comment,id))
            conn.commit()

return render_template("message.html",message="You have changed your comment␣
↪successfully")
```

The same procedure for delete hold true for update except that it is reached by update button in home.html. SQL query seeks the comment to be updated and changes its time and content. And returns the message.html template which flashes a success message.

Note: A non-existing id is not handled in update and delete operations, since user input can not be a non-existent id.

Danger: However by typing comment_delete/comment_update manually, a user may try to delete or update a non-existent entry in which server stops execution halfway informing user.

Danger: This component belongs to the early stages of the project so no session data is checked. Therefore anyone can delete/update any comment by typing comment_delete/<id> or comment_update/<id>. However, this behavior does not apply to the user interface as no delete button appears to user for comments which is not written by him/her.

```
@app.route('/')
def home_page():
        ### .....
        comments= []
        for img in data:
            crs.execute("select comment_id, user_id,image_id,time,comment,username␣
↪from comments join users on comments.user_id = users.ID where image_id=%s",
↪(img[0],))
            conn.commit()
            comments.append(crs.fetchall())
        ### ......
return render_template('home.html', current_time=now.ctime(), list = images,␣
↪images_app = images_app, comment_app = comment_app,comment_list=comments, likes␣
↪= userlikes,tags_app=tags_app,tags=tags)
```

Inside of home page root comments need to be passed in template in a manner that every image element has a comments list associated with itself(So it is basically a 2D-List of comments). This is achieved by joining `users` and `comments` table and filtering the query on `image_id` for each element in images to be shown on home page.

Note: It would be better not to execute the query for every element but to execute it once and map the result list to a 2D-List on photo_id.

3.1.3 Content Reports

Database Design

```
CREATE TABLE IF NOT EXISTS content_reports(
    report_id serial primary key,
    user_id INT REFERENCES users (ID) ON DELETE CASCADE,
    image_id INT REFERENCES images (image_id) ON DELETE CASCADE,
    report_comment text,
    status text,
    time date
);
```

Content report has

- a unique surrogate key: `report_id`
- a reference to the user who has issued the report `user_id`
- a reference to the image that has been reported `image_id`
- a text on the report cause by the issuer `report_comment`
- a status field whether if it is pending or accepted `status`
- time of the report issue

Controller Code

```
from flask import render_template, request, jsonify
from flask import Blueprint, current_app
import psycopg2

reports_app = Blueprint("reports_app",__name__)
```

As with other components this one uses Flask's Blueprint interface to modulate project into seperate files and therefore isolates workspace, improves readability and most importantly eliminates conflicts.

```
@reports_app.route('/initiate_report/<content_id>')
def initiate_report(content_id):

    with psycopg2.connect(current_app.config['dsn'])  as conn:
        crs = conn.cursor()
        crs.execute("select path from images where image_id=%s",(content_id))
        conn.commit()
        data = crs.fetchone()
    return render_template("report.html",content_id=content_id,content=data)
```

The route `initate_report/<content_id>` have an argument on which image is reported, and uses this object to select corresponding image via a disposable connection to application database. This function returns a template which shows up the aforementioned image with a form inquiring the cause of report and sends data to `report_content` route, the next element on the Content Reporting pipeline.

```
@reports_app.route('/report_content/<content_id>',methods=["POST"])
def report_content(content_id):
    report_text = request.form['report_text']
    status ='pending'
    with psycopg2.connect(current_app.config['dsn']) as conn:
        crs = conn.cursor()
        crs.execute("insert into content_reports (report_id,user_id, image_id,␣
↪report_comment, status, time) values (DEFAULT,%s, %s, %s, %s, now())",(1,content_
↪id,report_text,status))
        conn.commit()
    return render_template("message.html",message="Content successfully reported.")
```

The next function in Content Report system gets the argument `content_id` from the form on "Report" template page.

- `report_text = request.form['report']` gets users' report on the content.
- `status ='pending` hold the initial status: pending

A connection is established to the database and and Insert query is dispatched to fill in content_reports page which is later used to view and process issues.

Note: `user_id` being default is because of the website did not have session management when this feature has been added.

A quick fix on that line would be:

```
crs.execute("insert into content_reports (report_id,user_id, image_id, report_
↪comment, status, time) values (DEFAULT,%s, %s, %s, %s, now())",(session.get(
↪"user_id"),content_id,report_text,status))
```

When viewing issues page an administrator(a feature which is not implemented) can go two ways with report, either accept or reject the deletion proposal.

```
@reports_app.route('/issue_approval/<content_id>',methods=["POST"])
def issue_approval(content_id):
    with psycopg2.connect(current_app.config['dsn']) as conn:
        crs = conn.cursor()
        crs.execute("delete from images where image_id = %s",(content_id))
        conn.commit()
    return render_template("message.html",message="Content removed successfully.")
```

```
@reports_app.route('/issue_reject/<content_id>',methods=["POST"])
def issue_reject(content_id):
    with psycopg2.connect(current_app.config['dsn']) as conn:
        crs = conn.cursor()
        crs.execute("update content_reports set status='rejected' where image_id=%s
↪",(content_id))
        conn.commit()
    return render_template("message.html",message="Report rejected.")
```

If a deletion proposal is accepted, the form will go on to `issue_approval/<content_id>` route to delete image with the `content_id`. But, if a content report is rejected, its status will change from `pending` to `rejected`.

```
@app.route('/issues')
def issues():
    if session.get('logged_in')== None:
        return redirect(url_for("loginpage"))
    with psycopg2.connect(app.config['dsn']) as conn:
        crs = conn.cursor()
        crs.execute("select (username,image_id,report_comment,status,time) from␣
↪content_reports join users on content_reports.user_id= users.ID order by time")
        conn.commit()
        data = []
        ret = crs.fetchall()
        for tp in ret:
            str = tp[0]
            tmplist= []
            for s in str.split(','):
                tmplist.append(s)
            data.append(tmplist)
        print(data)
    return render_template("issues.html",data=data)
```

A join of `users` and `content_reports` are selected and passed into Issue template after a few formatting. Every element in result list which holds tuples is converted to string then split by delimiter "," and the result is a 2D-List.

3.1.4 Image Tags

Database Design

```
CREATE TABLE IF NOT EXISTS tags(
    tagger_id INT REFERENCES users (ID) ON DELETE CASCADE,
    tagged_id INT REFERENCES users(ID) ON DELETE CASCADE,
    photo_id INT REFERENCES images(image_id) ON DELETE CASCADE,
    time date,
    x INT,
    y INT,
    primary key (tagger_id,tagged_id,photo_id)
);
```

Image tags table consists of the following fields:

- a reference to the tagger's id: `tagger_id`
- a reference to the id of the user who has been tagged on image `tagged_id`
- a reference to the image that has been tagged `image_id`
- time of tagging
- x coordinate(percentage) of tag `x`
- y coordinate(percentage) of tag `y`
- a primary key consisting of id of tagger, tagged and image `primary key (tagger_id,tagged_id, photo_id)`

Controller Code

```
import psycopg2
from flask import Flask
from flask import render_template, request
from flask import Blueprint, current_app,session,redirect, url_for

#declaring sub app with blueprint
tags_app = Blueprint('tags_app', __name__)
```

As with other components this one uses Flask's Blueprint interface to modulate project into seperate files and therefore isolates workspace, improves readability and most importantly eliminates conflicts.

```
@tags_app.route('/add_tag/<photo_id>/', methods=["POST"])
def add_tag(photo_id):
    username = request.form["username"]
    x = request.form["x"]
    y = request.form["y"]
    with psycopg2.connect(current_app.config['dsn']) as conn:
        crs = conn.cursor()
        crs.execute("select ID from users where username=%s",(username,))
        conn.commit()
        tagged_id = crs.fetchone()
        if tagged_id == None:
            return render_template("message.html",message="User not found")
        ## if null show and error message
        crs.execute("insert into tags (tagger_id,tagged_id,photo_id,time,x,y)␣
↪values (%s,%s,%s,now(),%s,%s)",(session["user_id"],tagged_id,photo_id,x,y))
        conn.commit()
    return render_template('message.html',message="Successfully added tag")
```

:python:‘‘add_tag/<photo_id>‘ route gets a photo_id argument which holds the id of the image to be tagged. Following parameters are acquired from the form

- `username = request.form["username"]` holds the name of the user tagged.
- `x = request.form["x"]` holds the x coordinate that is clicked by tagger.
- `y = request.form["y"]` holds the y coordinate that is clicked by tagger.

On this controller two SQL queries are issued:

1. An select query to get id from username. If no user is matched then controller returns a message template which flashes `User not found`
2. A query that populates tags table with id of tagger (from session), id of tagged (from previous query), id of image,x,y (from form variables).

```
@tags_app.route('/update_tag/<photo_id>/', methods=["POST"])
def update_tag(photo_id):
```

```
    newUsername = request.form["username"]
    x = request.form["x"]
    y = request.form["y"]
    tagged_id=request.form["_id"]
    with psycopg2.connect(current_app.config['dsn']) as conn:
        crs = conn.cursor()
        crs.execute("select ID from users where username=%s",(newUsername,))
        newId = crs.fetchone()
        if newId == None:
            return render_template("message.html",message="User not found")
        print(tagged_id)

        ## if null show and error message
        crs.execute("update tags set tagged_id=%s,time=now(),x=%s,y=%s where␣
↪tagger_id=%s and tagged_id=%s and photo_id=%s  ",(newId[0],x,y,session["user_id
↪"],tagged_id,photo_id))
        conn.commit()
    return render_template('message.html',message="Successfully updated tag")
```

Update user controller works in the same fashion as `add_tag` does.

:python:‘‘update_tag/<photo_id>‘ route gets a photo_id argument which holds the id of the image to be tagged. Following parameters are acquired from the form

- `username = request.form["username"]` holds the name of the user tagged.
- `x = request.form["x"]` holds the x coordinate that is clicked by tagger.
- `y = request.form["y"]` holds the y coordinate that is clicked by tagger.

On this controller two SQL queries are issued:

1. An select query to get id from username. If no user is matched then controller returns a message template which flashes `User not found`
2. A query that updates id of tagger,x,y of the row that matches on primary key fields.

```
@tags_app.route('/delete_tag/<photo_id>/', methods=["POST"])
def delete_tag(photo_id,):
    tagged_id=request.form["_id"]
    with psycopg2.connect(current_app.config['dsn']) as conn:
        crs = conn.cursor()
        print(tagged_id)
        crs.execute("delete from tags where tagger_id=%s and tagged_id=%s and␣
↪photo_id=%s  ",(session["user_id"],tagged_id,photo_id))
        conn.commit()
    return render_template('message.html',message="Successfully deleted tag")
```

`delete_tag/<photo_id>` gets id of the photo that user wants to delete a tag on. Since a photo may have many tags a way to distinguish between tags was put in use, `tagged_id`. That way individual tags can be deleted. `tagged_id` field is gotten from a button when on clicked fills in hidden fields in form data with key `_id`.

```
@app.route('/')
def home_page():
        tags=[]
        for img in data:
                ### .....
                crs.execute("select username,tagged_id,time,x,y from tags join␣
↪users on users.ID = tags.tagger_id where photo_id=%s",(img[0],))
                conn.commit()
                tags.append(crs.fetchall())
                ### .....
        return render_template('home.html', current_time=now.ctime(), list =␣
↪images, images_app = images_app, comment_app = comment_app,comment_list=comments,
↪ likes = userlikes,tags_app=tags_app,tags=tags)
```

Inside of home page root tags need to be passed in template in a manner that every image element has a tags list associated with itself(So it is basically a 2D-List of tags). This is achieved by joining `users` and `tags` table and filtering the query on `photo_id` for each element in images to be shown on home page.

Note: It would be better not to execute the query for every element but to execute it once and map the result list to a 2D-List on photo_id.

3.1.5 Users and User Follow

Controller Code

I implemented this controller partially, so I left the part which was not written by me (except user block feature).

```
import psycopg2
from flask import Flask
from flask import render_template, request
from flask import Blueprint, current_app,session,redirect, url_for

#declaring sub app with blueprint
users_app = Blueprint('users_app', __name__)
```

As with other components this one uses Flask's Blueprint interface to modulate project into seperate files and therefore isolates workspace, improves readability and most importantly eliminates conflicts.

```
@users_app.route('/search_user/',methods=['GET'])
def search_user():
    with psycopg2.connect(current_app.config['dsn']) as conn:
        username = request.args.get('username')
        if(username == ""):
            username = " "
        print(type(username))
        crs = conn.cursor()
        crs.execute("select ID,username,photo_path from users where username like
↪%s",(username,))
        print(username)
        conn.commit()
        result = crs.fetchall()
    return render_template('search_results.html',result=result)
```

`search_user` is implementation of basic exact match search feature on username field. The form uses get method ,since it does not do any modifications on database. If no argument is provided , username is changed so that it can list every user registered.

```
@users_app.route('/show_profile/<user_id>')
def show_profile(user_id):
    with psycopg2.connect(current_app.config['dsn']) as conn:
        crs = conn.cursor()
        crs.execute("select ID,username,photo_path,email from users where ID = %s",
↪(user_id))
        conn.commit()
        result = crs.fetchone()
        crs.execute("select * from user_follow where follower_id=%s and  followed_
↪id=%s",(session.get("user_id"),user_id))
        conn.commit()
        follow_query=crs.fetchone()
        is_following = False if follow_query == None else True
        is_self = False
```

```
        if int(user_id) == session.get("user_id"):
            is_self = True # can not follow oneself
        crs.execute("select path from images where user_id =%s",(user_id))
        conn.commit()
        list_photos = crs.fetchall()
    return render_template('profile.html',result=result,is_following=is_following,
↪is_self=is_self,list_photos=list_photos)
```

This controller return a rendered profile page. Since anyone can view any profile it should support viewing any profile which is why it takes a `user_id` argument. On show profile section in home page user is simply routed to `show_profile/<user_id>` when `user_id` is `session.get("user_id")`.

Queries executed:

1. First query selects user information on given `user_id`
2. Second query selects the information on `user_follow` table so that target profile page can be rendered according to follow/unfollow situation between current user and and the user profile he/she views. `is_following` variable holds this information.
3. Third query selects paths to photos which are uploaded by the user with `user_id`

How is rendering modified:

- `is_following` variable change the rendering by changing between follow/unfollow buttons according to the current relation between current user and viewed user. If current user follows the viewed one than "Unfollow" button appears, otherwise a "Follow" button appears.
- `is_self` variable removes follow/unfollow buttons altogether since a user cannot unfollow himself/herself.

```
@users_app.route('/user_follow/<followed>')
def user_follow(followed):
     with psycopg2.connect(current_app.config['dsn']) as conn:
        crs = conn.cursor()
        crs.execute("insert into user_follow (follower_id,followed_id,time) values␣
↪(%s,%s,now())",(session["user_id"],followed))
        conn.commit()
     return render_template('message.html',message="Successfully followed")
```

This function allows current user to follow another user with id of `followed`.

```
@users_app.route('/user_unfollow/<followed>')
def user_unfollow(followed):
    with psycopg2.connect(current_app.config['dsn']) as conn:
        crs = conn.cursor()
        crs.execute("delete from user_follow where follower_id=%s and followed_id=
↪%s",(session["user_id"],followed))
        conn.commit()
    return render_template('message.html',message="Successfully unfollowed")
```

This function allows current user to unfollow another user with id of `followed`.

```
@users_app.route('/show_followers/<user_id>')
def show_followers(user_id):
    with psycopg2.connect(current_app.config['dsn']) as conn:
        crs = conn.cursor()
        crs.execute("select ID,username,photo_path from users where ID in (SELECT␣
↪follower_id from user_follow where followed_id = %s)",(user_id,))
        conn.commit()
        ulist =crs.fetchall()
    return render_template('user_list.html',ulist=ulist,user_id=user_id)
```

This function lists all of the followers of user with id of `user_id`. Gets every user with if they have their id in the set which is return by SQL subquery which get `follower_id` where `followed_id` is `user_id`.

```
@users_app.route('/show_followed/<user_id>')
def show_followed(user_id):
    with psycopg2.connect(current_app.config['dsn']) as conn:
        crs = conn.cursor()
        crs.execute("select ID,username,photo_path from users where ID in (SELECT␣
↪followed_id from user_follow where follower_id = %s)",(user_id,))
        conn.commit()
        ulist =crs.fetchall()
    return render_template('user_list.html',ulist=ulist,user_id=user_id)
```

This function lists all of the users followed by the user with id of `user_id`. Gets every user with if they have their id in the set which is return by SQL subquery which get `followed_id` where `follower_id` is `user_id`.

```
@users_app.route('/users_all')
def users_all():
    if session.get('logged_in')== None:
        return redirect(url_for("loginpage"))
    with psycopg2.connect(current_app.config['dsn']) as conn:
        crs = conn.cursor()
        session_userid = session['user_id']
        crs.execute("select Id, username from users where Id !=%s",(session_userid,
↪))
        conn.commit()
        fetched = crs.fetchall()
        crs.execute("select followed_id from user_follow where follower_id=%s",
↪(session_userid,))
        conn.commit()
        follows=crs.fetchall()
        follows = [user[0] for user in follows]

    return render_template('users_all.html', data = fetched,follows=follows)
```

This controller renders a page that lists all registered users.

1. First query selects every user except current one.
2. Second query selects followed user and creates a list of them and passes it to page so that followed users can have “Unfollow”; unfollowed users can have “Follow” button next to their username in list.

3.2 Parts Implemented by Rumeysa Bulut

User operations, event organizing, and creating groups is implemented by me.

3.2.1 General Database Design

3.2.2 User Operations

Database Design

```
CREATE TABLE IF NOT EXISTS users(
        ID serial primary key,
        username VARCHAR(50) NOT NULL,
        password text NOT NULL,
        photo_path text,
        email text NOT NULL UNIQUE
);
```

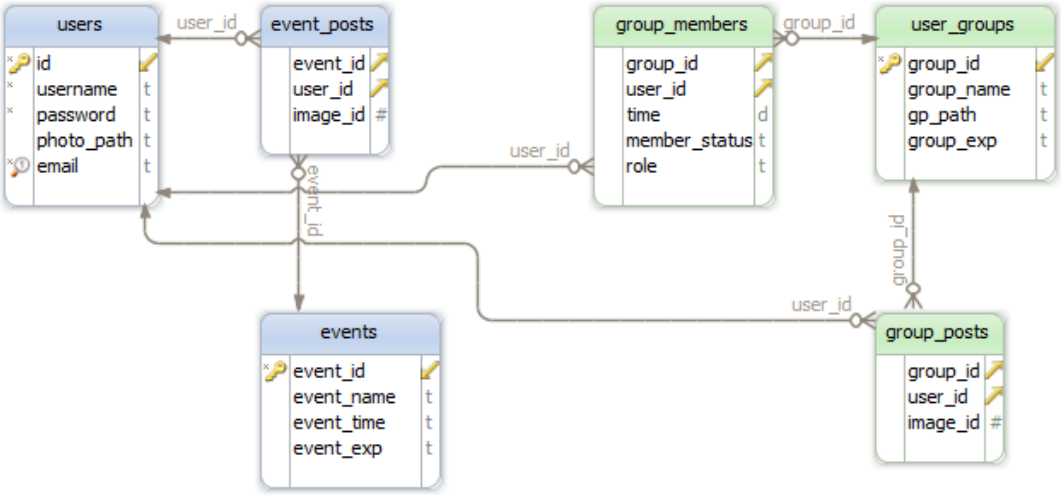


Fig. 3.2: ENTITY-RELATIONSHIP DIAGRAM OF MY PART

Users table has a serial primary key. This table is used by all members of the project.

Controller Code

```
from flask import Blueprint, current_app, render_template, request, session,␣
↪redirect, url_for
from passlib.hash import sha256_crypt
register_app = Blueprint('register_app', __name__)
```

We used Flask's Blueprint interface. Blueprint simplifies large application works by allowing us to separate the project into different files.

```
@register_app.route('/signup', methods = ['POST'])
def signup():
        data_username = request.form["username"]
        data_password = sha256_crypt.encrypt(request.form["password"])
        data_email = request.form["email"]
        print(data_password)
        with psycopg2.connect(current_app.config['dsn']) as conn:
                crs = conn.cursor()
                crs.execute("select username, email from users")
                data=crs.fetchall()
                for d in data:
                        if d[0] == data_username:
                                return render_template('message.html', message=
↪"The username is already exists")
                        elif d[1] == data_email:
                                return render_template('message.html', message=
↪"The email is already exists")

                crs.execute("insert into users (username, password, email) values (
↪%s, %s, %s)",(data_username,data_password,data_email))
                conn.commit()

        return render_template('message.html', message="Successfully registered")
```

This register section gets user information from signup.html. It hashes the password, and checks user name and

email whether they exists or not. If information satisfy the conditions, it adds the user to the database.

- A disposable connection to database server is created via 'with' command which gets configuration from main application(current_app) settings.
- Creates a cursor.
- Executes an SQL selection.
- Checks the data_username and data_email.
- Executes an SQL insertion.
- Commits the changes and save the result of the operation.

If the conditions fail in the control stage or the insertion is done successfully, function returns a rendered template message.html which says the result of the user action.

```
        @register_app.route('/login', methods=["POST"])
        def login():
                data_username = request.form["username"]
                data_password = sha256_crypt.encrypt(request.form["password"])

                with psycopg2.connect(current_app.config['dsn']) as conn:
                        crs = conn.cursor()
                        crs.execute("select ID from users where username = %s",␣
↪(data_username, ))
                        userid = crs.fetchone()

                        if userid:
                                crs.execute("select password, ID from users where␣
↪username = %s", (data_username,))
                                conn.commit()
                                data = crs.fetchone()

                        else:
                                return render_template('message.html', message=
↪"Invalid Credentials")
                        if (sha256_crypt.verify(request.form["password"],data[0])):
                                session['logged_in'] = True
                                session['user_id'] = data[1]
                                return redirect(url_for('home_page'))
                        else:
                                return render_template('login.html')

Login section proceeds in a similar way to sign up operation. It controls the␣
↪username and password are registered.

        * Creates a cursor.
        * Executes an SQL select to check the user is registered before.
        * If user is in the database, it gets the password and checks it.
        * Makes session changes.
```

The function returns the necessary pages under certain conditions. If username fails, a message says Invalid Credentials will be appear. If password fails, returns back to the login page. If entered information is true, directs users to the home page.

```
@register_app.route('/update_user',methods=["POST"])
def updateUser():
        id=session['user_id']
        data_username = request.form["username"]
        data_password = sha256_crypt.encrypt(request.form["password"])
        data_email = request.form["email"]
```

```
        with psycopg2.connect(current_app.config['dsn']) as conn:
                crs = conn.cursor()
                crs.execute("update users set username=%s, password=%s,email=%s␣
↪where ID = %s",(data_username,data_password,data_email,id))

        return render_template('message.html',message="Successfully updated")
```

This register section gets user information from update.html. If users want to update their information, this function gets current information from the form and the user ID from session.

- The function connects to the database driver.
- Creates a cursor
- Executes an SQL update with id.

Then returns a "message.html" template which says "Successfully updated."

```
@register_app.route('/remove_user',methods=["POST"])
def removeUser():
        data_username = request.form["username"]
        with psycopg2.connect(current_app.config['dsn']) as conn:
                crs = conn.cursor()
                crs.execute("delete from users where username = %s",(data_username,
↪))
                data = conn.commit()

        return render_template('login.html')
```

Deleting an account almost follows the same process with update section.

- The function connects to the database driver.
- Creates a cursor.
- Executes an SQL delete with username.
- Commits the changes to the database.

The function returns to the login page.

3.2.3 User Groups

Database Design

```
CREATE TABLE IF NOT EXISTS user_groups(
        group_id serial primary key,
        group_name text,
        gp_path text,
        group_exp text
);
```

Controller Code

```
@groups_app.route('/create_group')
def create_group():
        if not session.get('user_id'):
                return redirect(url_for('home_page'))

        with psycopg2.connect(current_app.config['dsn']) as conn:
                crs = conn.cursor()
                crs.execute("select * from users where ID in (select followed_id␣
↪from user_follow where follower_id = %s)", (session['user_id'],))
```

```
                conn.commit()
                data = crs.fetchall()

        return render_template('listfollowed.html', data=data)
```

Creating groups feature is activated after users logged in. When users click the new group icon on the dropdown menu, a new page will be appear. On this page, all people they followed will be listed. They can determine the group name and the group description. Then, they can select the members of the group among the listed people.

- At first, the function controls the session.
- If user is logged in, it connects to the database.
- Creates a cursor.
- Executes an SQL select query to list the followed users.

The function returns to the group creation page.

```
@groups_app.route('/addtogroup',  methods = ['POST'])
def addtogroup():
        name = request.form['name']
        desc = request.form['desc']
        members = request.form.getlist('members')

        with psycopg2.connect(current_app.config['dsn']) as conn:
                crs = conn.cursor()
                crs.execute("insert into user_groups (group_name, gp_path, group_
↪exp) values (%s, %s, %s) returning group_id", (name, "/", desc))
                conn.commit()
                data = crs.fetchone()
                id = data[0]

                for m in members:
                        crs.execute("insert into group_members(group_id, user_id,␣
↪time, member_status, role) values (%s, %s, now(), 'active', 'admin')", (id, m))
                        conn.commit()

        return redirect(url_for('groups_app.show_group', group_id = id))
```

This function does the main job. Creating group with specified name and description and adding the selected users to this group is processed in this function.

- It gets the information from the form that is in the previous stage.
- Then connects to the database and creates a cursor.
- It inserts the group with name and description with an SQL insert and gets the group id.
- At last, it inserts the selected users into the created group.

After the operation is done, it returns to the page which shows the newly created group.

```
@groups_app.route('/show_group/<group_id>')
def show_group(group_id):
        with psycopg2.connect(current_app.config['dsn']) as conn:
                crs = conn.cursor()
                crs.execute("select u.username, u.id from group_members as g inner␣
↪join users as u on u.id = g.user_id where group_id = %s", (group_id, ))
                memberdata = crs.fetchall()
                crs.execute("select group_name, gp_path, group_exp from user_
↪groups where group_id = %s", (group_id,))
                data = crs.fetchone()
                conn.commit()
        return render_template('groupinfo.html', data=data, memberdata=memberdata)
```

This function shows only the group which has been just created.

- It gets the group id from the previous function, addtogroup.
- The function does 2 SQL select query to list the group and its members.

It returns to the groupinfo.html to display the group information with its members.

```
@groups_app.route('/allgroups')
def allgroups():
        with psycopg2.connect(current_app.config['dsn']) as conn:
                crs = conn.cursor()
                crs.execute("select group_name, group_exp, group_id from user_
↪groups")
                data = crs.fetchall()
                crs.execute("select u.username, u.id from group_members as g inner␣
↪join users as u on u.id = g.user_id")
                memberdata = crs.fetchall()
        return render_template('allgroups.html',data=data,memberdata=memberdata)
```

Users can list the current groups by clicking the groups icon on the dropdown menu.

- The function selects all groups and their members.

It sends the group data and member data to allgroups.html.

```
@groups_app.route('/delete_member/<id>')
def delete_member(id):
        with psycopg2.connect(current_app.config['dsn']) as conn:
                crs = conn.cursor()
                crs.execute("delete from group_members where user_id = %s", id)
                conn.commit()
        return render_template('message.html', message="Successfully removed.")
```

Users can delete a member from a group after they create the group by clicking cross sign.

- It gets id.
- Performs the delete operation according to the id.

Then the function returns a rendered template message.html which gives a message that says removing is successful.

```
@groups_app.route('/delete_group/<id>')
def delete_group(id):
        with psycopg2.connect(current_app.config['dsn']) as conn:
                crs = conn.cursor()
                crs.execute("delete from user_groups where group_id = %s", (id, ))
                conn.commit()
        return redirect(url_for('groups_app.allgroups'))
```

Users also delete a group by clicking the cross sign in the page which lists all groups.

- It gets the id.
- Performs delete operation.

Then it returns to the page lists all groups.

```
@groups_app.route('/updateform')
def updateform():
        return render_template('update_group.html')

@groups_app.route('/update_group',methods=["POST"])
def update_group():
        old_name = request.form['oldname']
        new_name = request.form['name']
```

```
        desc = request.form['desc']
        with psycopg2.connect(current_app.config['dsn']) as conn:
                crs = conn.cursor()
                crs.execute("update user_groups set group_name=%s, group_exp=%s␣
↪where group_name = %s", (new_name, desc, old_name, ))

        return redirect(url_for('groups_app.allgroups'))
```

This 2 functions allow the users to update their groups name and description. First one returns to the update_group.html to get the current information. Second one gets the information from the update_group.html.

- Second one connects to the database.
- It performs the update operation with an SQL update.

Then it redirects to the page that lists all groups.

3.2.4 Events

Database Design

```
CREATE TABLE IF NOT EXISTS events(
        event_id serial primary key,
        event_name text,
        event_time text,
        event_exp text
);
```

Controller Code

```
@events_app.route('/create_event', methods = ['POST'])
def create_event():
        new_name = request.form["event-name"]
        explan = request.form["event-exp"]
        time_event = request.form["event-time"]
        with psycopg2.connect(current_app.config['dsn']) as conn:
                crs = conn.cursor()
                crs.execute("insert into events (event_name, event_exp, event_
↪time) values (%s, %s, %s)", (new_name, explan, time_event))
                conn.commit()
        return redirect(url_for('events_app.show_events'))
```

create_event function allows the users to organize new events. It works quite similar to the create_group function. Users can use this feature by clicking the new event icon on the dropdown menu.

- It gets the data from the form.
- Connects to the database.
- Creates a cursor.
- Executes an SQL insertion to create the event.

Then, the function redirects to the page which shows all events with their information.

```
@events_app.route('/show_events')
def show_events():
        if session.get('logged_in')== None:
                return redirect(url_for("loginpage"))
        with psycopg2.connect(current_app.config['dsn']) as conn:
                crs = conn.cursor()
```

```
                crs.execute("select event_name, event_exp, event_time, event_id␣
↪from events")
                conn.commit()
                data = crs.fetchall()
        return render_template('allevents.html', data=data)
```

Users can display the events they created. This feature is activated after users logged in as in the user groups sections.

- It controls the session.
- If the user is logged in, it executes an SQL select query.

It sends the data to the allevents.html to show all events with their information.

```
@events_app.route('/delete_event/<id>')
def delete_event(id):
        with psycopg2.connect(current_app.config['dsn']) as conn:
                crs = conn.cursor()
                crs.execute("delete from events where event_id = %s", (id, ))
                conn.commit()
        return redirect(url_for('events_app.show_events'))
```

Deleting an event is also possible. Users can delete the event by clicking the cross sign. Thus, the function gets the event id.

- Connects to the database.
- Creates a cursor.
- Executes an SQL deletion to remove the event from the database using id.

Then, the function redirects to the page which shows all events with their information.

```
@events_app.route('/updateEvent')
def updateEvent():
        return render_template('update_event.html')

@events_app.route('/update_event',methods=["POST"])
def update_event():
        old_name = request.form["old-name"]
        new_name = request.form["event-name"]
        explan = request.form["event-exp"]
        time_event = request.form["event-time"]
        with psycopg2.connect(current_app.config['dsn']) as conn:
                crs = conn.cursor()
                crs.execute("update events set event_name=%s, event_exp=%s, event_
↪time=%s where event_name = %s", (new_name, explan, time_event, old_name, ))

        return redirect(url_for('events_app.show_events'))
```

Updating the event also works very similar to the group section. After the pencil icon is clicked, a form page comes to the screen. Users can fulfill the form with current information. Second function does the main work.

- It gets data from the from.
- Connects to the database.
- Creates a cursor.
- Executes an SQL update operation to renew the event.

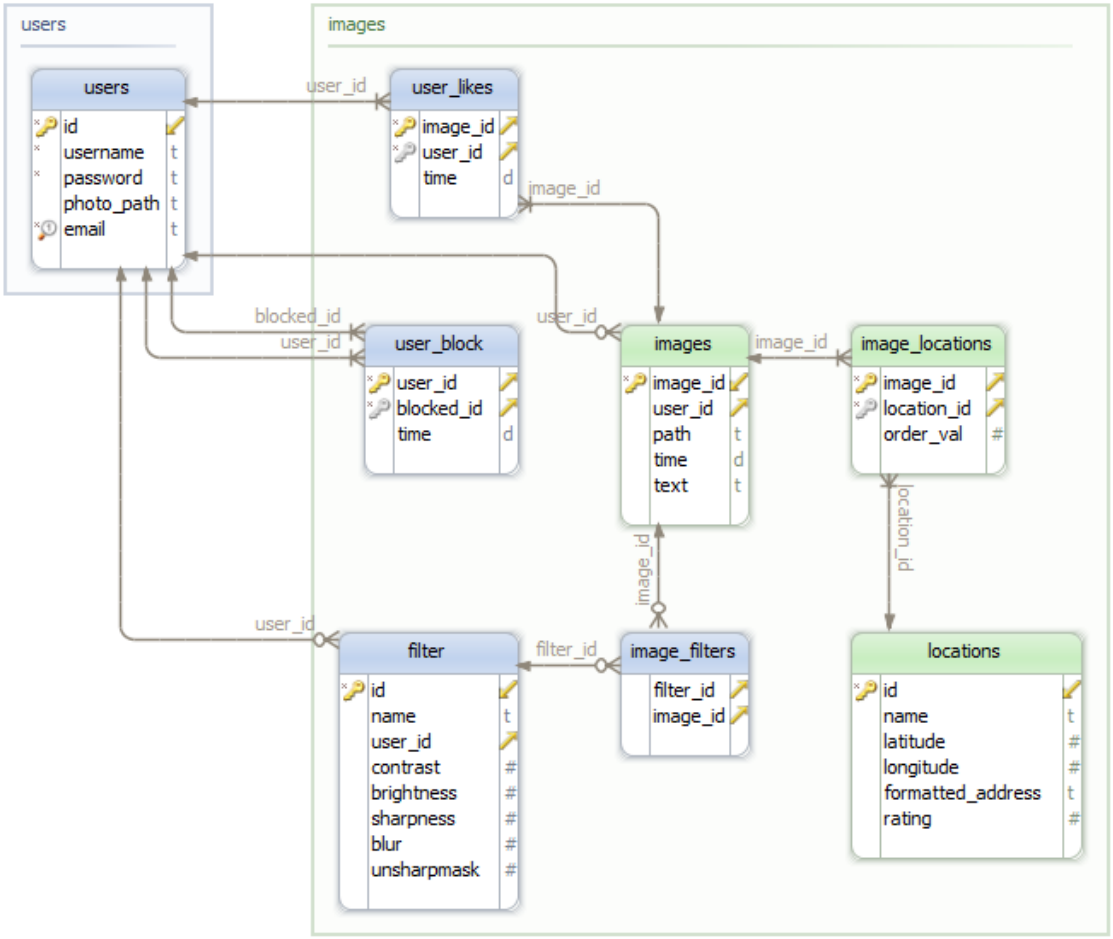
Then, the function redirects to the page which shows all events with their information.

3.3 Parts Implemented by Alim Özdemir

You can find all informations about images, locations and filters here.

3.3.1 General Database Design

ER DIAGRAM



Images

```
CREATE TABLE IF NOT EXISTS images(
    image_id serial primary key,
    user_id int REFERENCES users(ID) ON DELETE CASCADE,
    path text ,
    time date ,
    text text
);
```

The table above has critical fields that effects the whole system. Image_id is primary key and forgein key for other associated tables. A user could have many images. Therefore the table has one to many relationship with user_id. Path field is the absolute url of uploaded image. Text field is description of that image.

Associated tables

```
CREATE TABLE IF NOT EXISTS user_likes(
    user_id int REFERENCES users (ID) ON DELETE CASCADE,
    image_id int REFERENCES images (image_id) ON DELETE CASCADE,
    time date,
    primary key(image_id, user_id)
);

CREATE TABLE IF NOT EXISTS user_block(
    user_id int REFERENCES users (ID) ON DELETE CASCADE,
    blocked_id int REFERENCES users (ID) ON DELETE CASCADE,
    time date,
    primary key(user_id, blocked_id)
);
```

The tables above are associated tables between users and images.

Locations

```
CREATE TABLE IF NOT EXISTS locations(
    Id serial primary key,
    name text,
    latitude numeric,
    longitude numeric,
    formatted_address text,
    rating real
);
CREATE TABLE IF NOT EXISTS image_locations(
    image_id int REFERENCES images (image_id) ON DELETE CASCADE,
    location_id int REFERENCES locations (Id) ON DELETE CASCADE,
    order_val int DEFAULT 0,
    primary key (image_id, location_id)
);
```

Locations table represents of the geographic information of taken image. Sending text field to googlemaps REST API returns the fields latitude, longitude, formatted_address. Also, there exists many to many relationship between images and locations. (image_locations is the pivot table)

Filters

```
CREATE TABLE IF NOT EXISTS filter(
    id serial primary key,
    name text,
    user_id int REFERENCES users (ID) ON DELETE CASCADE,
    Contrast int,
    Brightness int,
    Sharpness int,
    Blur int,
    UnsharpMask int
);
CREATE TABLE IF NOT EXISTS image_filters(
    filter_id int REFERENCES filter(id) ON DELETE RESTRICT,
    image_id int REFERENCES images(image_id) ON DELETE CASCADE
);
```

The filter table stores information about user's custom data with pillow package of python. A user could have many filters. And, there exists many to many relationship between images and filters. (image_filters is the pivot table)

3.3.2 Implementation using Flask on Python

Controllers

Inserting of Image & Location & Filter

```
@images_app.route('/upload', methods = ['POST'])
def upload_post():
    if not session.get('user_id'):
        return redirect(url_for('home_page'))

    comment = request.form['comment']
    location = request.form['location']
    upload_file = request.files['image']
    filters = request.form['filters']
    contrast = request.form['contrast']
    brightness = request.form['brightness']
    sharpness = request.form['sharpness']
    blur = request.form['blur']
    unsharpmask = request.form['unsharpmask']
    session_user_id = session['user_id']
    if upload_file:
        upload_file.save(os.path.join('static/uploads', upload_file.filename))
    else:
        return render_template('message.html', message = "Please select an image..
↪")

    img = Image.open(os.path.join('static/uploads', upload_file.filename))

    needToSave = 0
    if blur != "0":
        img = img.filter(ImageFilter.GaussianBlur(float(blur)))
        needToSave = 1

    if unsharpmask != "0":
        img = img.filter(ImageFilter.UnsharpMask(float(unsharpmask)))
        needToSave = 1

    if sharpness != "0" :
        enhancer = ImageEnhance.Sharpness(img)
        img = enhancer.enhance(float(sharpness))
        needToSave = 1

    if contrast != "0" :
        enhancer = ImageEnhance.Contrast(img)
        img = enhancer.enhance(float(contrast))
        needToSave = 1

    if brightness != "0" :
        enhancer = ImageEnhance.Brightness(img)
        img = enhancer.enhance(float(brightness))
        needToSave = 1

    if needToSave == 1:
        img.save(os.path.join('static/uploads', upload_file.filename))

    print(filters)
```

```
    gmaps = googlemaps.Client(key='AIzaSyDurbt3tU9F8lDMqyHAnXVjCPphapNu0FM')
    with psycopg2.connect(current_app.config['dsn']) as conn:
        crs=conn.cursor()

        crs.execute("insert into images (user_id, path, time, text) values (%s,%s,
↪now(), %s) RETURNING image_id", (session_user_id, upload_file.filename, comment))
        image_id = crs.fetchone()[0] #Get image id
        #filter part
        if filters == "0":
            crs.execute('insert into filter (name, user_id, contrast, Brightness,
↪Sharpness, Blur, UnsharpMask) values (%s, %s, %s, %s, %s, %s, %s) RETURNING id',
↪ ("Saved Settings", session_user_id,contrast, brightness, sharpness, blur,
↪unsharpmask))
            filter_id = crs.fetchone()[0]
            crs.execute('insert into image_filters (image_id, filter_id) values (
↪%s, %s)', (image_id, filter_id))
            conn.commit()
        else:
            crs.execute('update filter set contrast = %s, brightness = %s,
↪sharpness = %s, blur = %s, unsharpmask = %s where id = %s and user_id = %s',
↪(contrast, brightness, sharpness, blur, unsharpmask, filters, session_user_id))
            filter_id = filters
            crs.execute('insert into image_filters (image_id, filter_id) values (
↪%s, %s)', (image_id, filter_id))
            conn.commit()
        if location:
            locs = location.split(',')
            order = 0
            #location check
            for loc in locs:
                #print(loc)
                crs.execute("select * from locations where name = %s", (loc,))
                loc_data = crs.fetchone()
                loc_id = 0
                #get location id with insert or select
                if loc_data:
                    crs.execute('update locations set rating = rating + 1 where Id=
↪%s', ([loc_data[0]]))
                    loc_id = loc_data[0]
                else:
                    gcode = gmaps.geocode(loc)
                    formatted = gcode[0]['formatted_address']
                    location = gcode[0]['geometry']['location']
                    lng = location['lng']
                    lat = location['lat']
                    crs.execute('insert into locations (name, latitude, longitude,
↪formatted_address, rating) values (%s, %s, %s, %s, %s) RETURNING Id', (loc, lat,
↪lng, formatted, 1))
                    loc_id = crs.fetchone()[0] #Get last insertion id

                #add it to image_locations relation table
                crs.execute('insert into image_locations (image_id, location_id,
↪order_val) values (%s, %s, %s)', (image_id, loc_id, order))
                order = order + 1

        #notification insertion will use the logged user's information after the
↪respective functionality is added - Halit
        crs.execute("select photo_path, username from users where Id !=%s",
↪(session['user_id'],))
        data = crs.fetchone()
        crs.execute("insert into notifications(user_id, notifier_id, notifier_name,
↪ icon, details, read_status, follow_status) values (%s, %s, %s, %s, %s, %s, %s)",
↪ (session['user_id'], session['user_id'], data[1] ,data[0], comment , 'FALSE',
↪'TRUE'))
```

```
        data = conn.commit()

    return render_template('message.html', message = "Uploaded..")
```

This action is responsible for

1. processing the image using *pillow*,
   (a) using user's existing filters
   (b) creating new filter data
2. getting geographic data from googlemaps REST API,
3. uploading the image,
4. storing its data to the database
5. notifying the user/users about the image

The user selects the image that should be filtered and uploaded to system along with the informations such as description about the image, location of the image, desired filter informations for *pillow* library to process, and then uploads the image. The action checks about if user is logged in, and if the file input isn't empty. Applies image processing via *pillow* library using submitted filter data. Gets geographic data sending a query to googlemaps API via location input.

```
@images_app.route('/image_delete/<id>')
def image_delete(id):
    #id = request.args.get('id')
    with psycopg2.connect(current_app.config['dsn']) as conn:
        crs=conn.cursor()
        crs.execute("delete from images where image_id = %s", (id))
        data = conn.commit()

    return render_template('message.html', message = "Image deleted..")
```

This action deletes the image and its associated relations that are cascade.

```
@images_app.route('/image_update', methods = ['POST'])
def image_update():
    #inline editable plugin gives pk and value
    id = request.form['pk']
    newText = request.form['value']
    data = ""
    with psycopg2.connect(current_app.config['dsn']) as conn:
        crs=conn.cursor()
        crs.execute("update images set text=%s where image_id = %s", (newText, id))
        data = conn.commit()
        return jsonify(data)

    return jsonify(0)
```

This JSON action updates the text field of the image.

```
@images_app.route('/image_like', methods = ['POST'])
def image_like():
    if not session.get('user_id'):
        return redirect(url_for('home_page'))

    id = request.form['id']
    user_id = session['user_id']

    with psycopg2.connect(current_app.config['dsn']) as conn:
        crs=conn.cursor()
        crs.execute("select * from user_likes where user_id = %s and image_id = %s
↪", (user_id, id))
```

```
        exist = crs.fetchone()
        if exist:
            return jsonify(-1) #already liked.
        else:
            crs.execute("insert into user_likes (user_id, image_id, time) values (
↪%s, %s, now())", (user_id, id))
            data = conn.commit()
    return jsonify(1)
```

This JSON action inserts a user like to the user_like table.

```
@images_app.route('/image_unlike', methods = ['POST'])
def image_unlike():
    if not session.get('user_id'):
        return redirect(url_for('home_page'))
    id = request.form['id']
    user_id = session['user_id']

    with psycopg2.connect(current_app.config['dsn']) as conn:
        crs=conn.cursor()
        crs.execute("select * from user_likes where user_id = %s and image_id = %s
↪", (user_id, id))
        exist = crs.fetchone()
        if exist:
            crs.execute("delete from user_likes where user_id = %s and image_id =
↪%s", (user_id, id))
            data = conn.commit()
        else:
            return jsonify(-1)
    return jsonify(1)
```

This JSON action deletes a user like from the user_like table.

```
@images_app.route('/update_delete_loc_save', methods = ['POST'])
def update_delete_loc_save():
    id = request.form['id']
    locs = request.form['locs']
    locations = locs.split(',')

    gmaps = googlemaps.Client(key='AIzaSyDurbt3tU9F8lDMqyHAnXVjCPphapNu0FM')
    #collect updated or inserted ids
    collect = []

    with psycopg2.connect(current_app.config['dsn']) as conn:
        crs=conn.cursor()
        for loc in locations:
            crs.execute("select * from locations where name = %s", (loc,))
            loc_data = crs.fetchone()
            if loc_data:
                crs.execute('update locations set rating = rating + 1 where Id = %s
↪', ([loc_data[0]]))
                collect.append(loc_data[0])
            else:
                gcode = gmaps.geocode(loc)
                formatted = gcode[0]['formatted_address']
                location = gcode[0]['geometry']['location']
                lng = location['lng']
                lat = location['lat']
                crs.execute('insert into locations (name, latitude, longitude,␣
↪formatted_address, rating) values (%s, %s, %s, %s, %s) RETURNING Id', (loc, lat,␣
↪lng, formatted, 1))
                loc_id = crs.fetchone()[0] #Get last insertion id
                collect.append(loc_id)
```

```
        crs.execute('select location_id from image_locations where image_id = %s',␣
↪(id))
        currentLocs = crs.fetchall()

        #tuple array to int array
        currentLocsInt = []
        for cur in currentLocs:
            currentLocsInt.append(cur[0])

        finded = []
        for cur in collect:
            if cur not in currentLocsInt:
                crs.execute('insert into image_locations (image_id, location_id)␣
↪values (%s, %s)', (id, cur))
                finded.append(cur)
        #Delete from database that not match
        for cur in currentLocsInt:
            if cur not in collect:
                crs.execute('delete from image_locations where image_id = %s and␣
↪location_id = %s', (id, cur))

        #get all locations and update order
        crs.execute('select location_id from image_locations where image_id = %s',␣
↪(id))
        updateLocs = crs.fetchall()
        order = 0
        for u in updateLocs:
            crs.execute('update image_locations set order_val = %s where image_id␣
↪= %s', (order, id))
            order = order + 1
        conn.commit()
    return render_template('message.html', message = "Locations updated..")
```

This action contains all operations (such as updating, inserting, deleting) about the location using *tagsinput* library.

```
@images_app.route("/locations")
def locations():
    if not session.get('user_id'):
        return redirect(url_for('home_page'))
    with psycopg2.connect(current_app.config['dsn']) as conn:
        crs=conn.cursor()
        crs.execute('select * from locations order by rating desc')
        data = crs.fetchall()
    return render_template('locations.html', list = data)
```

This action lists all locations that persisted on the database.

```
@images_app.route('/remove_location/<id>')
def remove_location(id):

    with psycopg2.connect(current_app.config['dsn']) as conn:
        crs=conn.cursor()
        crs.execute('delete from locations where Id = %s', (id))
        conn.commit()

    return render_template('message.html', message = "Location has been removed␣
↪from database")
```

This action deletes given location with all of its associated data from the database.

```
@images_app.route('/location/<name>')
def location(name):
    with psycopg2.connect(current_app.config['dsn']) as conn:
        crs=conn.cursor()
        crs.execute('select * from locations where name = %s', (name,))
        data = crs.fetchone()
        if data:
            crs.execute('select count(*) from image_locations where location_id =
↪%s', ([data[0]]))
            count = crs.fetchone()[0]
        else:
            return render_template('message.html', message="No location with '{}'␣
↪name".format(name))
    return render_template('location.html', data = data, count = count)
```

This action used to show detailed view of the given location.

```
@filters_app.route('/filter/index')
def index():
    if not session.get('user_id'):
        return redirect(url_for('home_page'))

    session_user_id = session['user_id']

    with psycopg2.connect(current_app.config['dsn']) as conn:
        crs = conn.cursor()
        crs.execute("select id,name from filter where user_id = %s", (session_user_
↪id,))
        data = crs.fetchall()

    return render_template('filter_index.html', list = data)
```

This action lists the saved filters of logged in user.

```
@filters_app.route('/filter/fetch', methods = ['POST'])
def fetch():
    if not session.get('user_id'):
        return redirect(url_for('home_page'))

    id = request.form['id']
    session_user_id = session['user_id']

    with psycopg2.connect(current_app.config['dsn']) as conn:
        crs = conn.cursor()
        crs.execute("select * from filter where user_id = %s and id = %s",␣
↪(session_user_id,id))
        data = crs.fetchone()

    return jsonify(data)
```

This JSON action gets a single existing filter.

```
@filters_app.route('/filter/delete', methods = ['POST'])
def delete():
    if not session.get('user_id'):
        return redirect(url_for('home_page'))

    id = request.form['id']
    session_user_id = session['user_id']
    if id == "1":
        return redirect(url_for('home_page'))
```

```
    with psycopg2.connect(current_app.config['dsn']) as conn:
        crs = conn.cursor()
        crs.execute("select * from filter where user_id = %s and id = %s",␣
↪(session_user_id,id))
        data = crs.fetchone()
        if data:
            crs.execute("delete from filter where user_id = %s and id = %s",␣
↪(session_user_id, id))
            conn.commit()
        else:
            return render_template('message.html', message = "No record has found.
↪")

    return render_template('message.html', message = "filter deleted")
```

This action deletes the saved filter from the database.

```
@filters_app.route('/filter/update', methods = ['POST'])
def update():
    if not session.get('user_id'):
        return jsonify(None)

    id = request.form['id']

    contrast = request.form['contrast']
    brightness = request.form['brightness']
    sharpness = request.form['sharpness']
    blur = request.form['blur']
    unsharpmask = request.form['unsharpmask']

    session_user_id = session['user_id']

    with psycopg2.connect(current_app.config['dsn']) as conn:
        crs = conn.cursor()
        crs.execute("select * from filter where user_id = %s and id = %s",␣
↪(session_user_id,id))
        data = crs.fetchone()
        #update
        if data:
            crs.execute('update filter set contrast = %s, brightness = %s,␣
↪sharpness = %s, blur = %s, unsharpmask = %s where id = %s and user_id = %s',␣
↪(contrast, brightness, sharpness, blur, unsharpmask, id, session_user_id))
            conn.commit()

    return jsonify(True)
```

This JSON action updates the existing filter data.

Views

Images Views

/templates/home.html

/templates/upload.html

Location Views

/templates/upload.html

/templates/locations.html

/templates/location.html

/templates/update_loc.html

Filter Views

/templates/upload.html

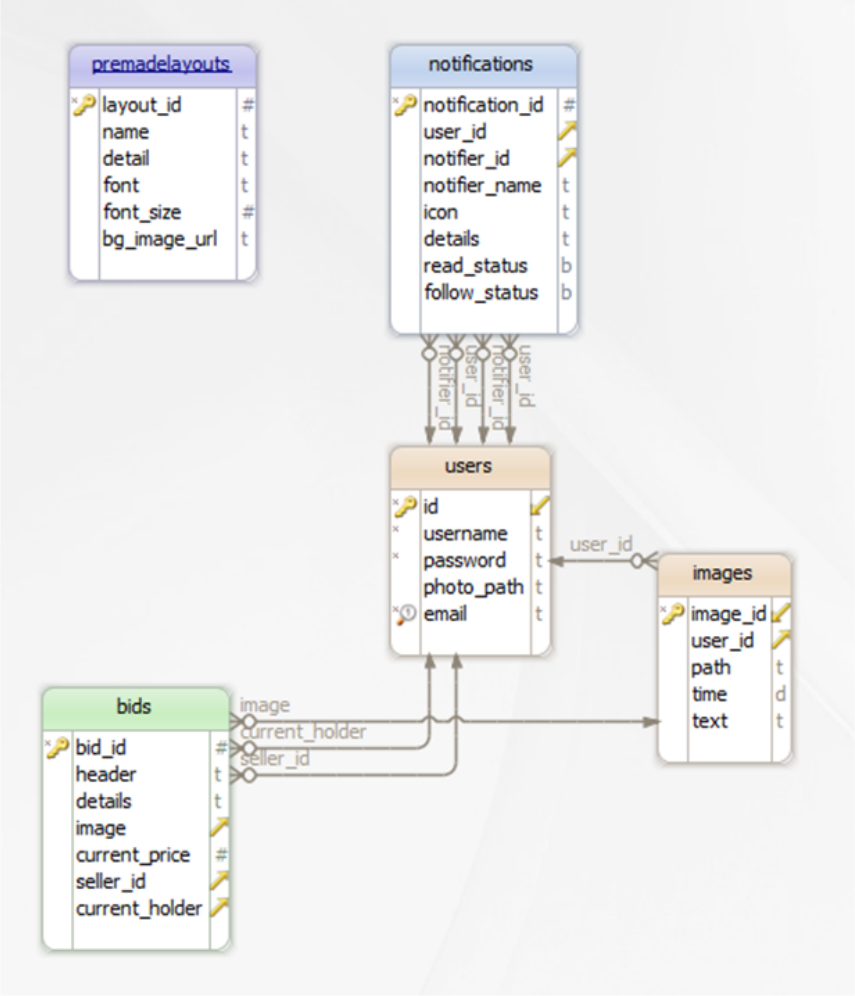
/templates/filter_index.html

3.4 Parts Implemented by Halit Uyanık

Includes notification, bidding and premadelayout pages.

3.4.1 General Database Design

Entities and Relations Graph



3.4.2 Blueprint

Blueprint is used for all methods and applications in pages. Below you can see an example of creating a new app.

An example blueprint in name 'notific_app' is created. This is used in server.py and eventually in html itself to call python functions.

```
notific_app = Blueprint('notific_app', __name__)
```

3.4.3 Notifications

Database Design

Notifications table includes primary key, two foreign keys for the users table. If a user is removed from the database then his/her notifications will also get removed.

```
CREATE TABLE IF NOT EXISTS notifications(
        notification_id serial primary key,
        user_id int REFERENCES users(ID) ON DELETE CASCADE,
        notifier_id int REFERENCES users(ID) ON DELETE CASCADE,
        notifier_name text,
        icon text,
        details text,
        read_status boolean,
        follow_status boolean
);
```

Controller Code

Notification pages uses 2 methods from controller.

- *Notification_delete*
- *Status_update*

Notification_delete

Notification_delete controller has an id parameter which is used to delete an existing notification from the server. After the operation is done user is directed to a message page.

```
@notific_app.route('/notification_delete/<id>')
def notification_delete(id):
        with psycopg2.connect(current_app.config['dsn']) as conn:
                        crs=conn.cursor()
                        crs.execute("delete from notifications where notification_
↪id = %s", (id))
                        data = conn.commit()

        return render_template('message.html', message = "Notification deleted..")
```

Status_update

Status_update controller has no parameter and a 'GET' method. It is invoked from user interface and takes the id and stat values from html request. Then according to the value of stat it updates the notification's status with that id to either TRUE or FALSE.

```
@notific_app.route('/status_update/', methods = ['GET'])
def status_update():
        id = request.args.get('id')
        stat = request.args.get('status')
        with psycopg2.connect(current_app.config['dsn']) as conn:
                        crs=conn.cursor()
                        if stat == "False":
```

```
                                crs.execute("update notifications set read_status␣
↪= TRUE where notification_id = %s", (id))
                        elif stat == "True":
                                crs.execute("update notifications set read_status␣
↪= FALSE where notification_id = %s", (id))
                        data = conn.commit()

        return render_template('message.html', message = "Notification status␣
↪updated..")
```

3.4.4 Bidding

Database Design

Bids table includes a serial primary key, 3 references; 2 to users table and 1 to images table. Bid name and details are stored in text column. Current bidded price is stored in numerical form. If the image for the bid is removed from the server, then the bidding will no longer exist. Also a user cannot remove his/her account without removing their bidding first.

```
CREATE TABLE IF NOT EXISTS bids(
        bid_id serial primary key,
        header text,
        details text,
        image int REFERENCES images(image_id) ON DELETE CASCADE,
        current_price numeric,
        seller_id int REFERENCES users(ID) ON DELETE RESTRICT,
        current_holder int REFERENCES users(ID) ON DELETE RESTRICT
);
```

Controller Code

Bidding page includes three controller:

- *Add_new_bid*
- *Update_bid*
- *Delete_bid*

Add_new_bid

Add_new_bid is a function which is used for POST methods. It takes new bids from the user interface as form info, processes these data, saves the image to server as both physical file and database info, then inserts the new bid into the database.

```
@bidding_app.route('/add_new_bid/', methods = ['POST'])
def add_new_bid():
        name = request.form['item_name']
        details = request.form['description']
        image = request.files['image']
        price = request.form['price']
        seller_id = session['user_id']
        current_holder = session['user_id']

        image.save(os.path.join('static/uploads', image.filename))

        with psycopg2.connect(current_app.config['dsn']) as conn:
                        crs=conn.cursor()
```

```
                        crs.execute("insert into images (user_id, path, time,␣
↪text) values (%s, %s, now(), %s) RETURNING image_id", (2, image.filename,␣
↪details))
                        im_id = crs.fetchone()
                        #print(im_id[0])
                        crs.execute("insert into bids (header, details, image,␣
↪current_price, seller_id, current_holder) values (%s, %s, %s, %s, %s, %s)",␣
↪(name, details, im_id[0], price, seller_id, current_holder))
                        conn.commit()

        return render_template('message.html', message = "Bid Successfully Added!")
```

Update_bid

Has one parameter <id>, and is used for 'POST' method. Takes the new price from form and checks the current price of the item. If the old price is higher the request is invalid, otherwise the bid is updated with the new entered one.

```
@bidding_app.route('/update_bid/<id>', methods = ['POST'])
def update_bid(id):
        new_price = request.form['price']
        with psycopg2.connect(current_app.config['dsn']) as conn:

                crs=conn.cursor()
                crs.execute("select current_price from bids where bid_id=%s", (id))
                data = crs.fetchone()

                if data[0] > float(new_price):
                        return render_template('message.html', message = "You need␣
↪to bid a higher price from current one!")

                crs.execute("update bids set current_price=%s, current_holder=%s␣
↪where bid_id=%s",(new_price, session['user_id'], id))
                conn.commit()

        return render_template('message.html', message = "You bid successfully␣
↪applied!")
```

Delete_bid

Has one parameter <id>, removes the bidding from database.

```
@bidding_app.route('/delete_bid/<id>')
def delete_bid(id):
        with psycopg2.connect(current_app.config['dsn']) as conn:
                crs=conn.cursor()
                crs.execute("delete from bids where bid_id=%s",(id))
                conn.commit()
        return render_template('message.html', message = "Your bid is successfully␣
↪removed!")
```

3.4.5 Premadelayout

Database Design

```
CREATE TABLE IF NOT EXISTS premadelayouts(
        layout_id serial primary key,
        name text,
        detail text,
        font text,
        font_size int,
        bg_image_url text
);
```

Premadelayout table includes a serial primary key, 4 text columns for name, detail, font, and background image url. Font-size is stored in int format. This table is prepared to be more flexible for the user at session level so there is no foreign keys in it.

Controller Code

Premadelayout page includes 4 methods from its app.

- *Add_new_layout*
- *Layout_delete*
- *Layout_update*
- *Layout_change*

Add_new_layout

When a user wants to add a new bid and fills the form following method is called in server. It requires to be a 'POST' method.

Form variables are taken, then inserted into the database. After that user is shown an operation message.

```
@layout_app.route('/add_new_layout/', methods = ['POST'])
def add_new_layout():
        name = request.form['name']
        details = request.form['detail']
        font = request.form['font_name']
        font_size = request.form['font_size']
        bg_image = request.form['image_url']

        with psycopg2.connect(current_app.config['dsn']) as conn:
                        crs=conn.cursor()
                        crs.execute("insert into premadelayouts (name, detail,␣
↪font, font_size, bg_image_url) values (%s, %s, %s, %s, %s)", (name, details,␣
↪font, font_size, bg_image))
                        conn.commit()

        return render_template('message.html', message = "Layout Successfully␣
↪Added!")
```

Layout_delete

User sends the id of the layout they wish to delete to this method, and the respective layout is deleted from database.

```
@layout_app.route('/layout_delete/<id>')
def layout_delete(id):
        with psycopg2.connect(current_app.config['dsn']) as conn:
```

```
                        crs=conn.cursor()
                        crs.execute("delete from premadelayouts where layout_id =
↪%s", (id))
                        data = conn.commit()

        return render_template('message.html', message = "Layout deleted..")
```

Layout_update

Layout update works similarly to layout insertion, the variables are taken from the user as a form and those values are processed in the method with updating the layout via its id.

```
@layout_app.route('/layout_update/<id>', methods = ['POST'])
def layout_update(id):
        name = request.form['name']
        details = request.form['detail']
        font = request.form['font_name']
        font_size = request.form['font_size']
        bg_image = request.form['image_url']

        with psycopg2.connect(current_app.config['dsn']) as conn:
                        crs=conn.cursor()
                        crs.execute("update premadelayouts set name=%s, detail=%s,␣
↪font=%s, font_size=%s, bg_image_url=%s where layout_id=%s",(name, details, font,␣
↪font_size, bg_image, id))
                        conn.commit()

        return render_template('message.html', message = "Layout updated..")
```

Layout_change

Rather then making a change in database, layout change takes the id of the user selected layout, pulls it from the database and inserts the information to session of the user.

```
@layout_app.route('/layout_change/', methods = ['POST'])
def layout_change():
        lay_id = request.form['layout']
        with psycopg2.connect(current_app.config['dsn']) as conn:
                crs=conn.cursor()
                crs.execute("select * from premadelayouts where layout_id = %s",␣
↪(lay_id))
                data = crs.fetchone()
                session['bimg'] = data[5]
                session['font'] = data[3]
                session['font-size'] = data[4]
                conn.commit()
        return render_template('message.html', message = "Layout Changed.")
```

3.5 Parts Implemented by Ömer Faruk İNCİ

Sending message to a specific user and direct message is implemented by me.

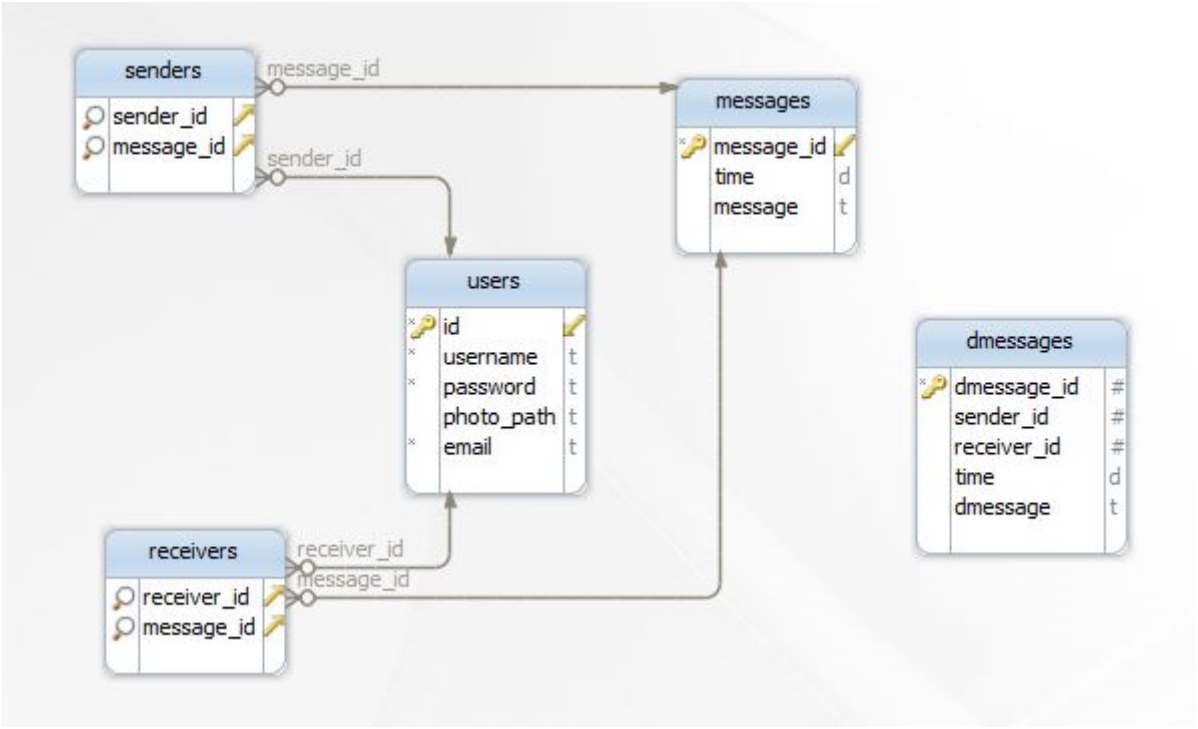


Fig. 3.3: ENTITY-RELATIONSHIP DIAGRAM OF MY PART

3.5.1 General Database Design

3.5.2 Sending message to a specific user

Database Design

```
CREATE TABLE IF NOT EXISTS messages (
 message_id serial primary key,
 time date,
 message text
);

CREATE TABLE IF NOT EXISTS senders (
    sender_id int REFERENCES users (ID) ON DELETE CASCADE,
    message_id int REFERENCES messages (message_id) ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS receivers (
    receiver_id int REFERENCES users (ID) ON DELETE CASCADE,
    message_id int REFERENCES messages (message_id) ON DELETE CASCADE
);
```

Message table has a serial primary key called message_id. Senders and receivers tables refenrece messages and users tables.

Controller Code

```
import os
import psycopg2
from flask import Flask
from flask import render_template, request
```

```
from flask import Blueprint, current_app

gmessage_app = Blueprint('gmessage_app', __name__)
```

Blueprint interface is used for this project. Blueprint make development easier by having the ability of separating the project into varied files.

```
@gmessage_app.route('/gmessage',methods=["POST"])
def gmessage():
    gmessage = request.form['gmessage']
    sender = request.form['senders']
    receiver = request.form['receivers']
    with psycopg2.connect(current_app.config['dsn']) as conn:
            crs=conn.cursor()
            crs.execute("insert into messages (time, message) values (now(),
↪%s)RETURNING message_id", (gmessage,))
            m_id = crs.fetchone()[0]
            crs.execute("select ID from users where username=%s", (sender,))
            sndr_id = crs.fetchone()
            crs.execute("insert into senders (sender_id, message_id) values (%s,
↪%s)", (sndr_id,m_id))
            crs.execute("select ID from users where username=%s", (receiver,))
            rcvr_id = crs.fetchone()
            crs.execute("insert into receivers (receiver_id, message_id) values (
↪%s, %s)", (rcvr_id,m_id))
            data = conn.commit()

    return render_template('message.html', message = "Message has been sent.")
```

This function is for sending message to a specific user. After getting the message, sender and receiver info, it adds their information to database.For getting the id of user by giving its name, SQL selection is used.

- A disposable connection to database server is created via 'with' command which gets configuration from main application(current_app) settings.
- Creates a cursor.
- Insert into messages table.
- Executes an SQL insertion.
- Executes an SQL selection.
- Commits the changes and save the result of the operation.

If the function runs properly, rendered template message.html is returned with the information of result of action.

```
@gmessage_app.route('/gmessage_delete/<id>')
def gmessage_delete(id):
    with psycopg2.connect(current_app.config['dsn']) as conn:
            crs=conn.cursor()
            crs.execute("delete from messages where message_id = %s", (id))
            crs.execute("delete from senders where message_id = %s", (id))
            crs.execute("delete from receivers where message_id = %s", (id))
            data = conn.commit()

    return render_template('message.html', message = "Message has been deleted.")
```

This function deletes the message, sender and the reciever information from the messages, receivers and senders table with given message id.

- The function connects to the database.
- Creates a cursor.
- Delete from messages, senders and receivers table.

- Commits the changes to the database.

If the function runs properly, rendered template message.html is returned with the information of result of action.

```
@gmessage_app.route("/gmessage_update/<id>",methods=["POST"])
def gmessage_update(id):
    updated_gmessage = request.form["new_gmessage"]
    with psycopg2.connect(current_app.config["dsn"]) as conn:
        crs = conn.cursor()
        crs.execute('update messages set time=now(),message=%s where message_id=%s
↪',(updated_gmessage,id))
        conn.commit()

    return render_template("message.html",message="Message has been updated.")
```

Uploading a message is enabled by this function with the given message id.

- The function connects to the database.
- Creates a cursor.
- Executes an SQL update with message id.
- Commits the changes to the database.

If the function runs properly, rendered template message.html is returned with the information of result of action.

3.5.3 Direct Messages

Database Design

```
CREATE TABLE IF NOT EXISTS directmessages (
    dmessage_id serial primary key,
    sender_id int,
    receiver_id int,
    time date,
    dmessage text
);
```

Controller Code

```
@app.route('/dmessage')
def dmessage():
    if session.get('logged_in')== None:
        return redirect(url_for("loginpage"))
    with psycopg2.connect(app.config['dsn']) as conn:
        crs=conn.cursor()
        crs.execute("select * from directmessages order by time desc")
        dmessages = crs.fetchall()

    now =datetime.datetime.now()

    return render_template('dmessage.html', current_time=now.ctime(), dmessage_app␣
↪= dmessage_app, dmessage_list=dmessages)
```

This function routes the dmessage page and it also sends some data to the dmessage page.

- Function looks at the seesion to see if the user is logged in or not.
- If user is logged in, it connects to the database.
- Creates a cursor.

- Executes an SQL select query to list the followed users.

Finally, it redirects to the dmessage.html.

```
@dmessage_app.route('/dmessage',methods=["POST"])
def dmessage():
    dmessage = request.form['dmessage']
    with psycopg2.connect(current_app.config['dsn']) as conn:
            crs=conn.cursor()
            crs.execute("insert into directmessages (sender_id, receiver_id, time,␣
↪dmessage) values (%s, %s, now(), %s)", (1, 2, dmessage))
            data = conn.commit()

    return render_template('message.html', message = "Message has been sent.")
```

This function adds a new message with the information of sender id, receiver id and the time.

- It requests message info from the dmessage.html
- Then connects to the database and also generates a cursor.
- Inserts the message to the directmessages table.
- Commits the changes to the database.

If the function runs properly, rendered template message.html is returned with the information of result of action.

```
@dmessage_app.route('/dmessage_delete/<id>')
def dmessage_delete(id):
    with psycopg2.connect(current_app.config['dsn']) as conn:
            crs=conn.cursor()
            crs.execute("delete from directmessages where dmessage_id = %s", (id))
            data = conn.commit()

    return render_template('message.html', message = "Message has been deleted.")
```

This function deletes the existing message with given message id.

- It connects to the database and also generates a cursor.
- Deletes the message from directmessages table.
- Commits the changes to the database.

If the function runs properly, rendered template message.html is returned with the information of result of action.

```
@dmessage_app.route("/dmessage_update/<id>",methods=["POST"])
def dmessage_update(id):
    updated_dmessage = request.form["new_dmessage"]
    with psycopg2.connect(current_app.config["dsn"]) as conn:
        crs = conn.cursor()
        crs.execute('update directmessages set time=now(),dmessage=%s where␣
↪dmessage_id=%s ',(updated_dmessage,id))
        conn.commit()

 return render_template("message.html",message="Message has been updated.")
```

This function updates the existing message with given message id.

- It connects to the database and also generates a cursor.
- Updates the message from directmessages table.
- Commits the changes to the database.

If the function runs properly, rendered template message.html is returned with the information of result of action.