

**Online Communication Platform
Software Requirements Specification
For Web Application**

Version 2.0

CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

Revision History

Date	Version	Description	Author
03/11/2020	1.0	First Version of CodeHub	Rehman Arshad Jorge Quiroz
05/1/2020	2.0	Design Report	Rehman Arshad Jorge Quiroz

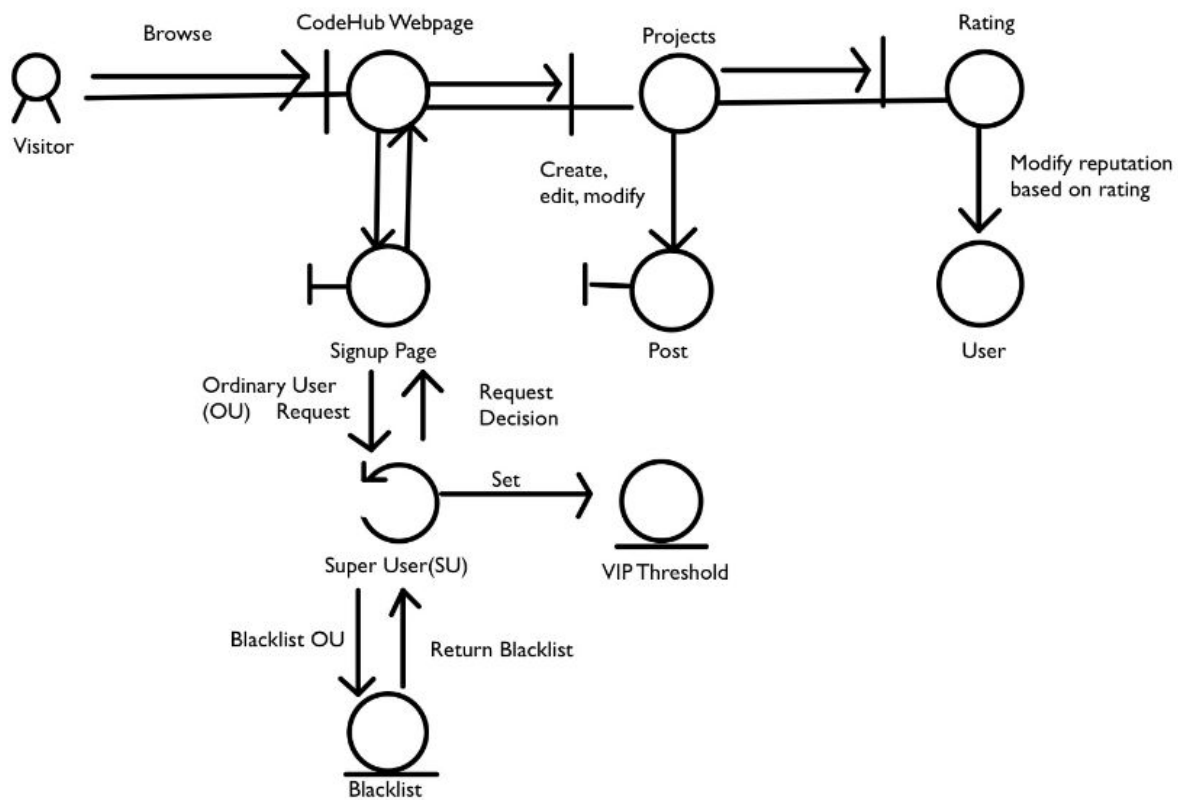
CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

Design Report

1. Introduction

This document presents the design details and functionalities of our online communication platform, **CodeHub**. The main purpose of this report is to provide further understanding of our system.

The collaboration class diagram below describes an overall picture of our online system in how the objects interact with the system and the main functionalities under typical use.



CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

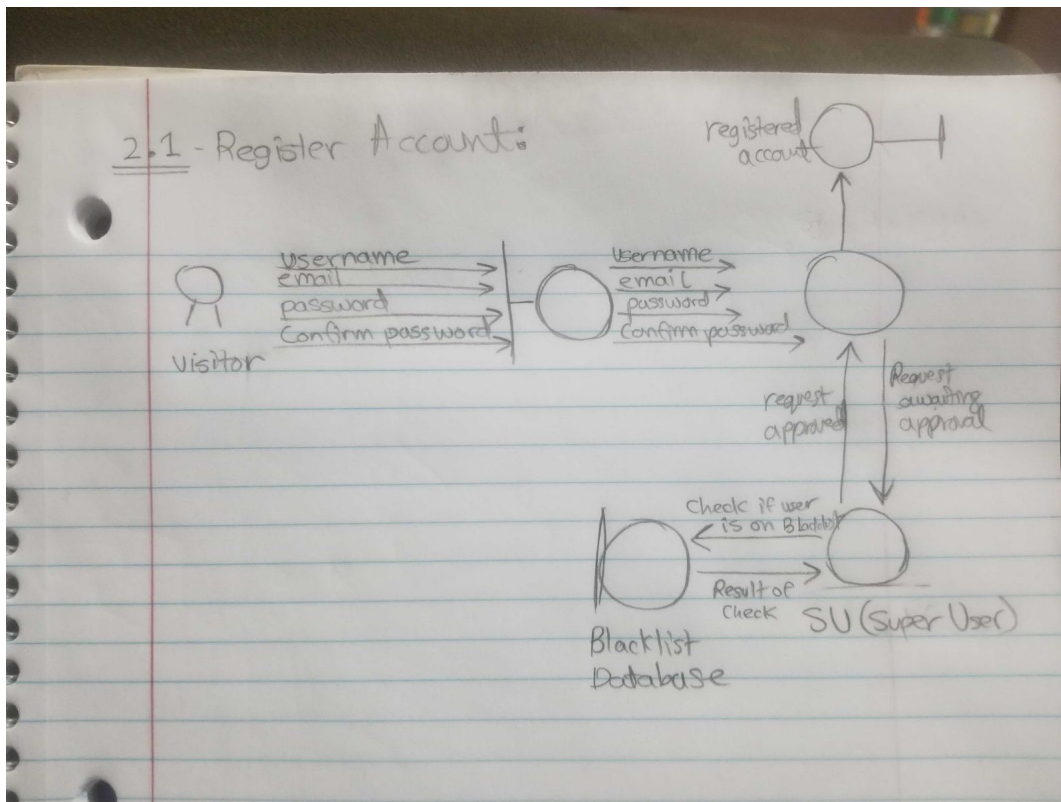
2. Use Case Analysis

For this section, we delve into more details for each of the main functionalities that were discussed in the specification report. We provide collaboration class diagrams to show the relationship between classes and objects. We use petri-nets to better understand the processes related to each of the use case diagrams from the specification report.

2.1 - Register Account:

Normal Scenario: In the home page, a visitor has the option to sign up for an account if they are not already registered. The visitor must provide their desired username, email, password and password confirmation.

Exceptional Scenario: The information provided by the visitor must go through a check. If the information given is present within the system's blacklist, the visitor will not be allowed to register. A message will appear informing them that they have been denied. Unregistered users are only visitors.

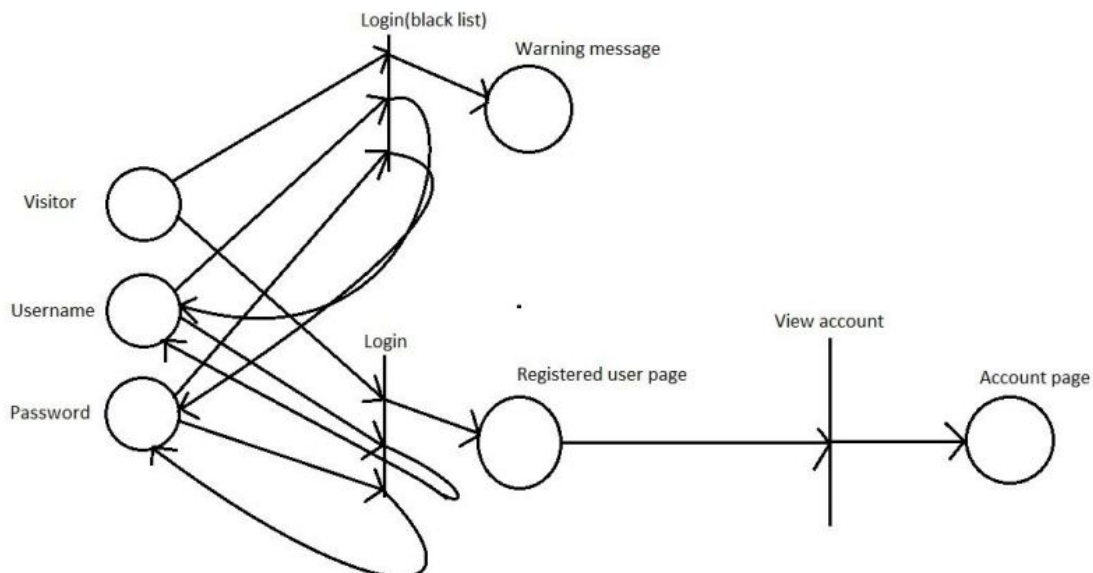
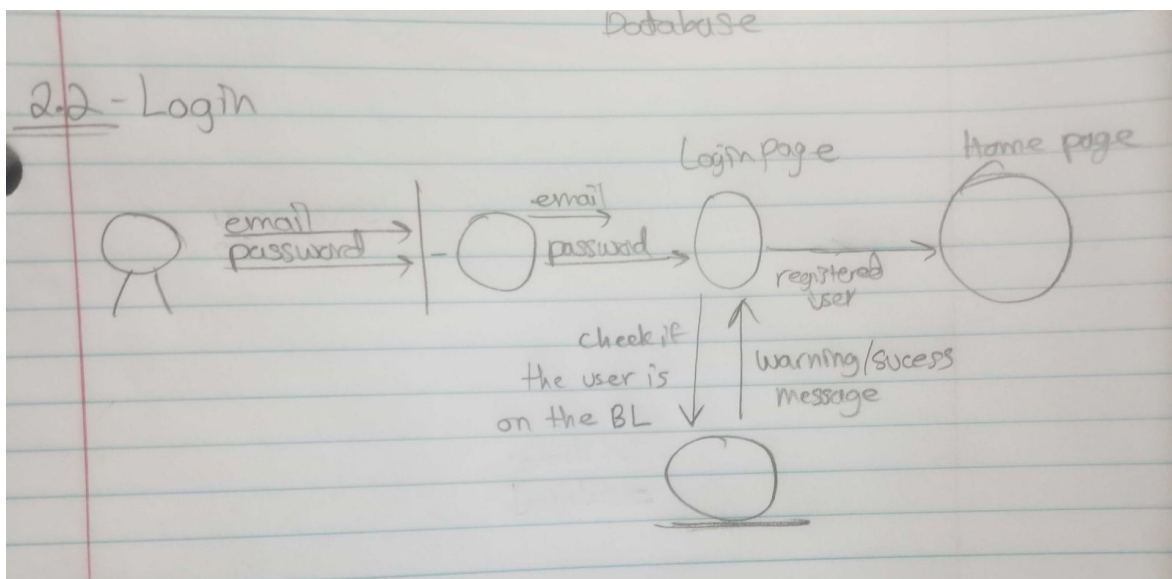


CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

2.2 - Login:

Normal Scenario: The visitor can sign in. The sign in page will prompt the user to enter their username and password. If the login is successful, he/she will gain access to their account page and will be able to engage with projects on the site. At this point, the visitor is now an ordinary user (OU) within the system.

Exceptional Scenario: If a user has a bad reputation, a super-user may place him/her on the blacklist. If this is the case, after the user inputs their login info, they will be met with a warning message and be denied access to login. If a visitor enters the wrong username and password, he/she will be redirected to the login page to try again.

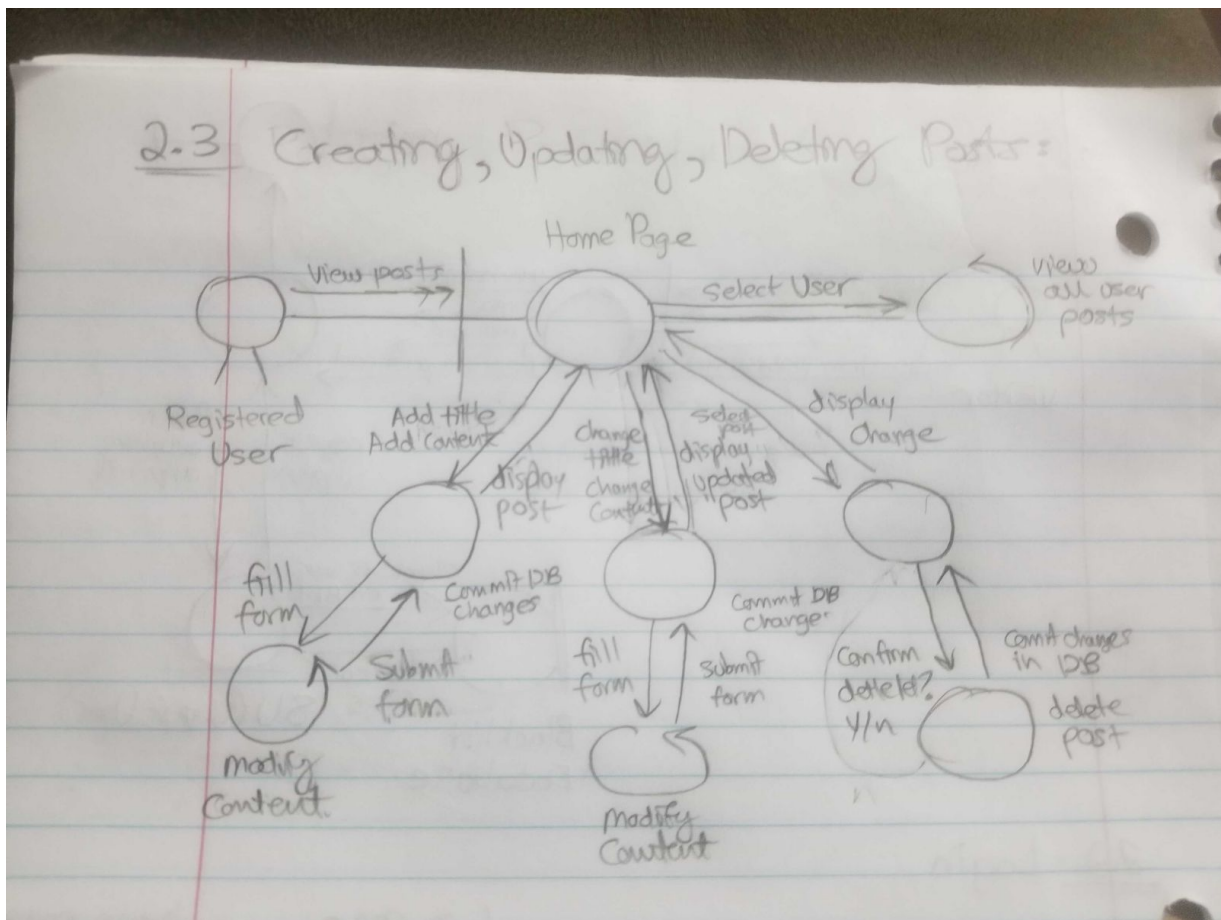


CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

2.3 - Projects and Posts:

Normal Scenario: Ordinary Users (OUs) can view current trending and available projects on the home page. The OU can commit to a project. Once the OU is a part of one or more projects, the projects will be displayed within their dashboard and they will have the ability to edit or modify projects they are a part of.

Exceptional Scenario: OUs may have their profile listed on the home page as current trending 'top' users, based on their reputation score. VIP Users (OUs with above a 30 reputation score) will also be displayed.



CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

2.4 - Reputation System

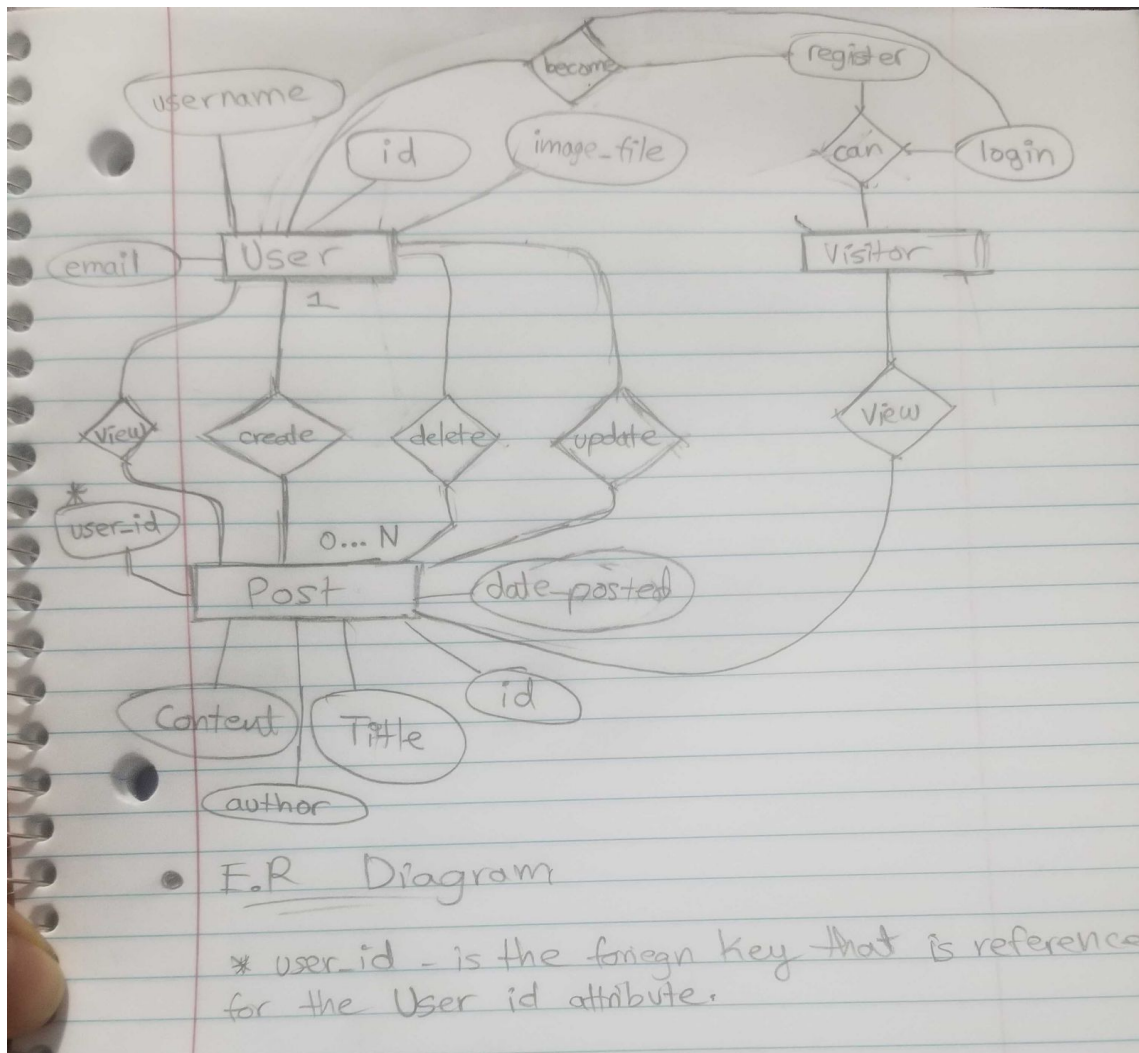
Normal Scenario: Normally, the OUs within a project will be able to rate one another.

Exceptional Scenario: Group members within a project can vote to issue a warning or praise to a particular member. This vote must be unanimous. A member that receives 3 warnings is immediately removed from the group/project and automatically gets a 5-point deduction to their reputation score. The group can also vote to kick out a member directly, the member will be removed from the group and receives a 10-point reputation score deduction. Once an OU's reputation score goes below 0, they are immediately removed from the system and placed on the blacklist. Once a project is complete or when the group members decide for any reason to close the group, an exit evaluation between members is conducted. Each member will receive the median reputation score given by all other members. Each member can decide if s/he is willing to put the other member to her/his white-box or black-box afterwards. After group closure, a SU will assign a VIP to evaluate the group and determine a reputation score for the entire group to be added/deducted for all members involved. The system will keep a ranking list of finished groups to be showcased. An OU whose reputation score is higher than 30 will be promoted to VIP; and a VIP whose score is lower than 25 will be demoted. All VIPs can vote one VIP as the democratic SU.

CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

3. ER Diagram for the entire system

The figure below is an ER diagram describing interrelated things of interest in a specific domain of knowledge. A basic ER model is composed of entity types and specifies relations that can exist between entities.



CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

4. Detailed design:

File Location: `app/main/routes.py*`

4.1. Home Page:

```
@main.route('/')
@main.route('/home')
def home():
    page = request.args.get('page', 1, type=int)
    posts = Post.query.order_by(Post.date_posted.desc()).paginate(page=page, per_page=5)
    return render_template('home.html', posts=posts)
```

Flask supports route duplication, so the pseudo code above creates 2 routes `'/'`, and `'home'`. These routes have a function called `home()`. The `home()` function accepts no parameters and renders the `'home.html'` page inside of the **templates directory**. In order to display all the posts from the SQLAlchemy database I used a local variable `Post`, where I stored each incoming post, in a descending query call. This was done so that the last created post would show up first. Meaning that the newer posts would appear on the top of the page, and the older posts would eventually sink to the bottom. The page variable is just for pagination so older posts eventually leave the page and go to the next page. The database query also has a `.paginate()` method which allows us to show the top 5 posts on the page. This will be tweaked later to 3 in order to meet the specifications. (For GUI refer to 5.1 - 5.2)

4.2. About Page:

```
@main.route('/about')
def about():
    return render_template('about.html', title='About us')
```

The `'/about'` route has the `about()` function which accepts no parameters and renders `'about.html'` from the **templates directory**.

File Location: `app/users/routes.py*`

4.3. Register Page:

```
@users.route('/register', methods=['GET', 'POST'])
def register():
    if current_user.is_authenticated:
        return redirect(url_for('main.home'))
    form = RegistrationForm()
    if form.validate_on_submit():
        hashed_password = bcrypt.generate_password_hash(form.password.data).decode('utf-8')
        user = User(username=form.username.data, email=form.email.data, password=hashed_password)
```

CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

```

db.session.add(user)
db.session.commit()
flash( f'Welcome {form.username.data}! Your account has been created!, 'success')
return redirect( url_for('users.login') )
return render_template('register.html', title='Register', form=form)

```

The **‘/register’** route has a **register()** function that accepts no parameters and renders the **‘register.html’** page from the **templates directory**. If the current user is authenticated, they are redirected to the home page. Keep in mind once a person is logged in I remove the register route, so it's no longer available in the navbar on the front-end. Over here we store the Registration form into the form variable. Registration form is made using flask-wtf, which I will elaborate more later on. If the form is valid on submission, **(i.e there is no blank/null, or invalid/illegal inputs)**, then the user's password is hashed using the **bcrypt** encryption algorithm, and that hashed key is then stored into the hashed_password local variable. Then we create a new user which is imported from our DB schema model. A user has a username, an email, and a password which is all provided from the Registration form. From here we simply add this newly created user into the Database, and commit our session changes. Then I choose to display a flash message to let the user know that they can log in now! **Success** is just a bootstrap class; it has a greenish hue. The final line redirects the user to the login page. Two important things to note, **url_for()** is flask's default way to utilize routes, and the flash message I displayed utilizes **‘F Strings’** which is only available in Python 3.6 and above! **(For GUI refer to 5.7)**

4.4. Login Page:

```

@users.route('/login', methods=['GET', 'POST'])
def login():
    if current_user.is_authenticated:
        return redirect(url_for('main.home'))
    form = LoginForm()
    if form.validate_on_submit():
        user = User.query.filter_by(email=form.email.data).first()
        if user and bcrypt.check_password_hash(user.password, form.password.data):
            login_user(user, remember=form.remember.data)
            next_page = request.args.get('next')
            return redirect(next_page) if next_page else redirect(url_for('main.home'))
        else:
            flash('Login Unsuccessful. Please check username and password', 'danger')
    return render_template('login.html', title='Login', form=form)

```

The **‘/login’** route has the **login()** function which accepts no parameters and renders the **‘login.html’** page from the **templates directory**. If the current user is already authenticated, then redirect them back to the home

CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

page. This is standard exception handling as the same user cannot log in more than once. Note that on the front-end once a user log's in. I removed the login route from the navbar on the front-end. This is an extra, but necessary level of exception handling on the back-end. From here we store the LoginForm into a local variable called form. If the form is valid on submission, we store the user's email address. However for the password there is an entirely different procedure, we essentially take the user's password that was submitted in the form as plaintext, and we re-hash it using bcrypt encryption algorithm again, and compare the hashed output of the users input, with the hashed_key stored in our Database. If the strings match the user is validated, and logged in redirected to the home page. However, if the strings don't match an error message will be displayed. **Danger** is just a bootstrap class; it has a reddish/pinkish hue. **(For GUI refer to 5.8)**

Why does the password have an entirely different procedure?

I implemented our password procedure using a double-blind comparison. This way even in the event of any security breach, if the database is leaked online, none of the passwords will be leaked. This is because **bcrypt** is a **one-way encryption algorithm**, there is no decryption algorithm for it. So by re-encrypting the user's input on the login page if the password is correct the hashed_key should be the same as the one stored in the Database. There are some other things I would like to implement here such as **SHA-256**, because **bcrypt** is a slow algorithm (which is good, because it prevents brute force attacks), however because it's a slow and resource expensive algorithm and a hacker can use this fact to overload and crash a server with too many requests in the form of a Distributed Denial of Service (**DDoS**) attack. **SHA-256** would limit the possible input to a max of 256 characters preventing any overload. Another thing is that **bcrypt** is fine if the hacker is armed with a **CPU**, and a **GPU**, but if the hacker is armed with an **FPGA**, then we are in a lot of trouble, because the hacker can potentially brute force a password, or atleast get more guesses per second than a **CPU/GPU** accelerated penetration attempt. For such a scenario I was thinking of using **PBKDF2** along with **SHA-256**, and a ton of salts to prevent any **rainbow table** attacks. Another goal is to implement 2 Factor Authentication (**2FA**) probably by using FireBase, but more research is needed here. I will also need to do more research into **OWASP** Standards and Guidelines on User Authentication.

4.5. Log out

```
@users.route('/logout')
def logout():
    logout_user()
    return redirect(url_for('main.home'))
```

The **‘/logout’** route has the **logout()** function which accepts no parameters and does not render any page. Once the link is clicked on the top right side of the navbar on the front-end, this route is launched, and the **logout()** function is executed. All this function does is call **logout_user()** which is a default logout method inside the package **flask_login**. The **logout_user** function de-validates the currently authenticated user, deletes

CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

their cookies and ends the current session. Flask has many useful modules that help with session management, but flask_login is one of the most important ones. The last line redirects the user back to the home page, but as a visitor (**not a registered user**). (For GUI refer to 5.1)

4.6. Account Page

```
@users.route('/account', methods=['GET', 'POST'])
@login_required
def account():
    form = UpdateAccountForm()
    if form.validate_on_submit():
        if form.picture.data:
            picture_file = save_picture(form.picture.data)
            current_user.image_file = picture_file

        current_user.username = form.username.data
        current_user.email = form.email.data
        db.session.commit()
        flash('Your account has been updated!', 'success')
        return redirect(url_for('users.account'))
    elif request.method == 'GET':
        form.username.data = current_user.username
        form.email.data = current_user.email
    image_file = url_for('static', filename='profile_images/' + current_user.image_file)
    return render_template('account.html', title='Account', image_file=image_file, form=form)
```

The **‘/account’** route has the **account()** function which accepts no parameters and renders the **‘account.html’** page from the **templates directory**. This route **requires login**, meaning no visitor has access to this. As with the Registration and login page, we have another form made using flask-wtf, however the only difference here is unlike those pages these forms are not empty by default, they are populated with the users information. If the form is valid on submission (**i.e there is no blank/null, or invalid/illegal inputs**) then using the forms input data the user’s account information is updated. Basically we overwrite the old data with the new data being provided in the forms. This way a user can change his/her email address, username, and profile photo. Then we commit our session’s changes into the database. Note for clarity we are not creating a new user in our database, we are modifying an existing user in our database. Once the changes have been committed a flash message appears to notify the user that the changes have been saved. **Success** is just a bootstrap class, it has a greenish hue. On the second last line the **image_file** variable is storing the user’s profile photo which is then passed on to the **render_template** as a parameter to display that specific user’s profile photo on the account page, recall each user has a unique id. So when storing these images in the **static/profile_images directory** I needed to make sure each image had a unique name otherwise the program would either reach an

CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

unhandled exception and crash when saving a profile image or overwrite an existing user's image of the same name. To solve this every image stored into our database would be renamed and each file would be given a cryptographic 32 bit hashed string and then I concatenated that string with the users id, which in my Database Schema is required to be unique. By doing this even if a user decides to change their profile photo later on, a new cryptographic hash string is generated for the image uploaded, so even if it shares the same unique user id, it will still have a different name and not overwrite anything. **(For the GUI refer to 5.4)**

4.7. User Post Page:

```
@users.route('/user/<string:username>')
def user_posts(username):
    page = request.args.get('page', 1, type=int)
    user = User.query.filter_by(username=username).first_or_404()

    posts = Post.query.filter_by(author=user)\
        .order_by(Post.date_posted.desc())\
        .paginate(page=page, per_page=5)
    return render_template('user_posts.html', posts=posts, user=user)
```

The `‘/user/<string:username>’` route contains the `user_posts()` function which accepts the `username` parameter and renders `‘user_posts.html’` from the `templates directory`. Each username is unique and this function then creates a page and filters the database to find all posts made from the username parameter we just passed in. Either we find the first username in the database and display all posts that user has made, or we give a **404 error** meaning no such user exists. The posts are stored in the posts variable which we obtain by making a database query and filter all posts in a descending order (**meaning the newer posts will show up on the top**) that match the username parameter we passed in earlier. The `.paginate()` method shows 5 posts per page, but this will be tweaked later on to 3 in order to meet the specifications. **(For GUI refer to 5.6)**

4.8.1 Password Reset Page (part 1):

```
@users.route("/reset_password", methods=['GET', 'POST'])
def reset_request():
    if current_user.is_authenticated:
        return redirect(url_for('main.home'))
    form = RequestResetForm()
    if form.validate_on_submit():
        user = User.query.filter_by(email=form.email.data).first()
```

CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

```

    send_reset_email(user)
    flash('An email has been sent with instructions to reset your password.', 'info')
    return redirect(url_for('users.login'))
return render_template('reset_request.html', title='Reset Password', form=form)

```

The `/reset_password` route has the `reset_request()` function which renders the `request_request.html` from the templates directory. If the current user is already authenticated they are redirected to the home page. If they are not authenticated then the user can fill out the password reset request form, which when validated on submission (**i.e no invalid emails are inserted**) finds the user in the database using the provided email address, and then calls the `send_reset_email()` function, passing in the user we found in our database as a parameter. After that we display a message to indicate to the user that an email has been sent with a password reset link. Info is just a bootstrap class, it has a light bluish hue. The next line redirects the user to the login page where they can use their new password to login again. **(For GUI refer to 5.11 - 5.15)**

4.8.2 Password Reset Page (part 2):

```

@users.route("/reset_password/<token>", methods=['GET', 'POST'])
def reset_token(token):
    if current_user.is_authenticated:
        return redirect(url_for('main.home'))
    user = User.verify_reset_token(token)
    if user is None:
        flash('That is an invalid or expired token', 'warning')
        return redirect(url_for('users.reset_request'))
    form = ResetPasswordForm()
    if form.validate_on_submit():
        hashed_password = bcrypt.generate_password_hash(form.password.data).decode('utf-8')
        user.password = hashed_password
        db.session.commit()
        flash('Your password has been updated! You are now able to log in', 'success')
        return redirect(url_for('users.login'))
    return render_template('reset_token.html', title='Reset Password', form=form)

```

Recall Flask allows route duplication, so using the same route as **4.8.1** we can now use the `reset_token()` function, which generates a unique token for a unique page timed URL, that creates a timed reset page. If the token is valid, when the user clicks this link in their email they are greeted with the reset page. Over here they can enter a new password, which is yet again hashed using **bcrypt** using the same double blind comparison procedure described above. We commit our session changes in our database and flash a success message, while redirecting our user to the login page. If the token is not valid or expired then the user is redirected back

CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

to the password reset page in **4.8.1**, with a flash message informing the user that the token is either expired or invalid. Recall success is just a bootstrap class; it has a greenish hue. Warning is also a bootstrap class, it has a yellowish hue. **(For GUI refer to 5.11 - 5.15)**

File Location: app/posts/routes.py*

4.9 Create Post Page:

```
@posts.route('/post/new', methods=['GET', 'POST'])
@login_required
def new_post():
    form = PostForm()
    if form.validate_on_submit():
        post = Post(title=form.title.data, content=form.content.data, author=current_user)
        db.session.add(post)
        db.session.commit()
        flash('Your post has been created!', 'success')
        return redirect(url_for('main.home'))
    return render_template('create_post.html', title='New Post', form=form, legend='New Post')
```

The `‘/post/new’` route has the `new_post()` function which accepts no parameters, and renders `‘create_post.html’` from the templates directory. Contrary to general intuition `‘create_post.html’` is not an entire web page, it is a template or component/part of the web page which can be reused many times similar to posts. This is because I utilized **jinja2** syntax when creating my pages, more importantly though in flask jinja2 syntax supports template inheritance. Template inheritance allowed me to create a base template which each component page could inherit from, this saved me from a lot of redundant coding. This route and therefore the subsequent function can only be accessed if the user is logged in, meaning a visitor can not use this route. A visitor would be redirected to the login page even if they tried to access this route. So we take the user input from the form, store it into the post local variable, add that post into our database, commit our session changes, and display a success message as we redirect the user to the home page, where the users post should appear on the top. **(For GUI refer to 5.3)**

4.10. Post Page:

```
@posts.route('/post/<int:post_id>')
def post(post_id):
    post = Post.query.get_or_404(post_id)
    return render_template('post.html', title=post.title, post=post)
```

The `‘/post/<int:post_id>’` has the `post()` function, which accepts the `post_id` as a parameter. `Post_id` is an attribute for each post defined in our Database Schema. From there all we do is find the post, or return a **404**

CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

error if no such post exists; And render the ‘**post.html**’ page, which if you recall is simply a template inheritance based component in flask, not an entirely custom page. If the user is not logged in this page appears when they click on a post.

4.11. Post Update Page:

```
@posts.route('/post/<int:post_id>/update', methods=['GET', 'POST'])
@login_required
def update_post(post_id):
    post = Post.query.get_or_404(post_id)
    if post.author != current_user:
        abort(403)
    form = PostForm()
    if form.validate_on_submit():
        post.title = form.title.data
        post.content = form.content.data
        db.session.commit()
        flash('Your post has been successfully updated!', 'success')
        return redirect(url_for('posts.post', post_id=post.id))
    if request.method == 'GET':
        form.title.data = post.title
        form.content.data = post.content
    return render_template('create_post.html', title='Update Post', form=form, Legend='Update Post')
```

The ‘**/post/<int:post_id>/update**’ route has the **update_post()** function which accepts the **post_id** as a parameter, and render’s the same ‘**create_post.html**’ inherited template. First we find the post from the database using the **post_id** or return a **404 error** if no such post with that id exists. If this post does exist then we load a **PostForm()** and store it into local variable form. However just like the update account page these forms fields are already filled in with the same posts text, which we obtain using the **GET** request; which the user can now modify. Once the modification is complete, and the form is valid on submission (**i.e no illegal characters, etc**) the post title and content is overwritten with the new data we obtain from the forms. Then we commit our session changes in the database, followed by flashing a success message, and redirecting the user to the Post Update page where the modified post will appear. Visitors can not access this page, they will be redirected to the login page. Another security check is that no other user can modify another user’s post, this redirects to a **403: Forbidden error** code. Only the author of the original post can modify his/her own post. This page will only appear if the user is authenticated/logged in. (**For GUI refer to 5.5**)

CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

4.12. Delete Post

```
@posts.route('/post/<int:post_id>/delete', methods=['POST'])
@login_required
def delete_post(post_id):
    post = Post.query.get_or_404(post_id)
    if post.author != current_user:
        abort(403)
    db.session.delete(post)
    db.session.commit()
    flash('Your post has been successfully deleted!', 'success')
    return redirect(url_for('main.home'))
```

The `‘/post/<int:post_id>/delete’` route has the `delete_post()` function which accepts the `post_id` parameter, but does not render any page/template. We find the post, by its id if it exists otherwise we get a 404 error. To ensure that only the original author of the post can delete his/her own post, and no one else. If the `post.author` does not equal, to the `current_user` we abort and give a **403: Forbidden** error. If the `post.author` does equal to the `current_user` we delete the post from the database, and commit our session changes to the database, and flash a success message as we redirect the user back to the home page where their post is no longer available since it has now been deleted. Note `post.author` is defined in our Database Schema, however `current_user` is a variable created by flask_login for session management. (For GUI refer to 5.9)

File Location: `app/models.py*`

4.13. Database User Schema (Models):

```
class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(20), unique=True, nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)
    image_file = db.Column(db.String(20), nullable=False, default='default.jpg')
    password = db.Column(db.String(60), nullable=False)
    posts = db.relationship('Post', backref='author', lazy=True)
```

The **User** model has an id, which is the **super key**, a **username** which is a string of up to 20 characters, and must be unique, a username can not be null. **Email** is a string of up to 120 characters, it must be unique and cannot be null. **Image_file** is a string of upto 20 characters, it cannot be null, and has a default value of `‘default.jpg’` which is the default profile photo in the application. **Password** is a string of up to 60 characters, and it cannot be null. **Posts** sets the relationship between the Post model, and the User model, which for the

CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

record is a **one-to-many relationship**, because a user can create many posts, but each post can only have one author.

4.14. Database Post Schema (Models):

```
class Post(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(100), nullable=False)
    date_posted = db.Column(db.DateTime, nullable=False, default=datetime.utcnow)
    content = db.Column(db.Text, nullable=False)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)

    def __repr__(self):
        return f"Post('{self.title}', '{self.date_posted}')
```

The **Post** model has an **id**, which is the **super key**, a **title** consisting of a string of up to 100 characters and it cannot be null, **date_posted** which is automatically filled out using Python's `datetime` library. **Content** is stored into the db as text, but has no character limit whatsoever, but it cannot be null. **User_id** is just a foreign key into the user table, because each post needs a way to find its author. I may modify this model to add a character limit in each post similar to how twitter does it.

CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

File Location: `app/errors/handlers.py*`

4.15 Graceful Error Handling:

```
@errors.app_errorhandler(403)
def error_403(error):
    return render_template('errors/403.html'), 403

@errors.app_errorhandler(404)
def error_404(error):
    return render_template('errors/404.html'), 404

@errors.app_errorhandler(500)
def error_500(error):
    return render_template('errors/500.html'), 500
```

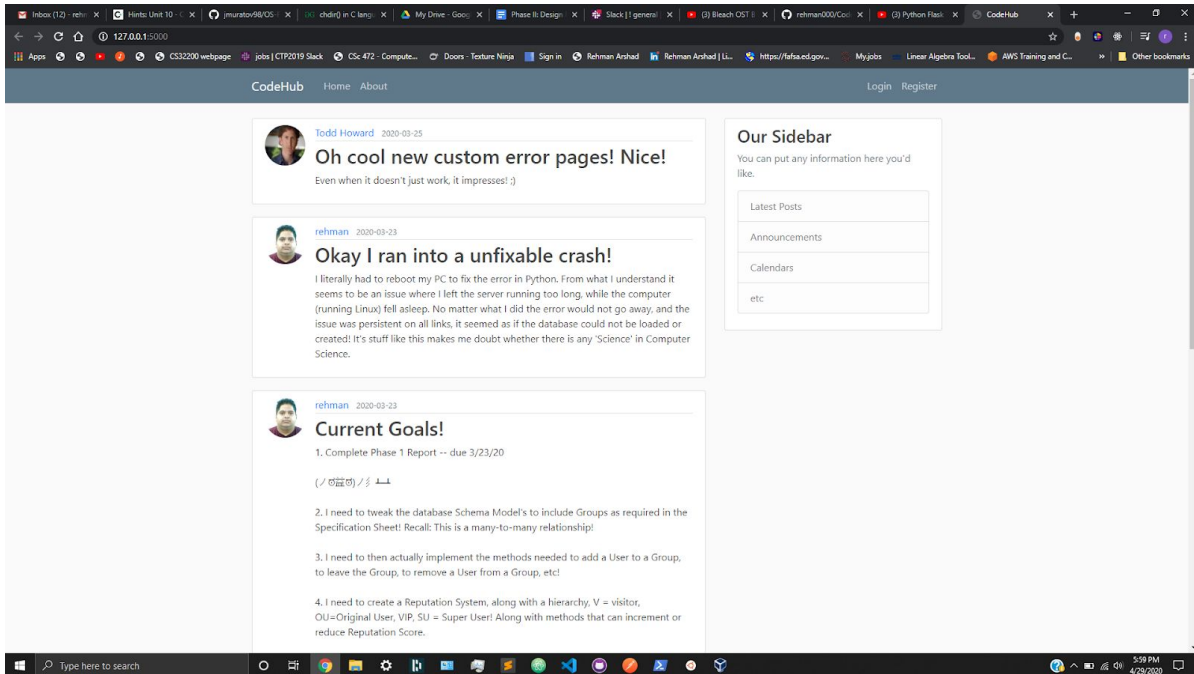
When a 403, 404, or 500 series error occurs the following pages are rendered. This is why the errors directory is in the root app directory. This is because it allows graceful error handling whenever it may occur within the application's lifespan. On some feedback received I might change the current working directory for the errors folder and move it into the templates directory. **(For GUI refer to 5.16 - 5.17)**

CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

5. System screens:

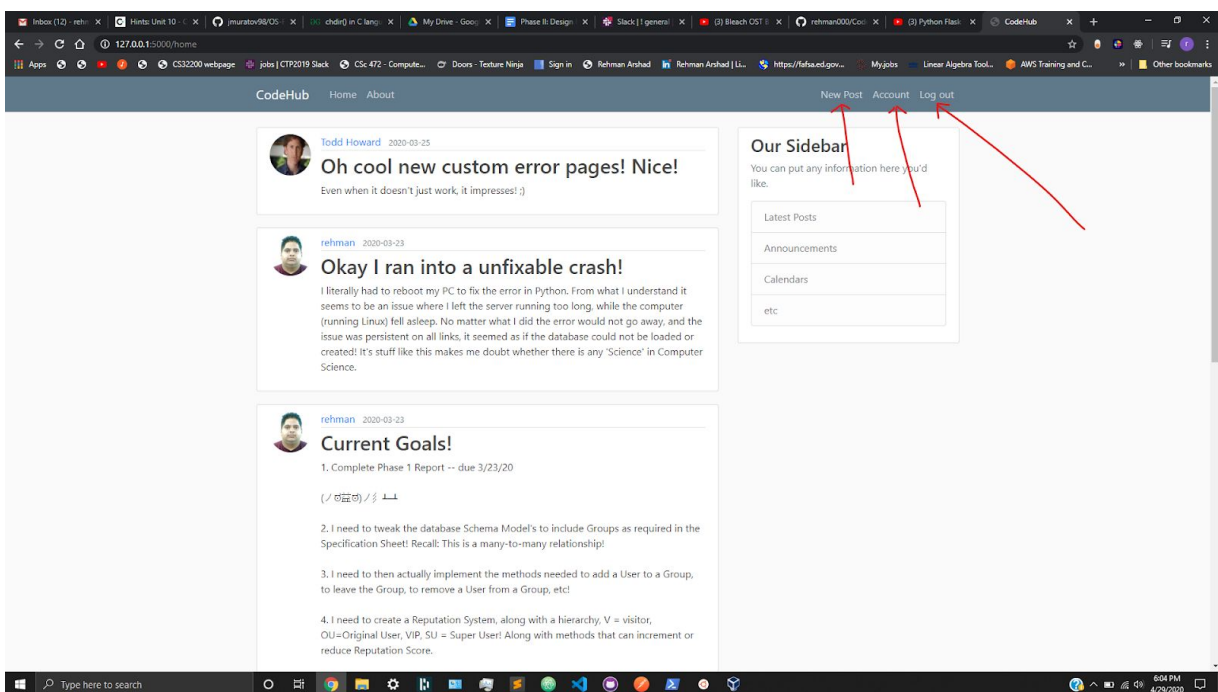
Before we begin I would like to clarify that all of these images are fully functioning and this site is responsive on all screen sizes, for desktop, tablets, and smartphones, and ADA compliant.

5.1. Home Page (Visitor):



On this page a visitor can view all posts, and hit the register button, or login button.

5.2. Home Page (Registered User):



Once logged in a registered user can create posts, view their account, or log out.

CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

5.3. Create Post Page (Registered Users ONLY):

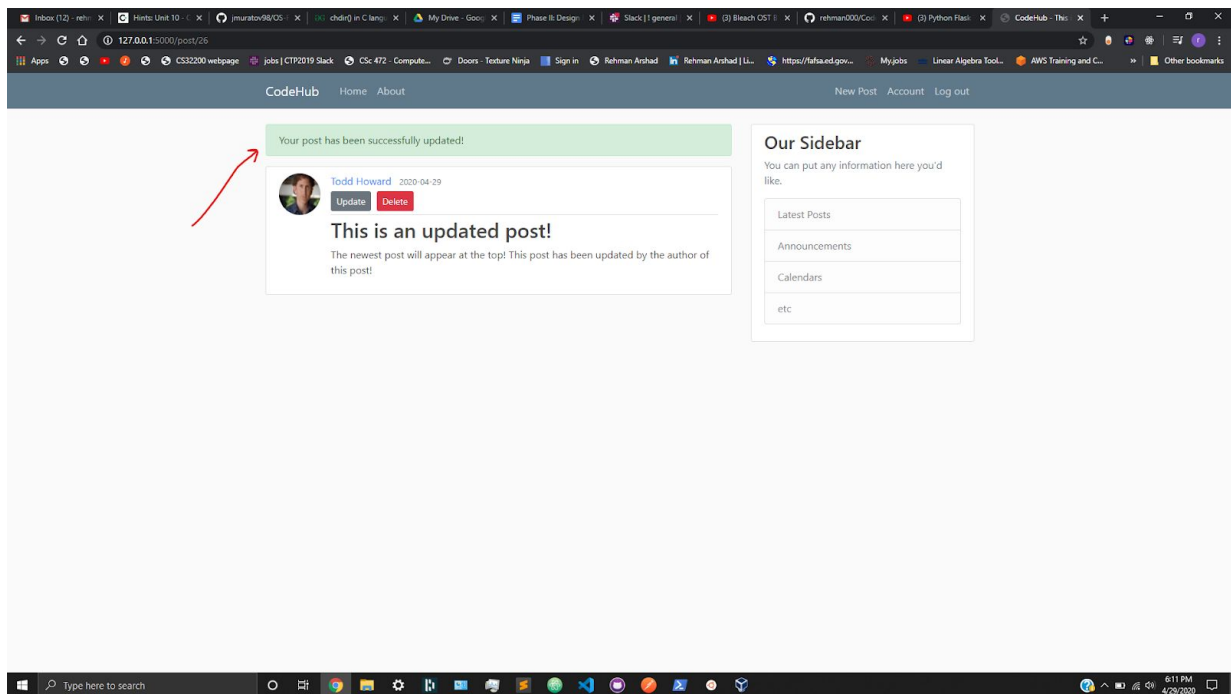
This page let's a registered user create a post.

5.4. Account Page (Registered Users ONLY):

This page let's a registered user change his/her email account, username, and profile photo.

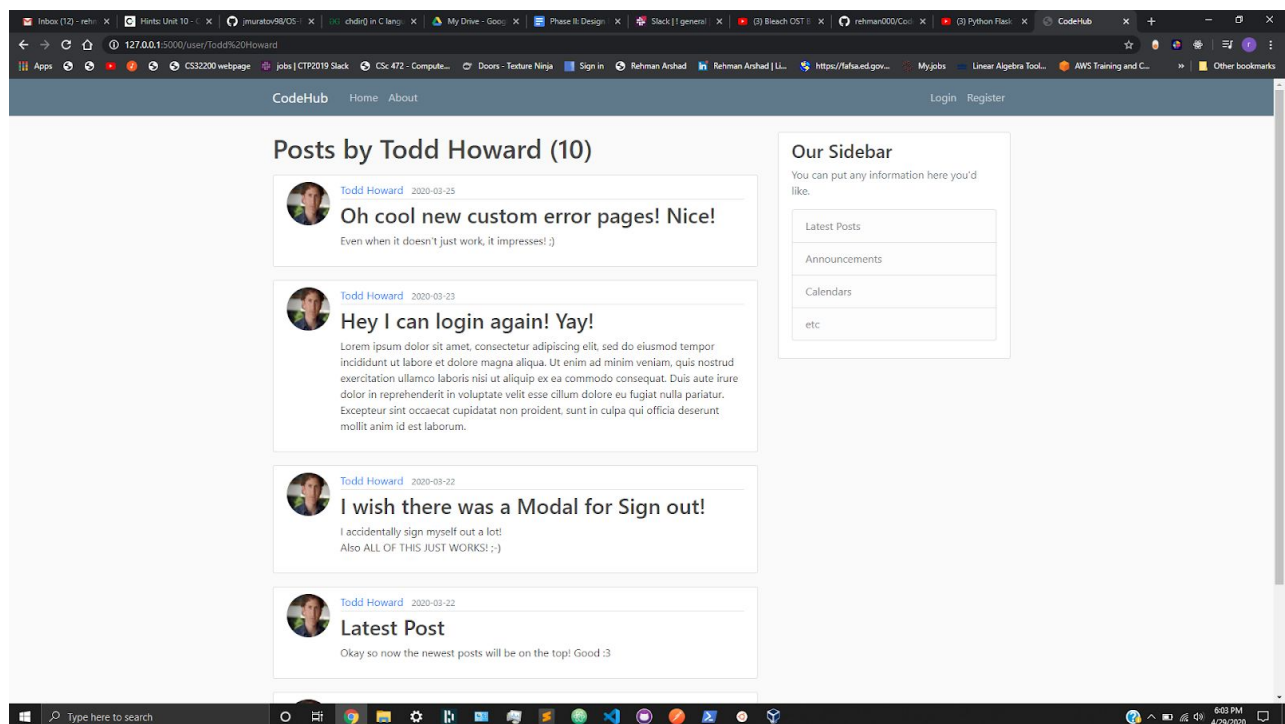
CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

5.5. Modify Post Page (Registered Users ONLY):



This page let's a registered user update or delete his post! You can access this page by clicking on a post.

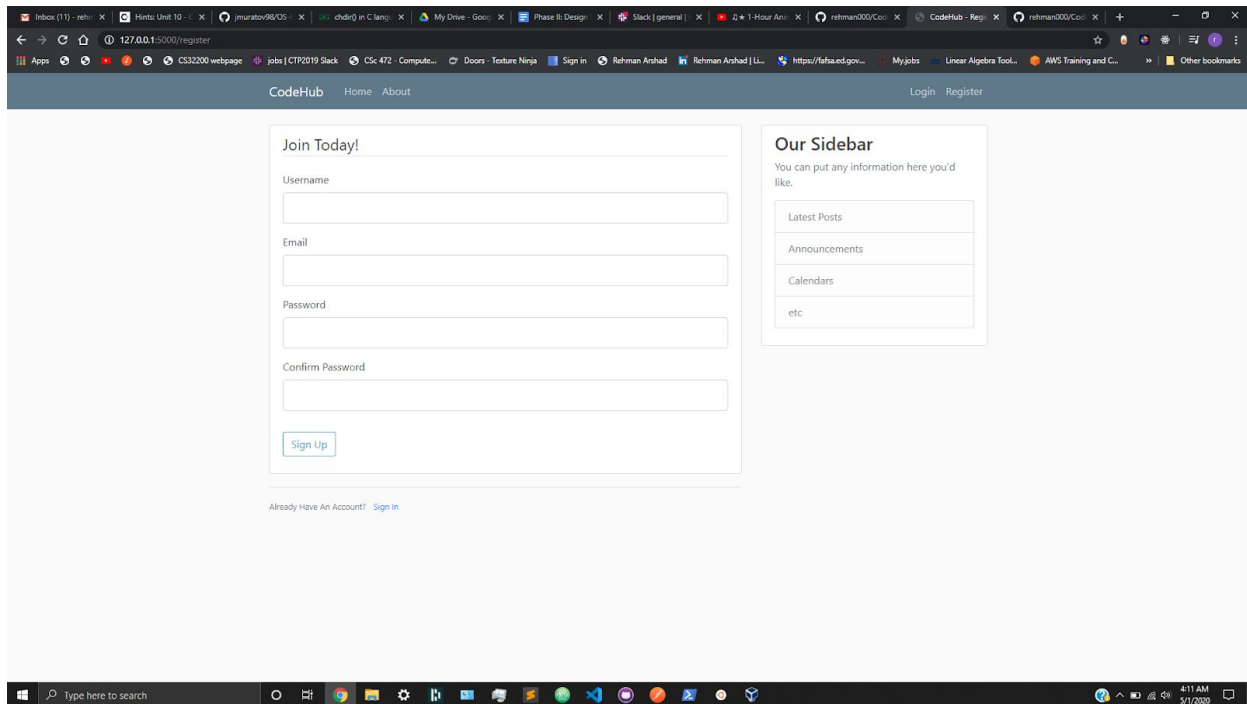
5.6. User Post Page:



This page displays all of the posts of a specific user! You can access this page by clicking on a username.

CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

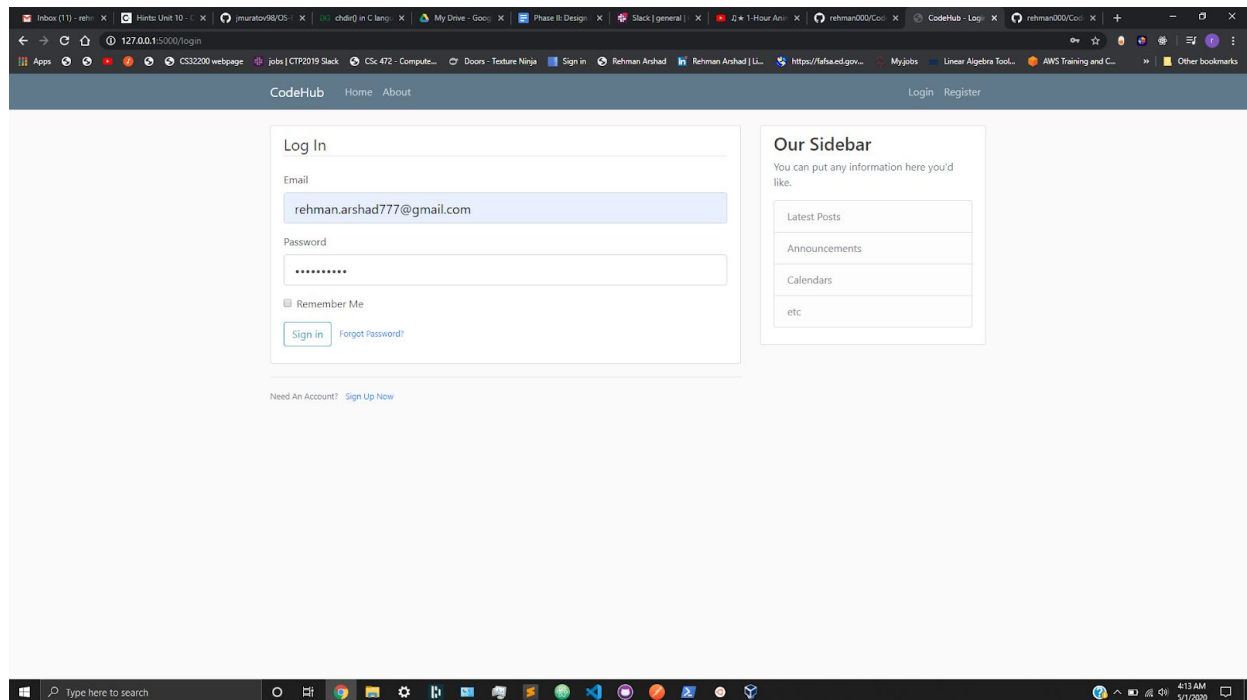
5.7. Registration Page:



The screenshot shows the registration page of CodeHub. The page has a dark blue header with the CodeHub logo and navigation links (Home, About, Login, Register). The main content area is white and contains a registration form titled "Join Today!". The form has four input fields: Username, Email, Password, and Confirm Password. Below the fields is a "Sign Up" button. To the right of the form is a sidebar titled "Our Sidebar" with a description "You can put any information here you'd like." and a list of categories: Latest Posts, Announcements, Calendars, and etc. At the bottom of the form, there is a link "Already Have An Account? Sign In". The browser's address bar shows the URL "127.0.0.1:5000/register".

On this page a visitor fills out the form and becomes a registered user.

5.8. Login Page:



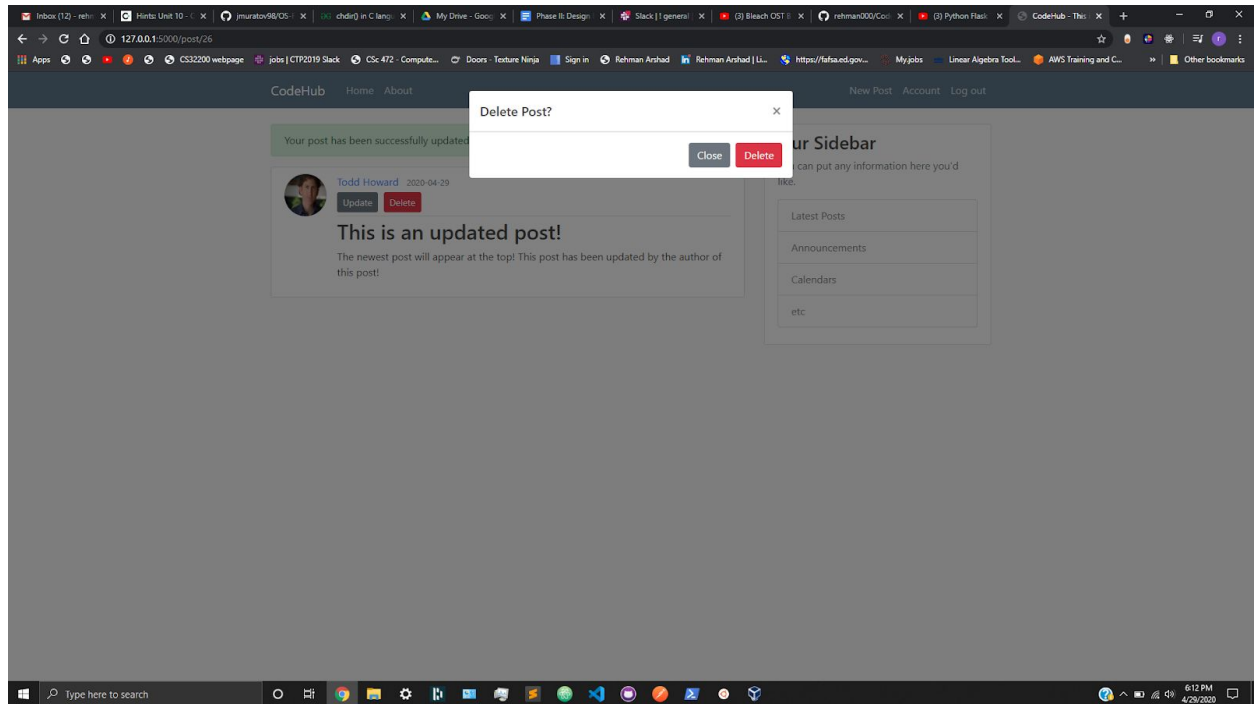
The screenshot shows the login page of CodeHub. The page has a dark blue header with the CodeHub logo and navigation links (Home, About, Login, Register). The main content area is white and contains a login form titled "Log In". The form has two input fields: Email and Password. The Email field is filled with "rehman.arshad777@gmail.com". The Password field is filled with "*****". Below the fields is a "Remember Me" checkbox and a "Sign In" button. To the right of the form is a sidebar titled "Our Sidebar" with a description "You can put any information here you'd like." and a list of categories: Latest Posts, Announcements, Calendars, and etc. At the bottom of the form, there is a link "Need An Account? Sign Up Now". The browser's address bar shows the URL "127.0.0.1:5000/login".

On this page a registered user can login to the system.

CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

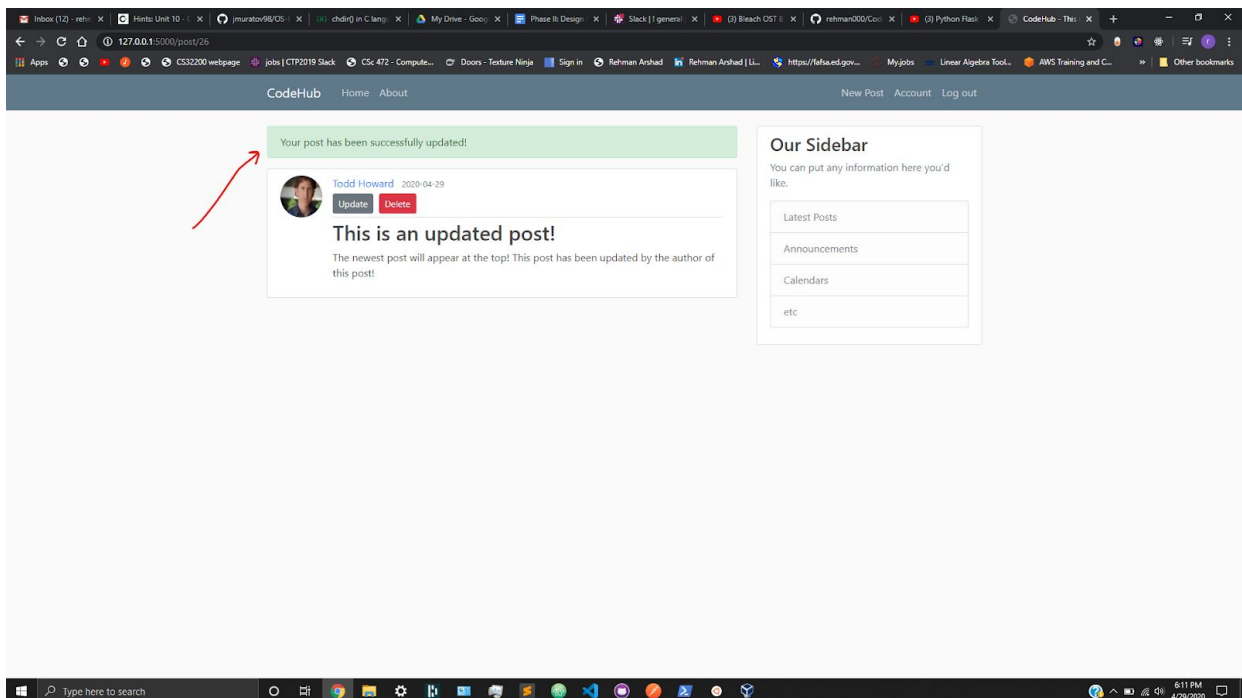
Functionality of my own choice:

5.9. Post Deletion Confirmation Prompt (Registered Users ONLY):



This is to prevent users from accidentally deleting a post, because such an action is irreversible.

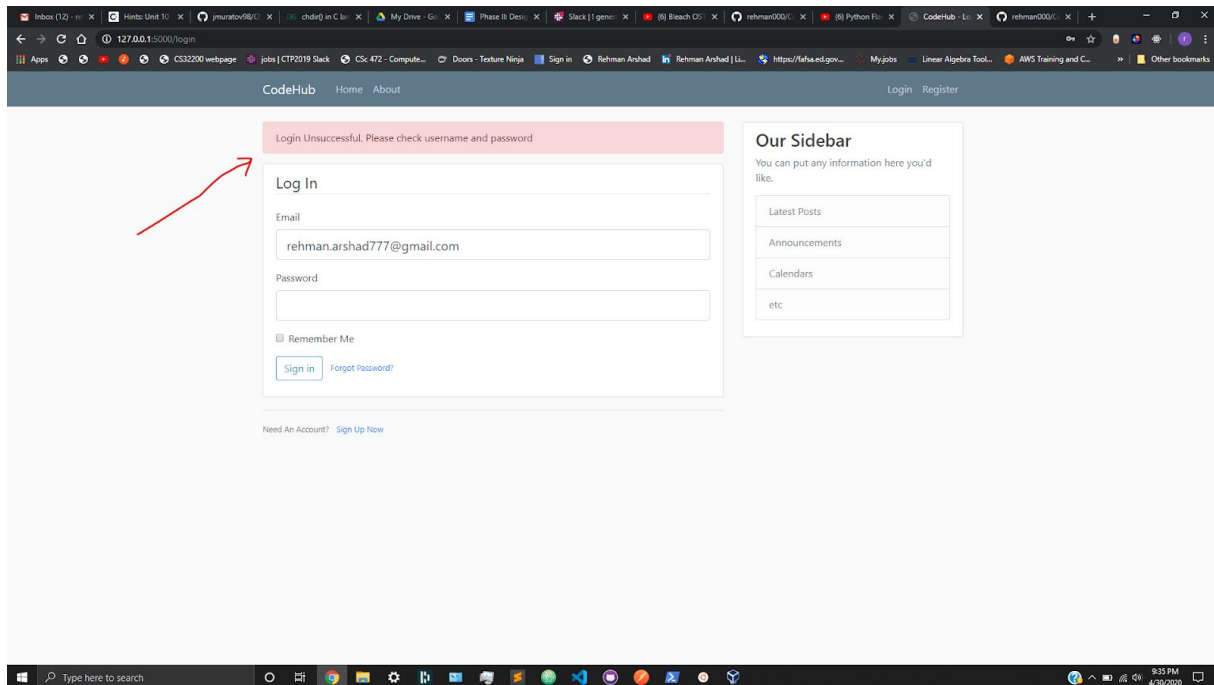
5.10. Post update/deletion notification prompt:



When a post is successfully updated/deleted the following prompt will appear.

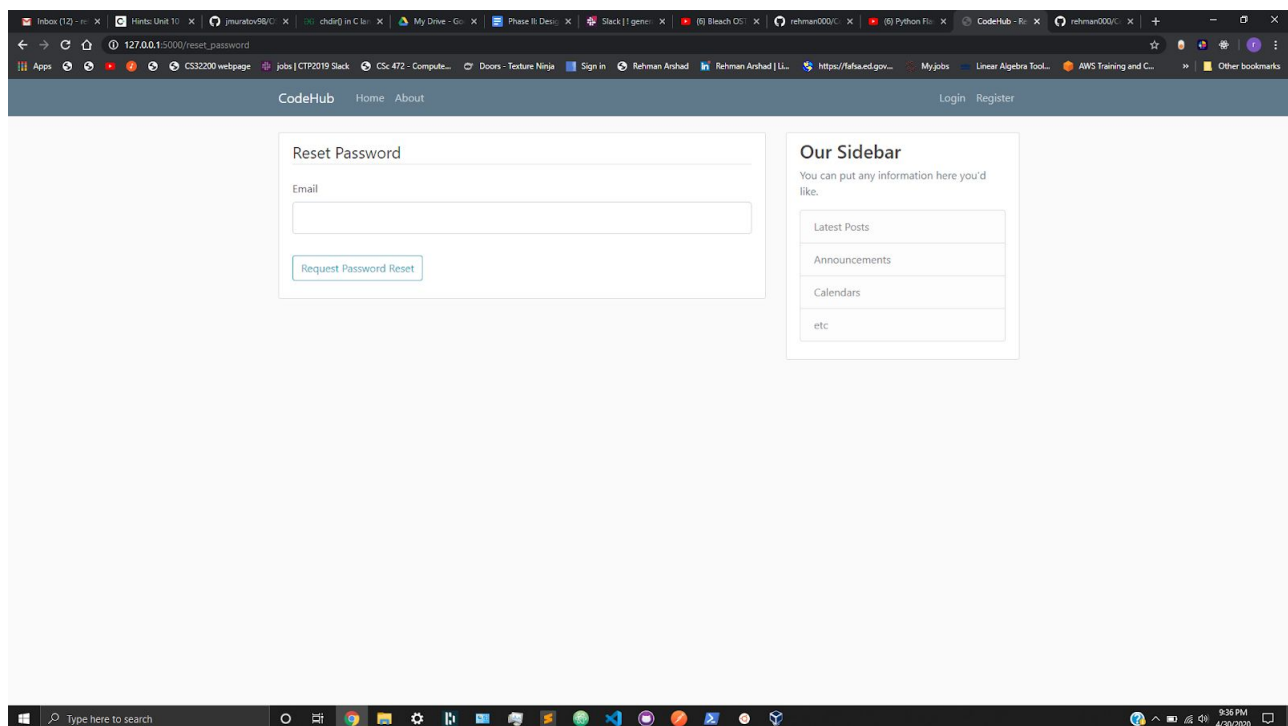
CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

5.11. Unsuccessful Login attempt prompt:



On an unsuccessful login attempt the following prompt will appear, this is utilizing the bootstrap danger class as it has a reddish hue.

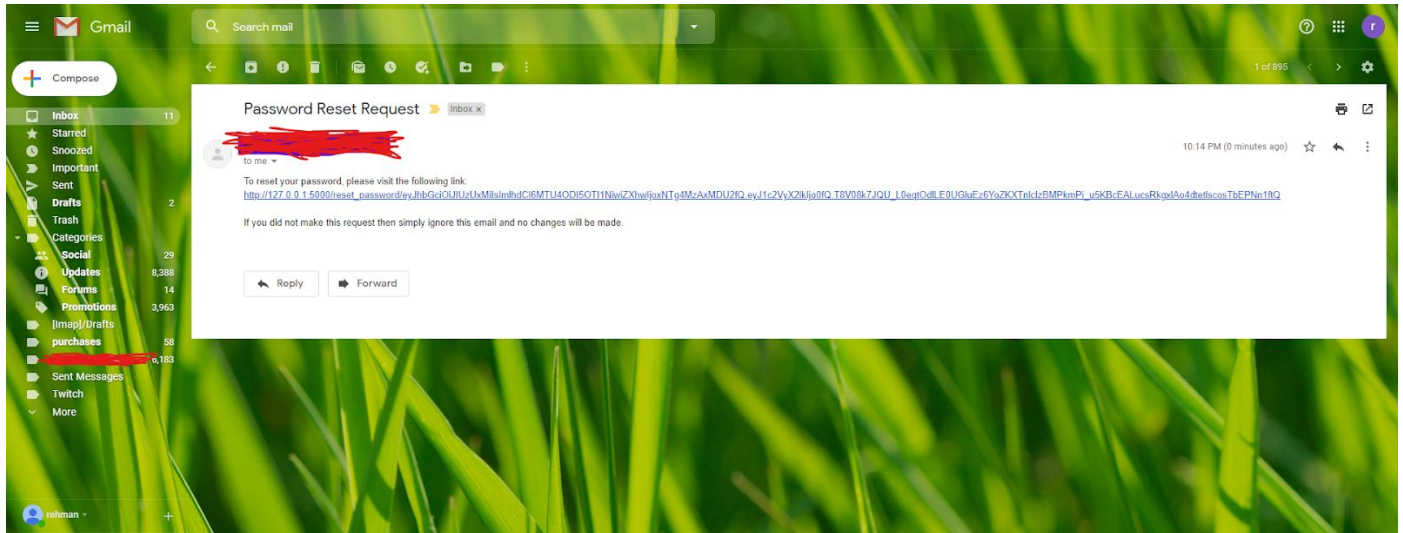
5.12. Password Reset Functionality:



This is the password reset form, over here the user can input their email address in order to get a password reset link sent to their email.

CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

5.13. Email Password Reset:



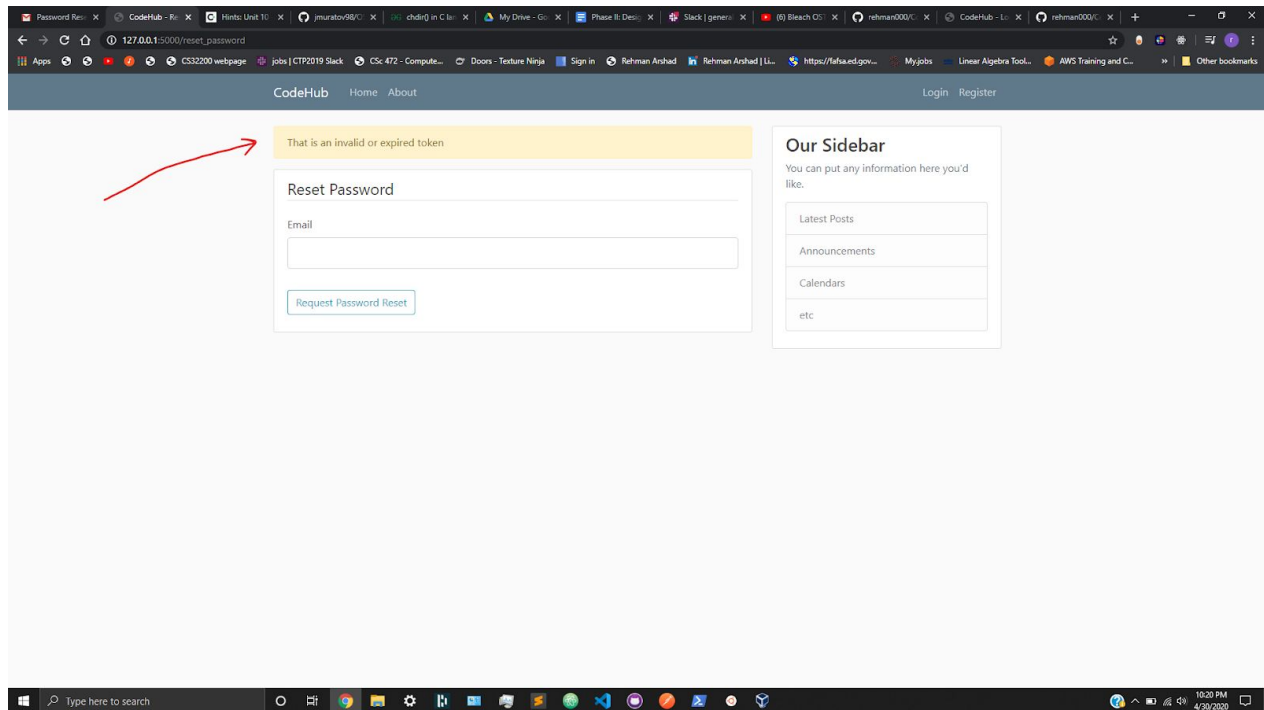
The Password Reset Link is sent to the users email.

5.14. Password Reset Form (Timed URL) with Valid Token:

When a valid token is provided the following form will appear. This is a timed event and the token expires within 40 minutes of making the request.

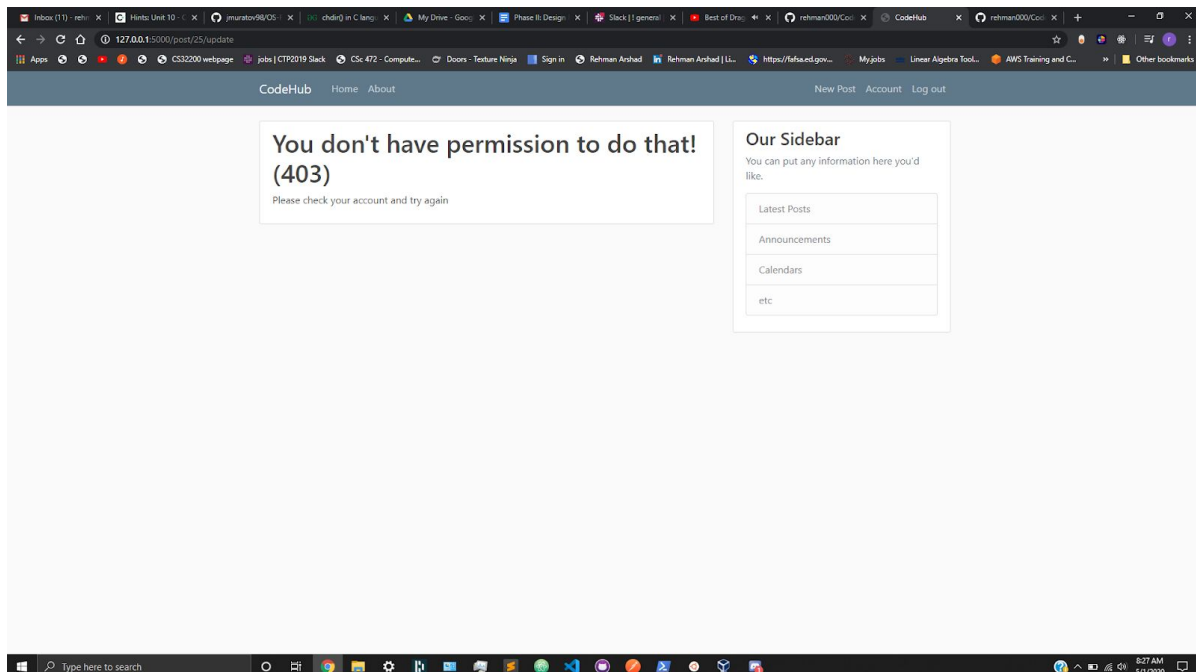
CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

5.15. With invalid Token, Redirect back to Password Reset Request Form:



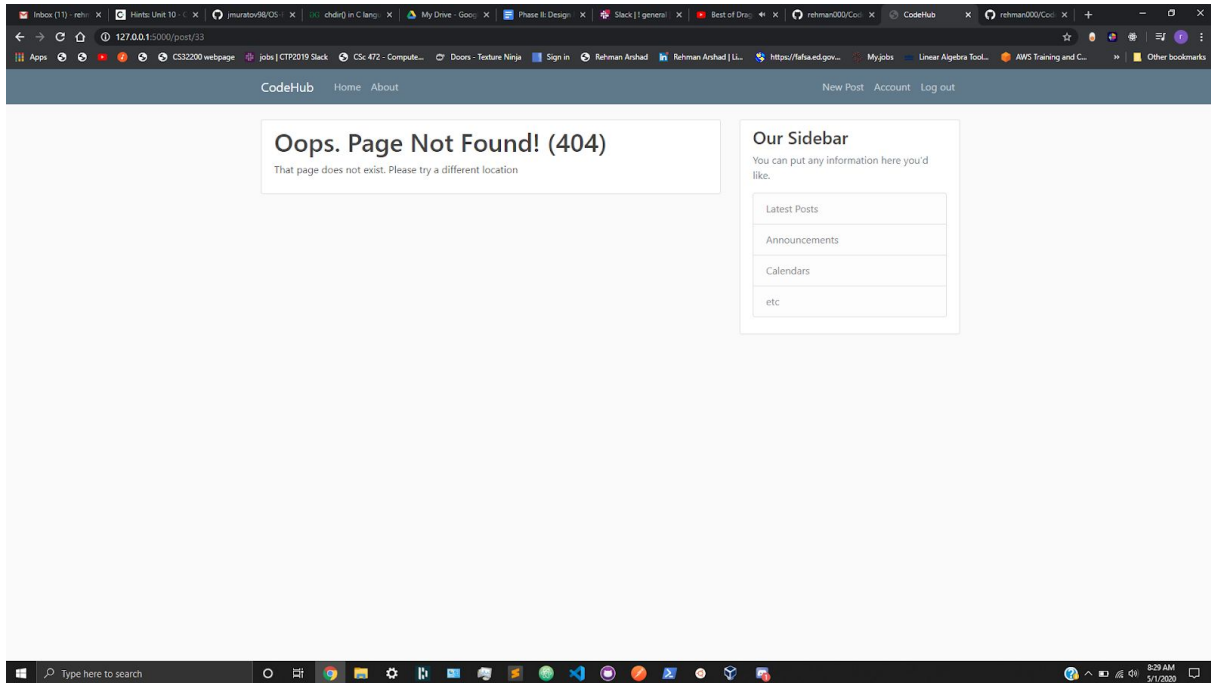
When an invalid token is provided the following form will appear. The user is redirected back to the password reset request form where he/she must re-submit their email address. The flashed message on the top is using the warning bootstrap class, and has a yellowish hue.

5.16. Graceful Error handling (403: Forbidden)



CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

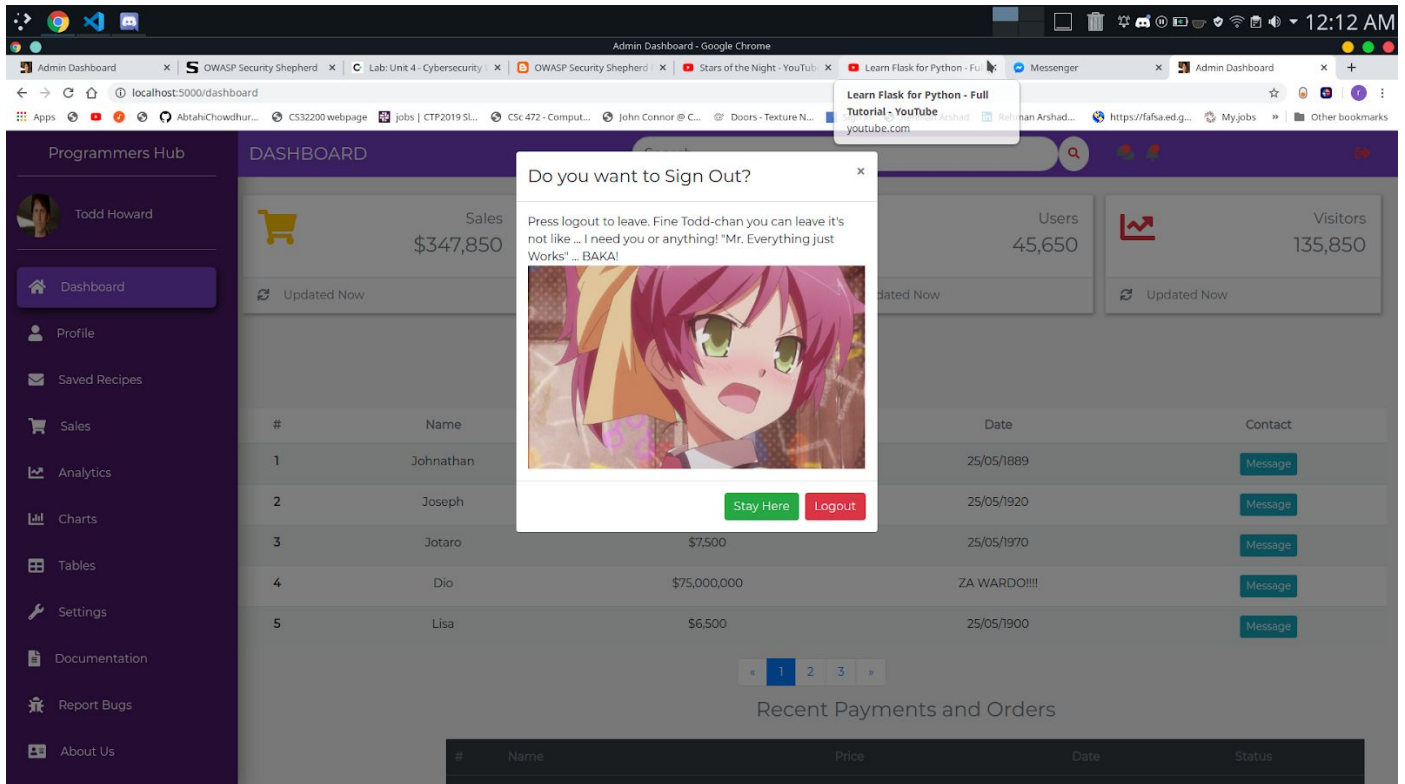
5.17. Graceful Error handling (404: Not Found)



CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

Prototypes:

Some Designs that I am currently working on that need more testing for responsiveness on all screen sizes. The SU (Super User) page, the back-end still needs to be implemented.



I made this template in the beginning of the semester and you'll see the title on the right side it says Programmer's hub, thankfully I realized that name was too generic, and changed it to Codehub. I may use this template as a baseline since Flask supports template inheritance to give the admin section a completely different look, feel, and vibe. Thankfully this template is fully responsive on all screen sizes. We just need to implement some back end logic to make this functional which will be difficult.

CodeHub	Version: 2.0
Design Report	Date: 05/1/2020
CodeHub-DesignReport.pdf	

6. Minutes of Group Meetings

The following are the topics of discussion from each of our group meetings:

1. **March 4th, 2020:** Discussed the tech stack to be used in creating the web application and divided the work evenly for the first phase report. Created GitHub repository.
2. **March 30th, 2020:** Divided the application into different modules to be divided amongst the group members. Also discussed sources of reference for our chosen framework, Flask.
3. **April 22nd, 2020:** Divided the modules and looked at the varying pages required in order to complete our system. Drew diagrams depicting our system and discussed the work required to implement the pages and features.

7. GitHub repo:

<https://github.com/rehman000/CodeHub>

Miscellaneous research:

1. Useful Cyber Security Insights:
 - [Do any security experts recommend bcrypt for password storage?](#)