To-Do List Project Documentation

Author: Blake Maxwell

12/15/2016

1. THE APPLICATION FROM THE USER PERSPECTIVE

A. ACCOUNT/LOGIN SYSTEM

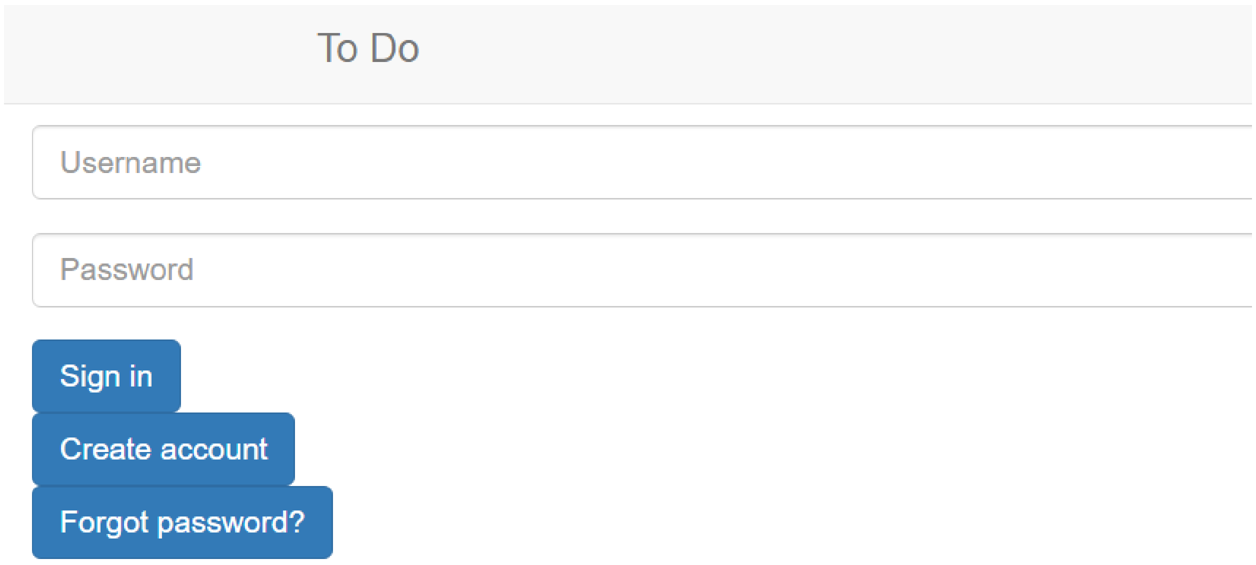


Figure 1: The login screen as viewed in a browser.

Upon loading the application for the first time in the browser, the user will see the login screen as shown above in Figure 1. At this point, the user has not created an account and may click the “Create account” button to create one. Clicking this button will lead them to a new page where they may enter the relevant information for their account as indicated by the four input fields on screen. After entering this information, the user can then click the “Create” button, and they will be redirected back to the login page. A message will flash at the top of the page to indicate that the account was successfully created. From there, the user can use their new login credentials to access their account. Figure 2 below shows the account creation page, while Figure 3 below shows the redirected login page with the success message flashed at the top:

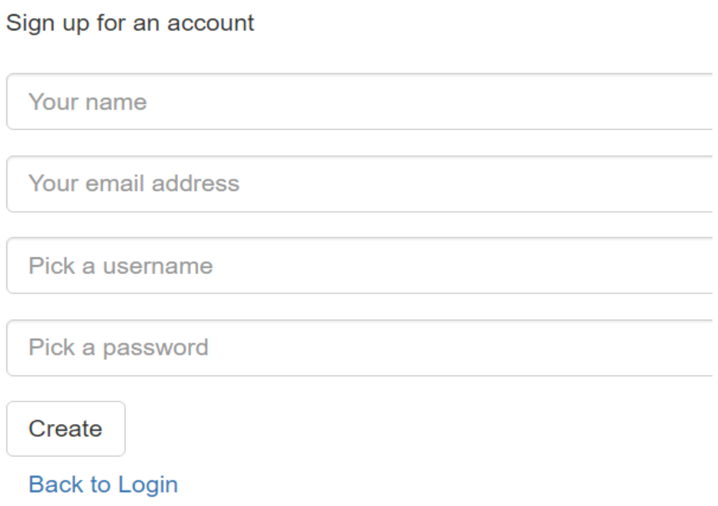


Figure 2: The account creation page

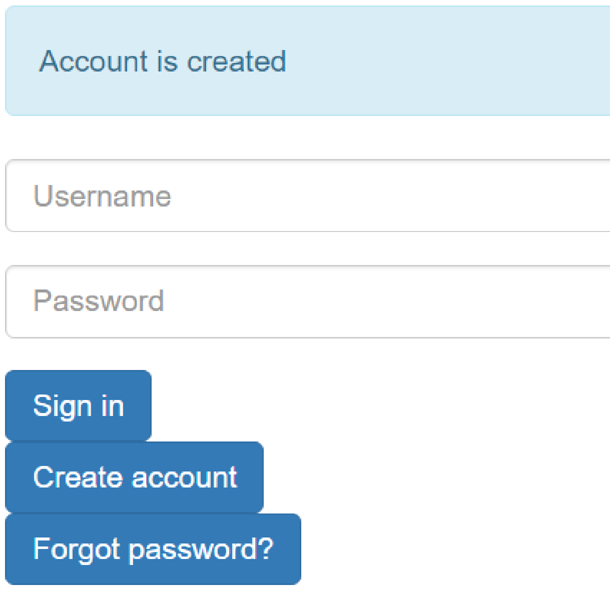


Figure 3: Login page after successful account creation

Looking at the login page, notice the "Forgot password?" button. Whenever a user forgets their password, they may click this button and be directed to another page with a single text input field that asks them to enter the e-mail address associated with their account. Once they enter the e-mail and click the “Submit” button below the field, the program will check to see if the entered e-mail address has been registered with our program. Assuming it has, the program will then generate a random seven-character long string. This string will act as the new password for that user’s account. After redirecting the user back to the login screen, a message will flash signifying that an e-mail was sent to the entered address. The user can now check their e-mail for this message containing the new password string. At the same time, the password records will be updated on our site, enabling the user to use the string right away on the login page. Figure 4 below shows the “Forgot password” page where the user can enter their e-mail, while Figures 5 and 6 on the next page show the redirect to the login page with a message shown indicating successful sending of the e-mail and the sent e-mail containing the random string respectively:

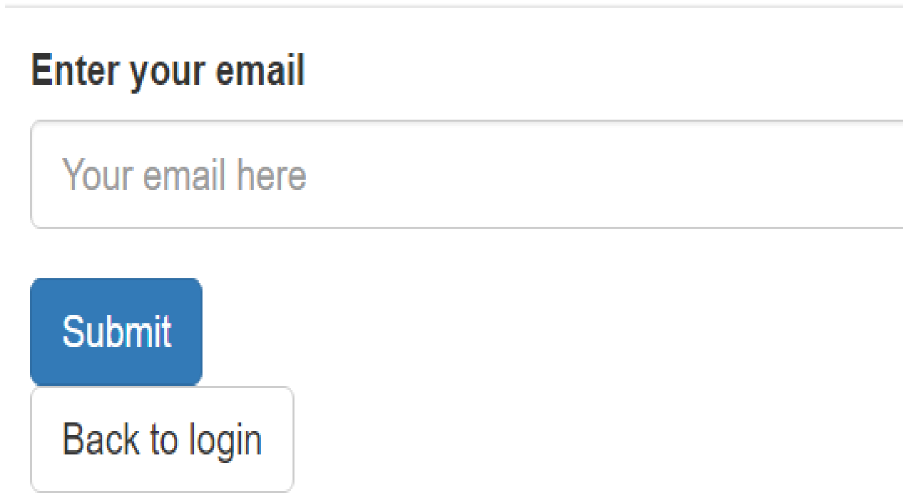
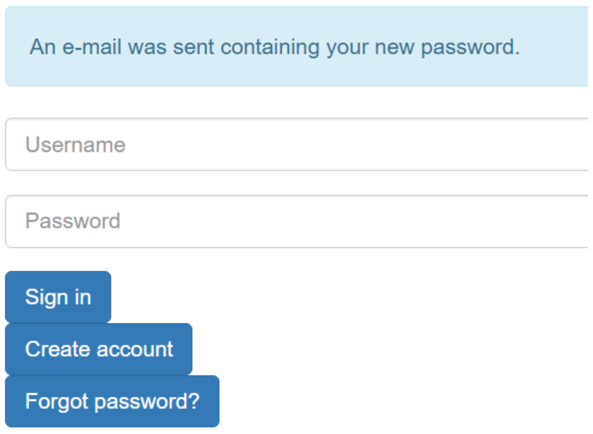
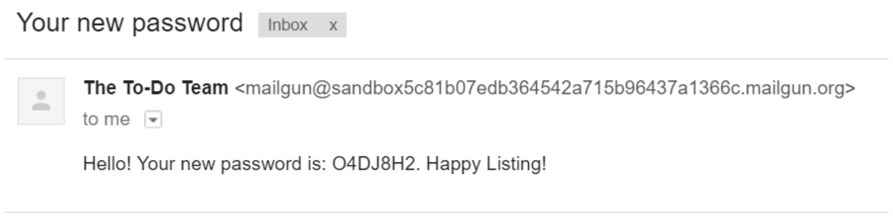


Figure 4: The e-mail entry page for the password reset process



An e-mail was sent containing your new password.

Figure 5: The login screen after successfully sending an e-mail for password reset



Your new password

The To-Do Team <mailgun@sandbox5c81b07edb364542a715b96437a1366c.mailgun.org>
to me

Hello! Your new password is: O4DJ8H2. Happy Listing!

Figure 6: The e-mail containing the randomly generated string

B. THE EMPTY TO-DO LIST

After successfully logging in for the first time, the user will see the screen shown in Figure 7 below. This is the main interface for the application. At this point, the user has not added any items to their list, leaving it empty. The empty state of the list is made apparent by the "You're all done" status message displayed underneath the highlighted "Current" heading. Any items on the list that the user has added but has yet to complete will be displayed under this "Current" heading, as it represents the items that are currently "active" and need attending to. Also of interest is the highlighted "Logged in as [username]" message displayed in the top left corner of the screen. This message will persist as long as that specific user is logged in, thus conveniently allowing anyone to know what user is currently using the application.

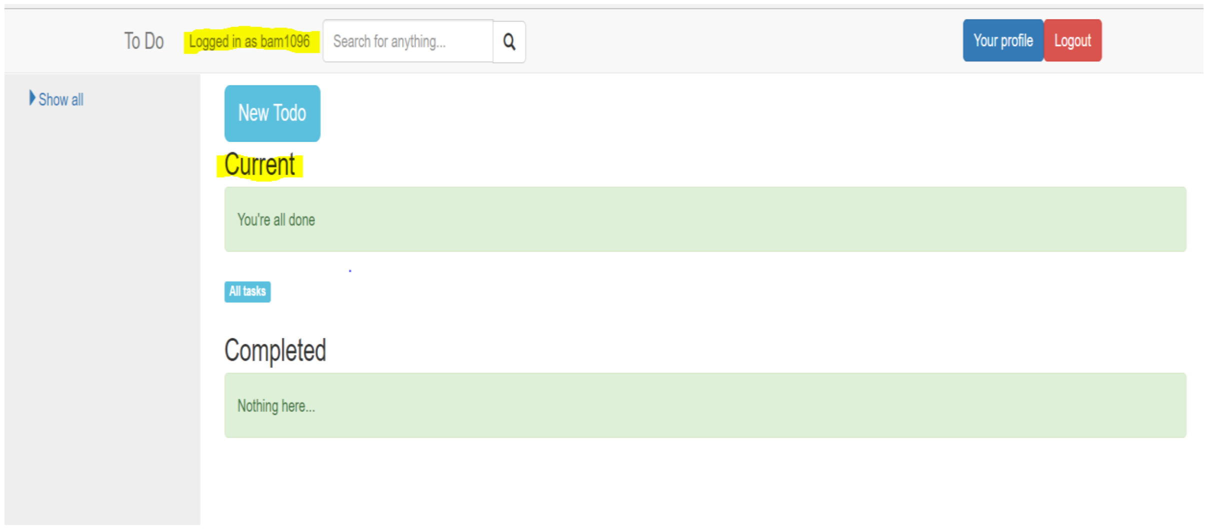


Figure 7: An empty to-do list after logging in for the first time

C. ADDING ITEMS TO THE LIST

Obviously, the to-do list is of no use without any items. To add a new item to the list, the user may simply click on the "New Todo" button as shown in Figure 7 above. After clicking the button, a small window will pop up containing five different input fields. Each field can hold information for the different parts of a list entry. These are: the "Task," the "Description", the "Time," the "Date," and the "Category." As such, the user may give the new list entry a name, attach a description to the entry, associate a time and date with it, and place it within a category so as to group it together by a common theme. These categories play a critical role in our method for filtering the list which will be described later. Figure 8 on the next page shows this window and its text fields:

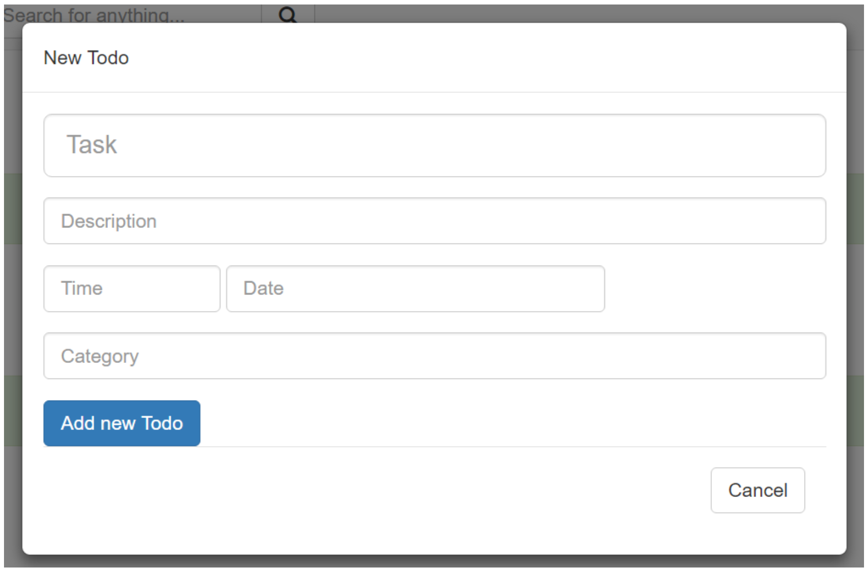


Figure 8: The window to add a new item to the list

While the input fields for the "Task," "Description," and "Category" sections of the entry are simple text input fields, we have attached a time and date picker to the "Time" and "Date" fields. When a user clicks on the "Time" field, for example, a menu of times will appear. The times in the menu are divided up by two minute intervals (5:00 PM, 5:02 PM, 5:04 PM, and so on, for instance). The user may scroll through the menu and select the appropriate time by clicking on it. The field will then automatically fill with the selected information. Figure 9 below shows the use of this time picker in adding an entry:

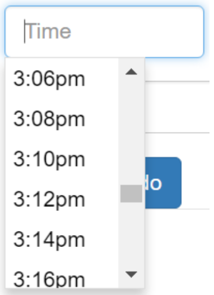
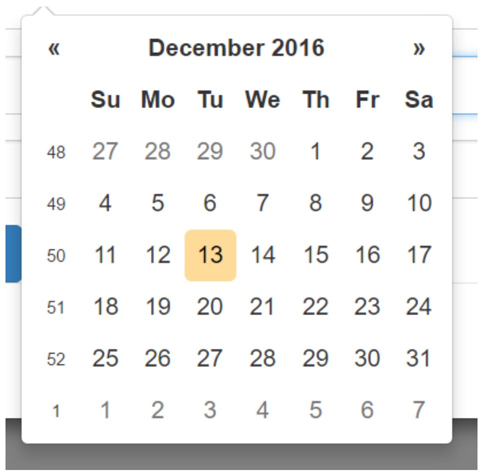


Figure 9: Use of the time picker

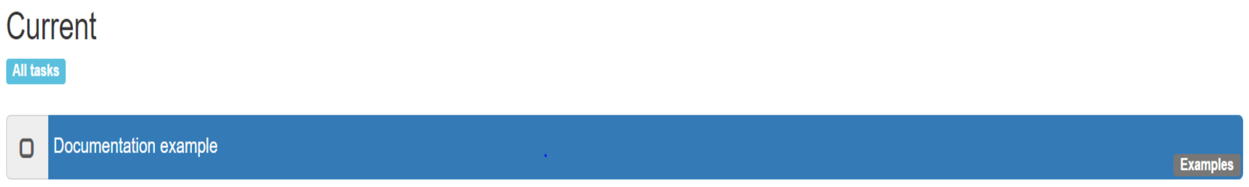
Likewise, when the user clicks on the “Date” field, they will be presented with a calendar. By default, the current date is selected on the calendar. However, by navigating the calendar using the arrows on top, the user can select any date they wish. Once a date has been selected by clicking on it, the field will automatically be filled with the relevant information in month/day/year format. Figure 10 below shows the use of this date picker:



	Su	Mo	Tu	We	Th	Fr	Sa
48	27	28	29	30	1	2	3
49	4	5	6	7	8	9	10
50	11	12	13	14	15	16	17
51	18	19	20	21	22	23	24
52	25	26	27	28	29	30	31
1	1	2	3	4	5	6	7

Figure 10: Use of the date picker

Use of these pickers helps to ensure consistent formatting of dates, and is important for the proper function of the e-mail reminder system that will be described later. After every field has been filled with the relevant information, the user can add the item to the list by clicking the “Add new Todo” button. The new entry will then appear in the list under the “Current” heading as shown in Figure 11 below:



Current

All tasks

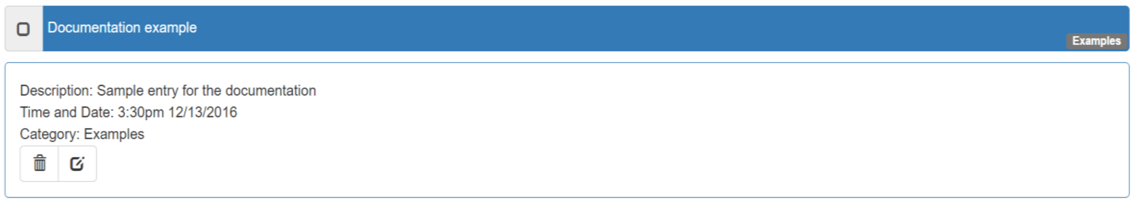
Documentation example

Examples

Figure 11: The “Current” list after posting a new entry.

D. ACCESSING ENTRY INFORMATION

Look back at Figure 11. When an entry is posted into the list, the user is immediately able to see two parts of it: the entry name, and the category. The entry name appears in white letters on the far-left side of the blue rectangle. In the case of Figure 11, this name is “Documentation example.” The category, however, appears at the bottom right of the blue rectangle. In the case of Figure 11, this category is “Examples.” It is likely that the user would like to be able to see the other parts of this entry at any given time. This may be accomplished by clicking within the blue rectangle. Once the blue rectangle is clicked, a small menu will drop down from it, revealing the entry description, date and time, and category. Figure 12 below shows this menu and the associated information:


Documentation example

Examples

Description: Sample entry for the documentation

Time and Date: 3:30pm 12/13/2016

Category: Examples

Figure 12: The additional entry information

E. DELETING AND EDITING ENTRIES

Notice the trash can and pen-to-paper icons at the bottom of the drop down menu in Figure 12. These icons allow the user to delete and edit an entry respectively. Deleting an entry is accomplished by simply clicking on the trash can icon. Clicking on the editing icon will bring up a menu nearly identical to the one used for adding entries. By entering information into the appropriately labeled text fields, or using the date and time pickers in the case of those fields, and then clicking the “Confirm” button, the entry will be updated to reflect the desired changes.

F. FILTERING BY CATEGORY

Recall our earlier statement that the categories attached to each entry play an important role for our method of list filtering. We have implemented a way for users to filter the list by category. Using this method, the users can choose to only see entries of a specific category if they like. In this way, lists containing many entries will become easier to manage, and users can look at tasks centered around a common theme or event. Looking back at Figure 7, notice the panel on the far-left side of the page. Each time an entry with a new category is added to the list, that category name will appear in this panel. Each category name there is a link that, when clicked, will filter the list so as to only show items of a specific category. We have also included a link that says “Show all” as seen in Figure 7. By clicking on this link, the user can restore the list to its original state before filtering, displaying all entries on the screen again. Also, notice the label “All tasks” underneath the “You’re all done” message in Figure 7. This label will change when the user enacts a filter on the list, showing the name of the category currently being filtered by. With this handy indicator, the user always knows what category, if any, they are currently viewing. Figure 13 below shows this filtering in action:

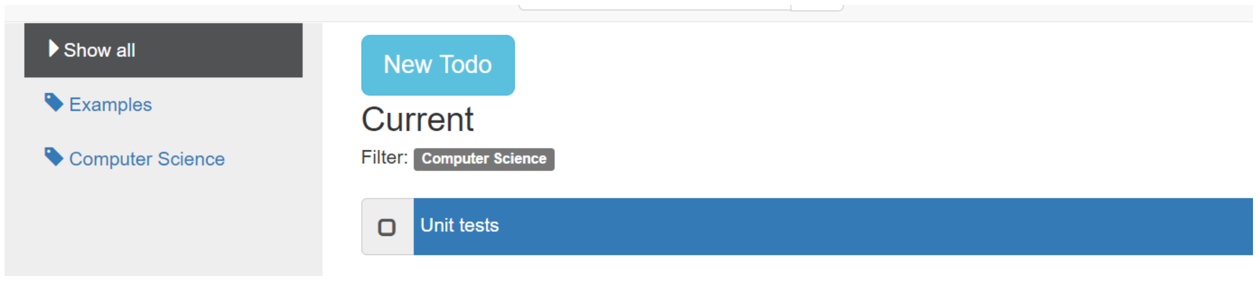
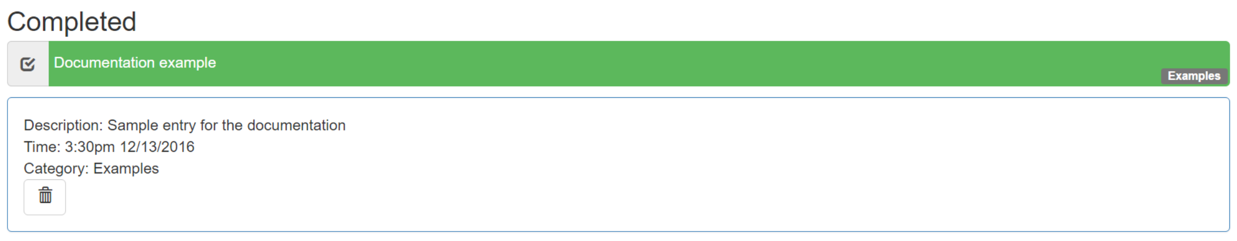


Figure 13: The list after filtering by the “Computer Science” category. Note that there is currently one entry under this category.

G. MARKING LIST ITEMS AS “COMPLETE”

Recall that there are two lists displayed to the user in this program. One of these is denoted “Current” while the other is denoted “Complete.” These lists can be clearly seen in Figure 7. The reason for these two lists is made apparent when we examine the “Documentation example” entry of Figure 11. Notice the gray tab with a small box on it to the far-left side of the entry. This box is a checkbox. When the user clicks it, they signify to the program that they have completed that item. Thus, the item is moved from the “Current” list to the “Completed” list. Once the item is in the “Completed” list, they may optionally delete the item as described in section E to avoid cluttering the list with unnecessary entries. These two lists add an additional layer of organization to the list by separating what still needs to be done versus what has been done. Thus, the user does not have to spend time recalling what events they have already completed. At any time, the user may also uncheck the checkbox, moving the entry back into the “Current” list. Figure 14 below shows an entry that has been marked and moved to the “Completed” list and has its drop down menu open in preparation for deletion. Notice how the color of the entry has changed from blue to green to signify completion:



Description: Sample entry for the documentation
Time: 3:30pm 12/13/2016
Category: Examples

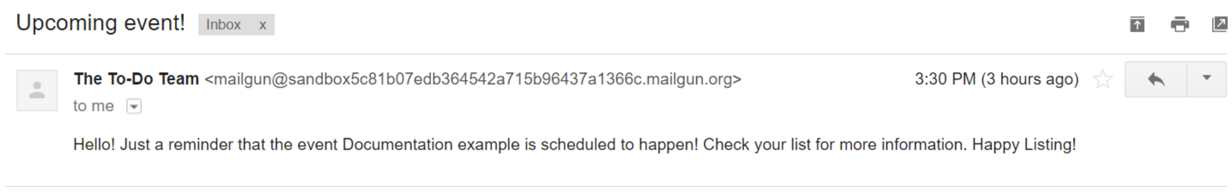
Figure 14: An entry in the “Completed” list

H. SEARCHING FOR LIST ITEMS

Once again, observe Figure 7. Notice the text input field at the top of the page next to the "Logged in as [username]" message. As the "Search for anything… text within the field suggests, this feature allows a user to type in any part of a list entry and click the search button. The program will then scan the database, looking for any entries that match the search by name, description, time, date, or category and return the results to the user by displaying them on screen. Having this feature allows the user to quickly look up any list items without having to scroll if the list is very long. They can then delete or edit the items they searched for as needed.

I. E-MAIL BASED REMINDERS

Perhaps the biggest difference between this program and the use of a traditional, hand-written to-do list is this program's ability to output e-mail based reminders for tasks. Using the inputted date and time of each list item, our program is set to scan through the current user's list every minute that the program is active. While scanning through this list, the program obtains the current date and time from the computer's clock and compares it to the date and time attached to each entry. If a match is detected, then the program will send out an e-mail to the user announcing that an event is currently scheduled to happen. This feature helps users to remember events they have scheduled within the program that they may have forgotten since last using the program. Additionally, the entire process is done automatically; the user does not have to do anything after creating a list entry. Figure 15 shows an example of an e-mail that has been sent via this process. By comparing the time that this e-mail was sent with the entry information as shown in Figure 12, we can see that the e-mail was sent at the exact time for this event as specified by the user:


Upcoming event! Inbox x

The To-Do Team <mailgun@sandbox5c81b07edb364542a715b96437a1366c.mailgun.org> 3:30 PM (3 hours ago)

to me

Hello! Just a reminder that the event Documentation example is scheduled to happen! Check your list for more information. Happy Listing!

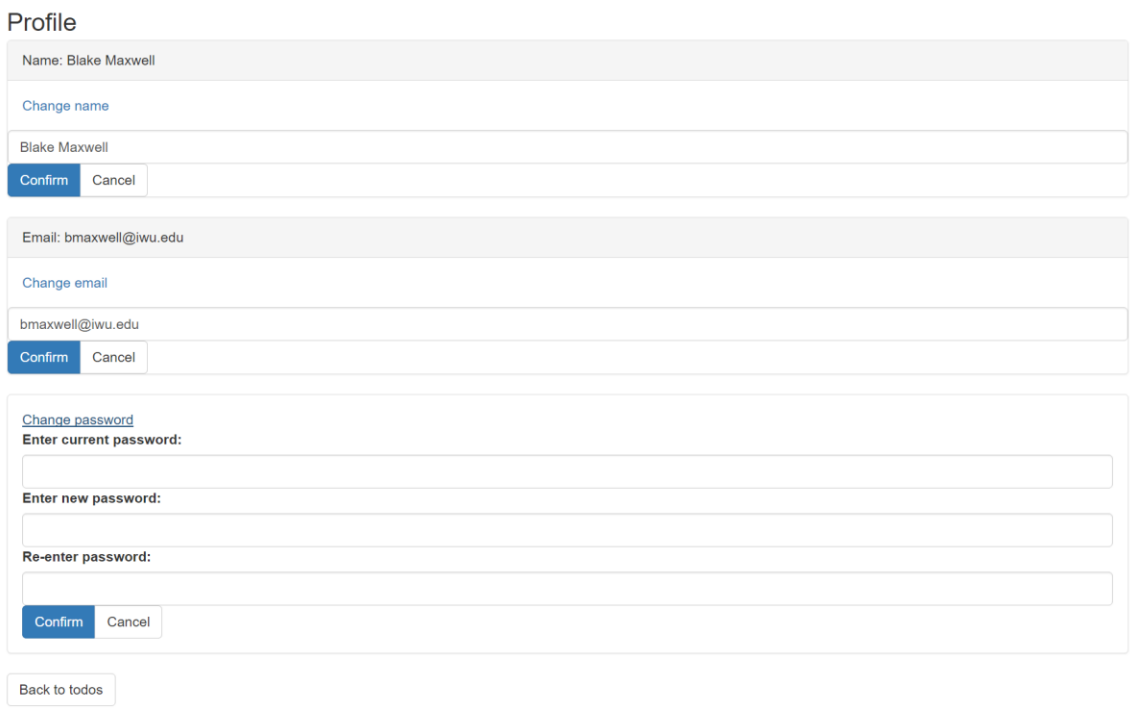
Figure 15: An example of an e-mail sent for reminder purposes

J. LOGGING OUT

When a user wants to end the current session, all they must do is click the "Logout" button displayed in the top-right corner of the page as shown in Figure 7. The user will then be redirected to the main login screen as shown in Figure 1.

K. CHANGING PROFILE SETTINGS

Notice the "Your Profile" button at the top-right of the screen as shown in Figure 7. Clicking this button will bring up a menu that allows the user to change the name, the e-mail address, and the password associated with their account. To change any of these settings, the user may click any of the links on the menu: "Change name," "Change e-mail," or "Change password." Doing so will reveal a text input field, or in the case of a password change, three text input fields, underneath the link where the user can type the relevant information to perform a change. After clicking the "Confirm" button at the bottom of this input field(s), the profile credentials will be updated in accordance with these changes. Figure 16 on the next page shows the menu where profile changes may be entered. Notice how we do not display the current user's password for the sake of account security:


Profile

Name: Blake Maxwell

Change name

Blake Maxwell

Confirm Cancel

Email: bmaxwell@iwu.edu

Change email

bmaxwell@iwu.edu

Confirm Cancel

Change password

Enter current password:

Enter new password:

Re-enter password:

Confirm Cancel

Back to todos

Figure 16: The profile updating menu

2. THE APPLICATION CODE

A. THE DATABASE

At the heart of our program lies a database, written using SQLite, where all of the information relevant to our program is stored. This data is stored using three separate tables: one for storing user profiles, one for storing to-do list items that are currently active, and one for storing to-do list items that the user has marked as complete. Figure 17 below shows the this database code:

```
/*Profile database*/
drop table if exists profile;
create table profile (
        id integer primary key autoincrement,
        'username' text not null unique,
        'password' text not null,
        'email' text not null,
        'name' text not null
);
```

```
/*user todos databases*/
drop table if exists todos;
create table todos (
        id integer primary key autoincrement,
        'username' text not null,
        'taskname' text not null,
        'taskdes' text not null,
        'tasktime' text not null,
        'taskcategory' text not null
);

drop table if exists completed;
create table completed (
        id integer primary key autoincrement,
        'username' text not null,
        'taskname' text not null,
        'taskdes' text not null,
        'tasktime' text not null,
        'taskcategory' text not null
)
```

Figure 17: The SQLite database code

Notice how through the use of three different tables, we avoid redundancy and confusion throughout the database. As highlighted, we can see that the tables often share row names, such as “username.” If we were to combine all of this information into one table, we would have to make sure that the variable names did not conflict with each other, leading to confusion as we try to distinguish between different items in the database as part of Python code. Additionally, observe how each table entre is set to have an ID value that increments automatically each time an entry is added to the table. In this way, each entry has a unique identifier that can be used to locate it specifically when users use the search function. As we will see later in Python code, the IDs are also used to order entries on the rendered screen.

B. ACCOUNT/LOGIN SYSTEM

If we look back to the rendered login page as presented in Figure 1, we see that there are two text input fields for the user to type their username and password respectively. Following these fields are the "Sign in," "Create account," and "Forgot password?" buttons. We are going to demonstrate how these features work through HTML and Python code by walking through an example of a user creating a new account. This functionality will then be the basis for many application features. To start, let us observe the login page in HTML code through Figure 18 below, specifically the "Create account" button:

```
<form action="{{ url_for('create') }}" method="post">
    <button type="submit" class="btn btn-primary">Create account</button>
</form>
```

Figure 18: The "Create account" button in HTML code

Observing this figure, we see that the "Create account" button is nested within two form tags. Notice the action attribute for this form. By specifying the action in this {{ url_for…}} format, we are essentially stating that when the button inside this form is clicked, a request for the URL associated with the Python 'create' function is made with the server and sent back to the browser. Therefore, you will see /create at the end of the URL in your browser. Now, let us look at the Python code related to the 'create' fnction in Figure 19 below:

```
@app.route('/create', methods=['POST'])
def create():
    return render_template('create_account.html')
```

Figure 19: The Python code for the 'create' function

Take note of the first line of this code. Lines like this that begin with the @ symbol are known as Python decorators. By using them, we specify that when the browser is directed to the URL ending with, in this case, “/create” , the function underneath the decorator should be run. The return render_template(‘create_account.html’) line that comprises the create function instructs the browser to render the ‘create_account.html page. This is one way in which we direct users to different pages on our site. Now, the user should see the page on screen as shown in Figure 2. We can now use this page to discuss the transfer of user input along with requests through HTML forms. Observe the HTML code for the account creation page shown below in Figure 20:

```
    <form action="{{ url_for('create_account') }}" method="post">
        <div class="form-group">
            <input class="form-control" type="text" placeholder="Your name"
name="name">
        </div>
        <div class="form-group">
            <input class="form-control" type="text" placeholder="Your email
address" name="email">
        </div>
        <div class="form-group">
            <input class="form-control" type="text" placeholder="Pick a
username" name="uname">
        </div>
        <div class="form-group">
            <input class="form-control" type="password" placeholder="Pick a
password" name="pw">
        </div>
            <button type="submit" class="btn btn-default">Create</button>
    </form>
```

Figure 20: The form within the HTML of the account creation page.

From this figure, we can see that both the “Create” button as well as all of the text input fields are nested inside the form. By nesting like this, we ensure that when the button is clicked to submit a request, the data entered by the user is submitted along with it. We can then access this data in Python by writing code in the form `request.args[‘name’]` or `request.form['name']`. Whether we use “args” or “form” depends on the method attached to a particular HTML form and Python function. If the method specified inside of the form is “get,” then “args” is used. However, if the method is “post,” then “form” is used. A specific use of “args” will be highlighted when discussing list filtering by category. Just know for now that we are using “form” as the method is “post.” This can be seen in the Python code of Figure 21 below:

```
      @app.route('/create_account', methods=['POST'])
      def create_account():
          db = get_db()
          cur = db.execute('SELECT username FROM profile WHERE username = ?',
[request.form['uname']])
          data = cur.fetchall()
          if len(data) == 0:
              db.execute('INSERT INTO profile (name, email, username,
password) VALUES (?, ?, ?, ?)',
[request.form['name'], request.form['email'], request.form['uname'],
hash(request.form['pw'])])
              db.commit()
              flash('Account is created')
              return redirect(url_for('index'))
          else:
              flash('Account already exists')
              return redirect(url_for('index'))
```

Figure 21: Python code for the ‘create account’ function

Notice the repeated use of `request.form['name']` to access information from the form input fields that was sent along with the request. In this syntax, the ‘name’ within brackets corresponds with the “name” attribute of each text input field within the form. Using these names, we specify what field we are retrieving information from. With this information in hand, we then check to see if the entered username is already in the database. If it is, then we redirect the browser back to the URL associated with the “index” function through the `return_redirect(url_for('index'))` statement. This index function, as shown in Figure 22 on the next page, then instructs the browser to render the login.html page on screen, as no user has been logged in yet:

```
@app.route('/')
def index():
    if 'logged_in' in session:
        return redirect(url_for('mainapp'))
    return render_template('login.html')
```

Figure 22: Python code for the 'index' function

If the username is not taken, we then insert the appropriate information from the form fields into the profile database, thus creating a profile. Of special note here is the use of the hash function on the entered password information prior to its entry in the database. Using this function adds security to the password storage by making it difficult to obtain the original password from raw data. A hash itself is a random string or number generated from a text string. The variations in this hash are where the difficulty in obtaining hashed passwords comes from, as each password has a unique hash.

Once our account has been created, we then use the same approach used within the create account code for logging in. As we need to keep track of which user is logged in at any time, we use the session object as provided by the Flask module within our login code. After comparing the input field data entered by the user on the login screen with information from the database, and finding that the login information is correct, we then pass the entered username into the session object, associating it with the '`logged_in`' key, where it will stay until the user logs out of the program. Then, we redirect the browser to the URL associated with the "index" function, where it will see the session object and thus render the 'mainapp.html' page on screen.

Thinking about how the forgotten password reset feature works, the only concept that is new is the introduction of the Mailgun API. Mailgun is a third-party site that allows users to send e-mails for free using their API via the Python ‘requests’ module. The code in Figure 23 demonstrates how this e-mail is constructed:

```
    def send_simple_message(address, newitem):
        return requests.post(

"https://api.mailgun.net/v3/sandbox5c81b07edb364542a715b96437a1366c.mailgun.o
rg/messages",
        auth=("api", "key-88cd21f006e0728e598dc4972c3b014f"),
        data={"from": "The To-Do Team
<mailgun@sandbox5c81b07edb364542a715b96437a1366c.mailgun.org>",
              "to": [address],
              "subject": "Your new password",
              "text": "Hello! Your new password is: " + newitem + "." + "
Happy Listing!"})
```

Figure 23: Constructing the e-mail

As shown, the e-mail is comprised of four distinct parts. Each of these parts is a key value of the ‘data’ dictionary. Notice the ‘address’ and ‘newitem’ variables used within this code. ‘address’ stores the e-mail address that we asked the user to provide in the form field on the forgotten password page. ‘newitem’ is randomly generated string of seven characters we previously mentioned in our view of the application from the user’s perspective. The code above was taken directly from Mailgun’s website, and represents the standard format for constructing e-mails in Python using this API. In addition to running this function when a user enters the e-mail address for their account, we also use an SQL UPDATE statement to update the “password” information for that user in the profile database table. In this way, after redirecting the browser back to the login page, the user can use the string presented in the e-mail as their password right away.

C. DISPLAYING LIST ITEMS

Since the list will be empty after a user logs in for the first time, let's look at how items within the list are displayed. Displaying the list is handled by the 'mainapp' function within our Python code. Within this code, we use SQL SELECT statements to obtain the relevant information for each item currently in the 'todos' and 'completed' tables, ordering them by their IDs. Storing the data from these SQL statements in variables, we then insert this data into the 'mainapp.html' page, accessing entries through dot notation like `entry.taskcategory`. By looking at the mainapp.html code, you can see this notation at work. We loop through the values stored in the variables and render them on screen through HTML, using Bootstrap and CSS to give each list entry its styling, from the blue rectangle that serves as the background, to the white text of the taskname. Observe the line of code presented in Figure 24:

```
  return render_template('mainapp.html', todo=data, category=data2,
user=data3, completed=data4, show=show)
```

Figure 24: Rendering the 'mainapp.html' page with variables

With this line at the end of the 'mainapp' function, we "pass" the information stored inside of each variable to the 'mainapp.html' page. Using this information, the page is rendered on screen, showing this information for the user.

D. ADDING LIST ITEMS

Adding list items, from a code standpoint, follows nearly the exact same pattern we discussed in the example of creating an account. After clicking the "New Todo" button, an HTML form will pop up that is associated with the URL for the Python function 'add_todo.' Nested within this form are the text input fields, and date and time pickers, needed for a user to create a new list entry. We achieve the effect of keeping this form hidden from view until the button is clicked by using the `data-toggle` and `data-target` attributes as shown below in Figure 25:

```
    <!--New todo button-->
        <button type="button" class="btn btn-info btn-lg" data-
toggle="modal" data-target="#addtodo">New Todo</button>

    <!--New todo form-->
        <div id="addtodo" class="modal fade" role="dialog">
```

Figure 25: The "New Todo" button and form in HTML

Notice how we set the `data-toggle` attribute to `'modal'` and the `data-target` one to `'#addtodo'`. This attribute is the same as the ID given to the todo form. Thus, when the button is clicked, the form appears. The `'modal'` attribute value is also in line with the `'modal fade'` class given to the todo form, ensuring that the form fades into view upon the button click. Finally, the role given to the todo form is `'dialog'`. Therefore, the form appears on screen overtop all else, similar to how a dialog box works.

When the "Add new Todo" button within this form is clicked, the information from each input field is sent with the request to the 'add_todo' function. Then, within this function, we use SQL INSERT statements along with `request.form[…]` to add the information entered by the user into the database and redirect the browser back to the URL associated with the 'mainapp' function, and the mainapp.html page is rendered again with the new entry. Of special note here are the date and time pickers, whose JavaScript code is shown in Figure 26 on the next page:

```
    $(document).ready(function() {
        $('input[name="time"]').timepicker({
            'scrollDefault' : 'now',
            'step' : 2
        });
        $('input[name="date"]').datepicker({
            'autoclose': true,
            'todayHighlight': true,
            'calendarWeeks': true,
        })
})
```

Figure 26: JavaScript code for date and time pickers

Writing the date and time pickers using JavaScript allows them to be *dynamic* in nature. That is, when the user clicks on the date or time field, these pickers appear and display behavior that is influenced by the time at which they clicked on it. For example, the `‘scrollDefault’: ‘now’,` line allows for the time picker scroll to start at the current time (or one minute away, if the current time ends in an odd number) in accordance from when the picker was clicked on.

E. ACCESSING ENTRY INFORMATION

Recall that, on screen, all that is shown for each list entry is the entry name and its associated category. To see the description, date, or time, you must click within the blue rectangle that forms the background of the entry to reveal a drop-down menu containing this information. We accomplish this effect by placing the entry within an HTML `<div>` tag with the class attribute ‘collapse.’ This attribute, possible through Bootstrap, makes it so that the panel below the blue rectangle appears collapsed (invisible to the user) until clicked on. Likewise, while this panel of information is open, the user can click on it to collapse it, closing it once again.

F. DELETING AND EDITING LIST ITEMS

Deleting items from the list follows the standard form and request protocol that we have established. With the button for deleting list items nested inside an HTML form associated with the URL for the Python function ‘delete_entry,’ the ID of the entry is passed along as input with the request. Within Python, we use an SQL DELETE statement to remove the entry containing that ID from the database. We then redirect the browser back to the URL associated with the ‘mainapp’ function, rendering the ‘mainapp.html’ page with the deleted entry gone.

Clicking on the edit button will bring up a form with text input fields where the user can type in the information that will be changed within the list item they have chosen. We achieve the effect of keeping this form hidden from view in exactly the same way the “New Todo” button works in HTML. Once the form is summoned, we follow the exact same protocol used for sending form requests we have previously discussed. The submission button for this set of inputs is nested within an HTML form associated with the URL for ‘edit_entry’ In Python, the “edit_entry” function is run with the sent request as the browser is directed to this URL, using SQL UPDATE statements to update the information in the database, replacing it with the information sent in the request. We then redirect the browser back to the URL associated with the ‘mainapp’ function, rendering the ‘mainapp.html’ page with the edited entry being shown on screen.

G. FILTERING BY CATEGORY

Within the ‘mainapp.html’ page is code for displaying the panel on the far-left side of the page containing the list of categories for all entries currently in the list. This code is shown on the next page in Figure 27:

```
<!--Sidebar with category-->
    <div class="col-md-2 col-sm-3 sidebar-nav">
        <ul class="nav category">
            <li><a href="?category=all"><span class="glyphicon glyphicon-
triangle-right"></span>Show all</a></li>

            {% for entry in category %}
            <li><a href="?category={{ entry.taskcategory }}"><span
class="glyphicon glyphicon-tag"></span>{{ entry.taskcategory }}</a></li>
            {% endfor %}
        </ul>
    </div>
```

Figure 27: The HTML code for the category sidebar

Notice in this code how each category rendered is an active link. By setting the `href` attribute within the link to `"?category=[either 'all' or {{ entry.taskcategory }}]`, we attach this to the end of the URL that the browser is directed to when a request is made using the link. Using `request.args['category']` within our Python mainapp function allows us to retrieve what the category is currently equal to. With this information, we can do a comparison to determine if the user wants only entries of a certain category to be displayed on screen. This comparison can be seen in the Python code of Figure 28 below:

```
    """Fetch data from database"""
            if "category" in request.args and request.args["category"] !=
"all":
                cur = db.execute(
                'SELECT id, taskname, taskdes, tasktime, taskcategory FROM
todos WHERE username = ? AND taskcategory = ? ORDER BY id DESC',
            [session['logged_in'], request.args["category"]])
            data = cur.fetchall()
```

Figure 28: The code for making a category comparison in Python

Notice the use of `request.args['category']` in this code to access the current category information. If the category is set to anything other than "all" (which is the case when a user clicks on the "Show all" link), then we select only the entries within that database pertaining to the category value for rendering on screen. Otherwise, all entries are selected. If you examine the 'mainapp' function even further, you will notice that, when fetching the categories to show on the sidebar, we use the SELECT DISTINCT SQL statement. This is because many entries on the list could have the same category. In this way, we avoid repeating categories on the sidebar.

H. MARKING LIST ITEMS AS "COMPLETE"

Recall the checkboxes to the far left of each entry in the to-do list. As stated earlier, checking this box signifies to the program that that item is complete and should be moved from the "Current" list to the "Completed" list. Alternatively, unchecking this box will move the entry in the opposite direction, back up to the "Current" list. Much like the delete buttons discussed earlier, the checkboxes are nested inside a form associated with a URL that calls for a Python function to execute when that URL is accessed. When the checkbox is checked, a request is sent along with the ID of that entry. In the case of checking items, that directs the browser to the URL for the Python ‘check’ function, and it is executed. With this information, we then use a combination of SQL INSERT and DELETE statements to delete the checked entry from the ‘todos’ database table, which holds all currently active list items, and insert it into the ‘completed’ database table, which holds all completed list items. We then redirect the browser back to the ‘/main’ URL, causing the ‘mainapp’ function to be run. The ‘mainapp.html’ page will then be rendered, with the entry appearing in the ‘completed’ list as it is now fetched and sent from the ‘completed’ database table rather than the ‘todos’ one. Once an item has been moved to the “Completed” list, it can still be deleted, but cannot be edited. Deletion is carried out in exactly the same way that items are deleted from the the “Current” list. By clicking on the entry, a drop-down menu will be revealed containing a delete button nested inside an HTML form associated with the URL for the ‘delete_complete’ Python function. Once the button is clicked, deletion proceeds as outlined in section F.

Likewise, the checkboxes attached to entries within the “Completed” list are embedded within a form associated with the URL for the Python ‘uncheck’ function. When a checkbox from this list is clicked, the ID for that entry is sent along with the request. The browser is directed to the URL for the ‘uncheck’ function. The function then uses the ID of the entry and SQL INSERT and DELETE statements to delete the entry from the ‘completed’ table and insert it into the ‘todos’ table. We then redirect the browser back to the ‘/main’ URL, causing the ‘mainapp’ function to be run. The ‘mainapp.html’ page will then be rendered, with the entry appearing in the ‘todo’ list as it is now fetched and sent from the ‘todos’ database table rather than the ‘completed’ one. Through the use of `<div>` tags and CSS, we apply different colors to the entries based on which part of the page they appear in. Thus, the entries can change from blue in the "Current" list to green in the "Completed" list and vice-versa as discussed earlier.

I. SEARCHING FOR LIST ITEMS

Recall the "Search for anything…" text input field as discussed earlier. This field and its use follow the standard protocol for sending a request and receiving information that we have discussed. Within the HTML of the 'layout.html' file, you will find this input field and its associated button nested within a form associated with the URL for the 'search' Python function. Once the button is clicked, the request, along with the inputted information, is sent to the server. The browser is directed to the URL for the search function, and the information entered is then used as the basis for four different SQL SELECT statements. These statements incorporate the WHERE and LIKE conditions so that we may search the 'todos' table for entries where the parts of each entry are like the inputted information. To visualize this more clearly, the code for this function is shown in Figure 29 on the next page:

```
    def search():
        db = get_db()
        cur1 = db.execute('SELECT id, taskname, taskdes, tasktime, 
taskcategory FROM todos WHERE taskname LIKE (?)',
                              ["%" + request.form['search'] + "%"])
        cur2 = db.execute('SELECT id, taskname, taskdes, tasktime, 
taskcategory FROM todos WHERE taskdes LIKE (?)',
                              ["%" + request.form['search'] + "%"])
        cur3 = db.execute('SELECT id, taskname, taskdes, tasktime, 
taskcategory FROM todos WHERE tasktime LIKE (?)',
                              ["%" + request.form['search'] + "%"])
        cur4 = db.execute('SELECT id, taskname, taskdes, tasktime, 
taskcategory FROM todos WHERE taskcategory LIKE (?)',
                              ["%" + request.form['search'] + "%"])
        data1 = cur1.fetchall()
        data2 = cur2.fetchall()
        data3 = cur3.fetchall()
        data4 = cur4.fetchall()
        data = data1 + data2 + data3 + data4
        data = set(data)
        user = session['logged_in']
        return render_template('search.html', result=data, user=user, 
keyword=request.form['search'])
```

Figure 29: The code for the Python 'search' function

As you can see, after comparing each aspect of a list item with the input, we build a set of results and pass it to the 'search.html' page. The 'search.html' page will then be rendered in the browser, displaying the entries on screen with the same styling and in the same format as they are on the 'mainapp.html' page. Note how we also pass the original search input back to this page through the 'keyword' variable. This is for rendering purposes in the case of no results being found as detailed in the code for the 'search.html' page.

J. E-MAIL BASED REMINDERS

The ability to produce e-mail based reminders for tasks requires the use of an entirely new Flask module: APScheduler. In order to use it, we must set up a Config class as displayed below in Figure 30:

```
  class Config(object):
      JOBS = [
         {
        'id': 'job1',
        'func': 'todo:job1',
        'args': None,
        'trigger': 'interval',
        'minutes': 1,
        'timezone': 'America/Chicago'
   }
]

SCHEDULER_API_ENABLED = True
```

Figure 30: The Config class as needed for APScheduler fuctionality

Within this class, a new job is created with ID of ‘job1,’ This job has the responsibility of running the function named ‘job1’ within our ‘todo.py’ script. We are passing no arguments to this function, and the running of the function will be triggered on an interval (periodically). In other words, this function will be executed once every minute from when the program begins running. As our team is based out of Illinois, we have chosen to measure the time at which the function is run within the Central time zone, as it is in Chicago.

We must now define the ‘job1’ function, as shown in Figure 31 on the next page:

```
  def job1():
  with app.app_context():
      db = get_db()
      cur = db.execute('SELECT tasktime FROM todos')
      items = cur.fetchall()
      now = datetime.now().replace(second=0, microsecond=0)

      for time in items:
          compare = datetime.strptime(time[0], '%I:%M%p %m/%d/%Y')
          if now == compare:
              curl = db.execute('SELECT taskname, username FROM todos WHERE
tasktime = ?', [time[0]])
              event = curl.fetchall()
              cur2 = db.execute('SELECT email FROM profile WHERE username =
?', [event[0][1]])
              address = cur2.fetchall()
```

Figure 31: The ‘job1’ function code sans e-mail building request

Note that this function will scan the database every minute, comparing the current time as obtained through datetime.now() with the date and time attached to each list entry in the ‘todos’ table. The purpose of the date and time pickers attached to these respective fields when a user adds a new item to the list is now made apparent, as they work to format the data in a certain way. Specifically, the time is given in “hours:minutes, (AM or PM),” while the date is given in

"MM/DD/YYYY" format. In formatting the task date and times in this way, they can be passed to the 'strptime' method of the Python 'datetime' class to be converted into 'datetime' objects. With datetime.now() and the new 'datetime' object assigned to the variable compare, we can now compare these objects to see if they are matching in each part of the date and time (year, month, day, hour, and minute). Assuming a match, then the name of the list item as well as the user's e-mail address are fetched from the 'todos' and 'profile' databases respectively and assigned to variables. The variables are then passed into the Mailgun API code, attached to this function, as discussed in section B, and an e-mail is sent warning users that the event is scheduled to happen. By scanning the database every minute, we ensure that no event will go without a reminder as minutes are the smallest unit of time compared between the two 'datetime' objects in this case. Note that to actually start the scheduler, we must use code to update the app so it will incorporate the settings as defined by 'JOBS' within the Config class. We thus initialize the scheduler using the code presented in Figure 32 below:

```
app.config.from_object(Config())

scheduler = APScheduler()
scheduler.init_app(app)
scheduler.start()
```

Figure 32: Code to start the APScheduler within our program

K. LOGGING OUT

The code for user logout follows the same protocol of sending a request to the server, causing a response to be sent back that we have discussed. Within the HTML code for the 'layout.html' page, you will find the "Logout" button imbedded within a form associated with the URL for the Python function ‘logout.’ When the button is clicked, a response is sent to the server, the browser is directed to the URL for the ‘logout’ function, and the function is executed. Within this function, we pop the current user as stored using the ‘logged_in’ key, from the session object, essentially decoupling the user from the session and logging them out. The browser is then redirected to the URL for the ‘index’ function, which causes the login screen to be rendered once again, ready for a new user.

L. CHANGING PROFILE SETTINGS

Recall that by clicking the “Your Profile” button, the user brings up a menu where they may change the name, e-mail address, or password associated with their account. The feature is particularly interesting because it combines HTML, Python, and JavaScript. The “Your Profile” button is nested within an HTML form within the ‘layout.html’ page that is associated with the URL for the Python function ‘profile.’ When the button is clicked, the function will execute as the browser is directed to this URL. The function simply calls for the browser to render the ‘profile.html’ page on screen. As we have seen, from the user perspective, this page contains a series of links that the user may click on to change whatever aspects of their account that they wish. The association of these HTML links with JavaScript is done through the ID attribute. We will look at an example below in Figure 33:

```
    <div id="show-email" class="panel-heading">Email: {{ entry.email }}</div>
    <div class="panel-body"><a href=# id="edit-email">Change email</a></div>
    <div id="info-email">
        <input class="form-control" type="text" value="{{ entry.email }}"
name="email">
    <div class="btn-group">
        <button id="conf-email" class="btn btn-primary">Confirm</button>
       <button id="cnl-email" class="btn btn-default">Cancel</button>
           </div>
        </div>
    </div>
    $('#edit-email').click(function(){
        $('#info-email').show();
       })

    $(function() {
        var change_info = function(change) {
            $.getJSON($SCRIPT_ROOT + '/edit_info',
        {
            name: $('input[name="name"]').val(),
            email: $('input[name="email"]').val(),
        },
            function(result) {
                $('[id^="info"]').hide();
                $('#show-name').text('Name:  ' + result.name);
                $('#show-email').text('Email:  ' + result.email);
                $('#alert').attr("class","alert alert-success")
                $('#alert').text("Your information has been updated")
            }
        );
        return false;
    };

        $('#conf-name').bind('click', change_info);
        $('#conf-email').bind('click', change_info);
        $('input[type=text]').on('keydown', function(e) {
            if (e.keyCode == 13) {
                change_info(e);
        }
    });

});
```

Figure 33: HTML and JavaScript code for e-mail changes

Observing this figure, there are several key points to notice. To start, notice how the link established in the very first line of HTML code is given the ID ‘edit_email.’ Then, the middle snippet of JavaScript is a function stating that, when this element with ID ‘edit_email’ is clicked, everything inside the `<div>` that follows with the ID ‘info-email’ will be shown. This *dynamic* design is achieved through the connection of HTML and JavaScript by element IDs. There also exists JavaScript code in this same format for the “Cancel” button shown in the HTML above. However, when that button is clicked, the information inside the `<div>` will be hidden. The fields for name and password changes are all accessed and hidden through these same methods. Additionally, notice the line within the third portion of JavaScript code that is highlighted in gray. By ID association, when the button with the ID 'conf-email' shown in the HTML above is clicked, the function 'change_info will run, and by extension, through the use of `.bind`, the Python function associated with the URL '/edit_info' will be run. This code uses `request.args['email']` to obtain the inputted email information sent with the request when the 'Confirm' button was clicked. It will then update this information in the 'profile' database table through an SQL UPDATE statement and return the obtained email information as JSON The JavaScript code, through the function 'result,' then outputs the updated information to the HTML `<div>` with ID 'show-email' utilizing the JSON to do so. It also alerts the user that the information was successfully updated. Use of JavaScript in this way allows for the page to change dynamically, as the current e-mail shown on screen will be changed right away to reflect changes within the database. While this example only details how the functionality of changing e-mails works, the functionality to change the name works through code that is formatted in the exact same way, only with different ID names for the respective components. The changing password functionality is slightly different in that the HTML is altered so as to not display the current password on screen, and the JavaScript and Python code take advantage of the three input fields provided when a user changes their password, as shown in Figure 16, by comparing the new password entered with the conformation password. The code is set up so that the password will be updated only if these two fields are determined to match. In this way, we can guard against typos in passwords, assuring that the user has the password they truly want.

3. IMPLEMENTATION OF REMAINING USER STORIES

A. GROUP

There were two user stories that our group failed to reach within the eight-week long project period: “Group” and “Message.” The goal of “Group” was to allow users to create groups within our application. By entering the username and e-mail of accounts that one wants to add to their group, users could collaborate on one to-do list. Managing multiple accounts and inputs from those accounts could prove to be quite tricky. In this case, one might use AJAX to update the list dynamically as user inputs are taken in by the application. In addition, you would want to find a way to associate each list entry with the group that works on it. To this end, you might attach group IDs or group names to each entry when it is entered into the database. You would need to research how to get this database information to persist across multiple computers on the same network. Websockets are one solution, but can be difficult to implement correctly. You would need to conduct further research on the use of websockets in Python.

B. MESSAGE

This user story aims to allow users that are working in a group to send messages to each other, similar to the chat function you may find on social networking sites like Facebook. Once again, you might turn to AJAX to update the page dynamically, pulling in new messages as they are received by the server. How you would get this chat to run in real time, persisting across multiple machines on the same network may be difficult, requiring further research into the use of AJAX or other solutions. As of now, AJAX is the only solution that I am currently aware of. Even then, I do not know how it would be implemented in this case.