**Sprint Self Evaluation of User Stories and Contributions**

**Programmer: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Larissa Ford\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**How to earn grades for the Sprint:**

|  |  |  |
| --- | --- | --- |
| Requirements | User Stories | 10 |
| Confirmations | 20 |
| Design | UML, ER Diagram, Software Architecture | 30 |
| Data Structures and Algorithms | 20 |
| Implementation | Code Structure and Readability | 30 |
| Documentation | 20 |
| Testing | Unit, System, and Acceptance (Confirmation) | 50 |
| Significance | Challenging Tasks | 20 |
| **Total Coding** | | **200** |

Contents

[**User Story:** As a Product Owner, I want to have a database so that I can save user data. 3](#_Toc56779926)

[**Confirmation** 3](#_Toc56779927)

[**Design** 4](#_Toc56779928)

[**Implementation** 4](#_Toc56779929)

[1. Create the app 5](#_Toc56779930)

[2. Add the app to Settings 5](#_Toc56779931)

[3. Make the model (according to the ER diagram) 5](#_Toc56779932)

[4. Make the database aware of the changes you made to the model 6](#_Toc56779933)

[5. Add the model to the admin page 6](#_Toc56779934)

[**User Story:** As a user, I want to have a home page where all my widgets are contained and my background image is displayed. 7](#_Toc56779935)

[**Confirmation** 7](#_Toc56779936)

[**Design** 7](#_Toc56779937)

[**Implementation** 8](#_Toc56779938)

[1. Create the app 8](#_Toc56779939)

[2. Add the app to Settings 8](#_Toc56779940)

[3. Make the templates Folder (this is Joe Sedivy’s work) 8](#_Toc56779941)

[4. Modify the home.html page in the templates folder 9](#_Toc56779942)

[5. Modify Home/views.py 9](#_Toc56779943)

[6. Modify Django’s URLs 10](#_Toc56779944)

[**User Story:** As a user, I want to be able to pixelate an image, so that this changes my background image. 11](#_Toc56779945)

[**Confirmation** 11](#_Toc56779946)

[**Design** 11](#_Toc56779947)

[**Implementation** 12](#_Toc56779948)

[Create URL path for Pixelating an Image 12](#_Toc56779949)

[Pixelating Functions 13](#_Toc56779950)

[Creating the View 15](#_Toc56779951)

[Connecting it to the Pixelating Image Button 15](#_Toc56779952)

[**User Story:** As a user, I want to be able to download the image to my computer so that I can access the picture without accessing the website. 16](#_Toc56779953)

[Confirmation 16](#_Toc56779954)

[Design 16](#_Toc56779955)

[Implementation 16](#_Toc56779956)

[**Design** 17](#_Toc56779957)

[**Testing** 19](#_Toc56779958)

[**Significance of code** 20](#_Toc56779959)

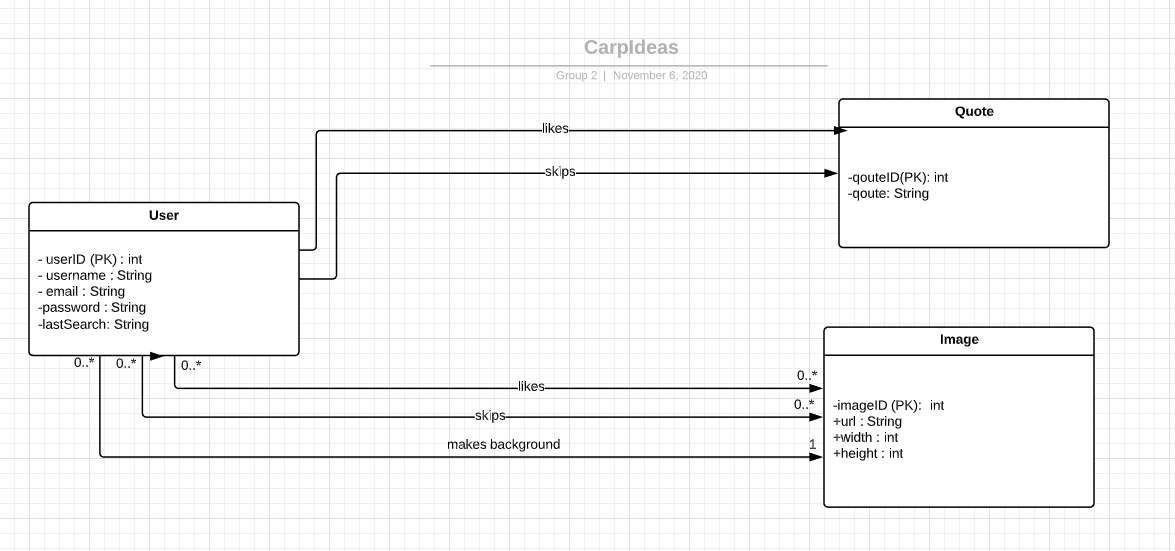
# **User Story:** As a Product Owner, I want to have a database so that I can save user data.

## **Confirmation**

|  |
| --- |
| The software developer will create an ER Diagram of the database. |
| The agile team shall go over the ER diagram |
| The software developer shall code the database |
| The software developer shall fill the database with test data. |
| The software developer shall create a series of tests. |
| The software developer shall debug the database |
| All test shall pass |
| learning and install time |

For this user story, we worked on it as a group. For my part, I helped design the ER diagram and helped create the database using Django. Joe Sedivy implemented the User class, and I helped in implementing the Image class into our database. Tracen Vail is working on the many-to-many relationships, such as adding liked and disliked images to the database.

## **Design**



Django uses a complicated structure of code that helps to make the code simpler in the long run but has its initial steep learning curve. Specifically, to implement a class for a database like Image, you need to have a virtual environment set up with Django installed, an “app” created and what goes along with that, the app’s model modified, and then its admin file modified so that you can easily add items to the database (which is useful for functional testing). The “app” is kind of like an abstract structure that Django uses for creating and filling the database, and also making it viewable from the server. Working with it, Django hopes to streamline coding a bit, getting rid of excessive boiler plate code. In general, it aims to make code more readable.

The steps are as follows:

1. Create the app
2. Add the app to settings
3. Make the model (according to the ER diagram)
4. Make the database aware of the changes you made to the model
5. Add the model to the admin page.

## **Implementation**

(Show some of your interesting code to demonstrate structure and readability. Also, include some code showing documentation.)

### Create the app

To start implementing the Image class from the ER diagram, you first need to make what Django calls an “app”. To do this, you need to go into your virtual environment (which I have named venv) through your terminal and go to where manage.py is located (which is src/carpideas for me). “manage.py” is kind of like Django’s way of talking to the server, like typing “npm start” or “npm run dev” with Node.js. Once in that directory and inside your virtual environment where Django was installed, you type:



This creates Django’s building blocks for adding things to the database and making it viewable from the server.

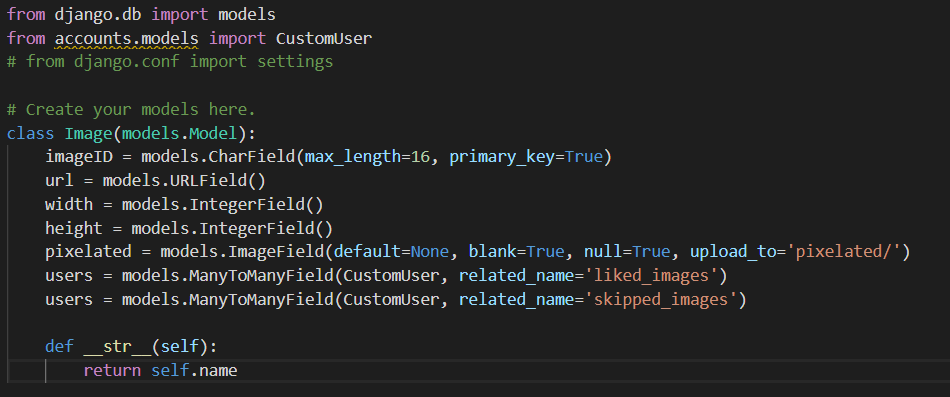
### Add the app to Settings

To make it official, you also have to go into your Django project’s settings file and add your custom app to the installed apps list.



### Make the model (according to the ER diagram)

To finish adding the Image class to the database, you need to modify the Image app’s model inside Image/models.py:



### Make the database aware of the changes you made to the model

Finally, to make your database aware of your newly made model, you type this in your virtual environment where manage.py is located:



### Add the model to the admin page

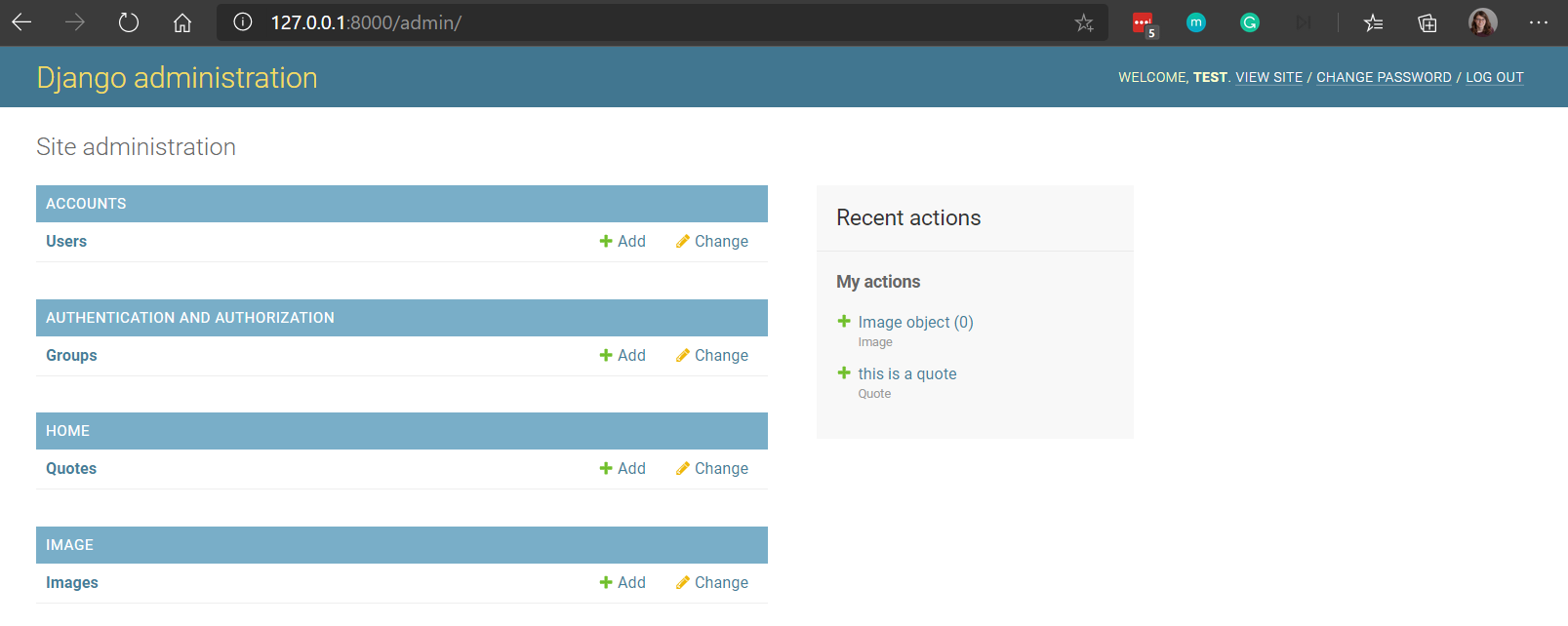
With this, the Image class has been created according to the Django way. As an extra but very useful step, you would also go into the admin.py to be able to display our app inside the admin page.



Now, after running the server using the command “python manage.py runserver”,



you can view the admin page like this:



# **User Story:** As a user, I want to have a home page where all my widgets are contained and my background image is displayed.

## **Confirmation**

|  |
| --- |
| create app in Django to house the home page |
| link the home page with the rest of the website through Django |
| Make a textbox with a welcome message. |

## **Design**

*(Include all design work for your user stories. Include description for all data structures and algorithms used with discussion on why you selected them):*

Django’s steep learning curve is especially prevalent when making a simple html page show up. After having the virtual environment set up properly, to implement an html page, you need to set up the view, the urls, templates, and the Django settings. The first two steps are the same as for adding to the database, but now we need to modify the app’s view, Django’s urls, and add templates for Django to access.

1. Create the app
2. Add the app to settings
3. Make the template folder
4. Modify the home.html page in the templates folder

## **Implementation**

*(Show some of your interesting code to demonstrate structure and readability. Also, include some code showing documentation.)*

### Create the app

Starting with the settings, you need to add a new app that you will eventually get to display on the server:



### Add the app to Settings

To make it official, you also have to go into your Django project’s settings file and add your custom app to the installed apps list.



### Make the templates Folder (this is Joe Sedivy’s work)

Apparently, by default, Django will look in each app for a templates folder where it should find the app’s HTML files. We did something similar but instead made a templates directory accessible from all the apps. To do this, we modify the Settings again. (This part was Joe Sedivy’s work.) Then, inside our project directory, So, first, you make the directory in the Home app called “templates”, and then you make the file called “home.html”.



### Modify the home.html page in the templates folder

What our home.html now looks like using Django infrastructure:

**

It is taking the html page layout from a base page we made in the same directory.

The base.html page was heavily reworked by Savannah Ford to include bootstrap and a lot of what we called “widgets” that you can see from our home page. These widgets include the analog clock, the quote of the day, a Spotify music player, the weather widget, and a digital clock. It also is linked to the CSS page where there is more than 200 lines of code.

Also, as you can see, it is getting something called “image” and “request.user”. This is being given to it from the view.py from within our home app. It is being passed as context from there.

### Modify Home/views.py

Here, a view is created to be used in the URL for our server. This view in particular is doing quite a bit. It is taking from a global class which I wrote for getting and setting and is initialized with a function that was made by Seth Schalinske. This image is passed as context to the html page as ‘image’ (and used like what was seen in the previous section).



### Modify Django’s URLs

Lastly, to be able to use this view, we need to modify Django’s URLs.



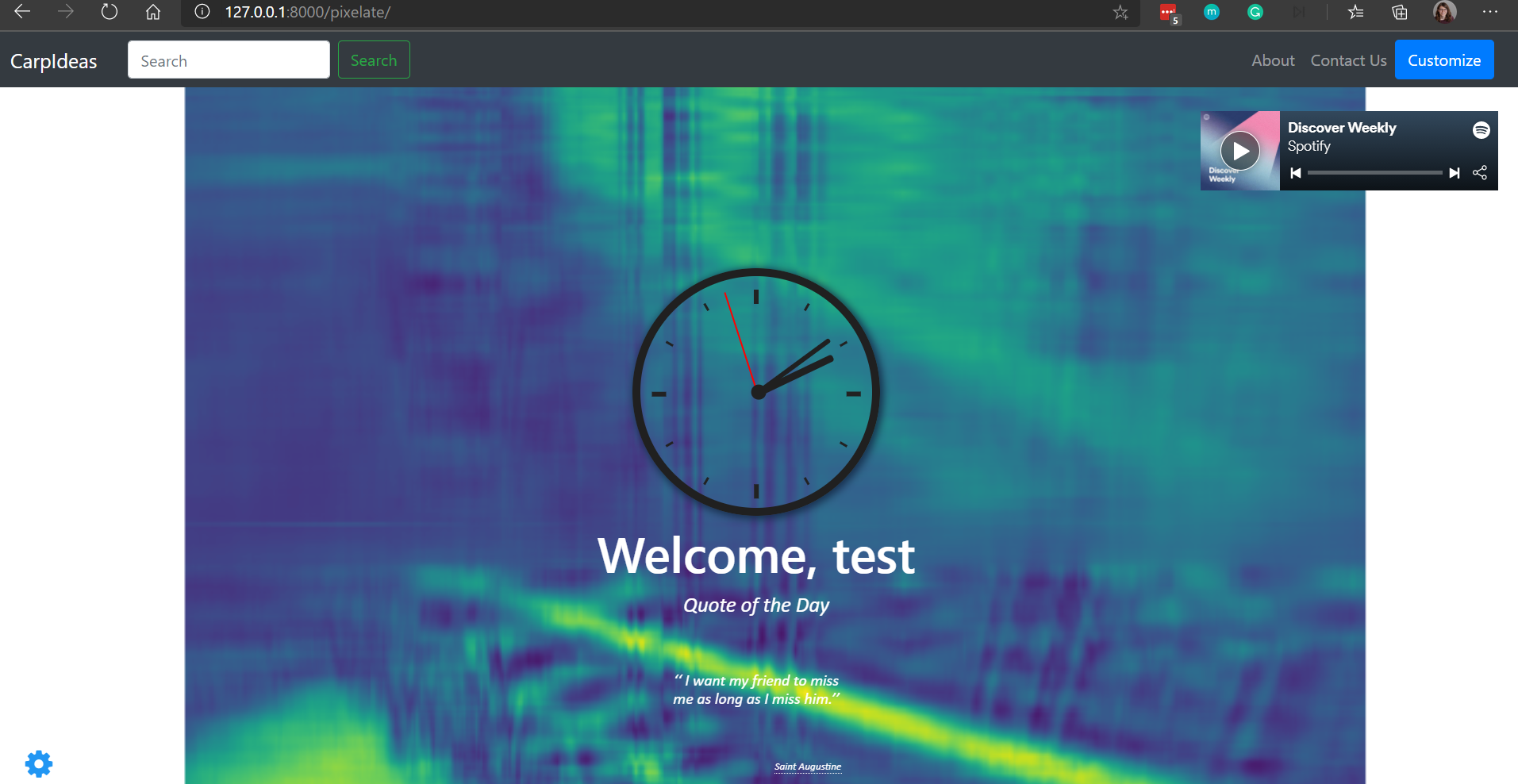
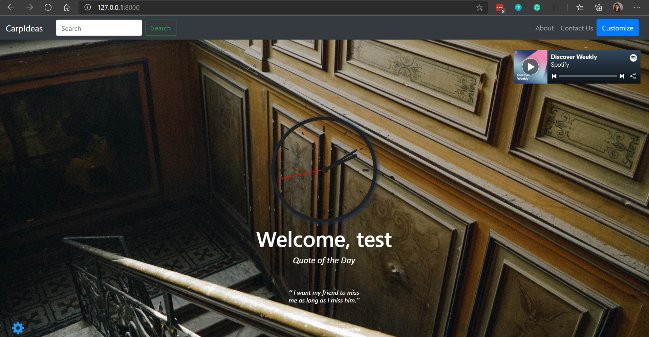
# **User Story:** As a user, I want to be able to pixelate an image, so that this changes my background image.

## **Confirmation**

|  |
| --- |
| create a pixelate button to pixelate the current background |
| when pixelate button is clicked, a drop down menu displays pixelation options (how much pixelation they want) |
| use numerical linear algebra to pixelate the image |

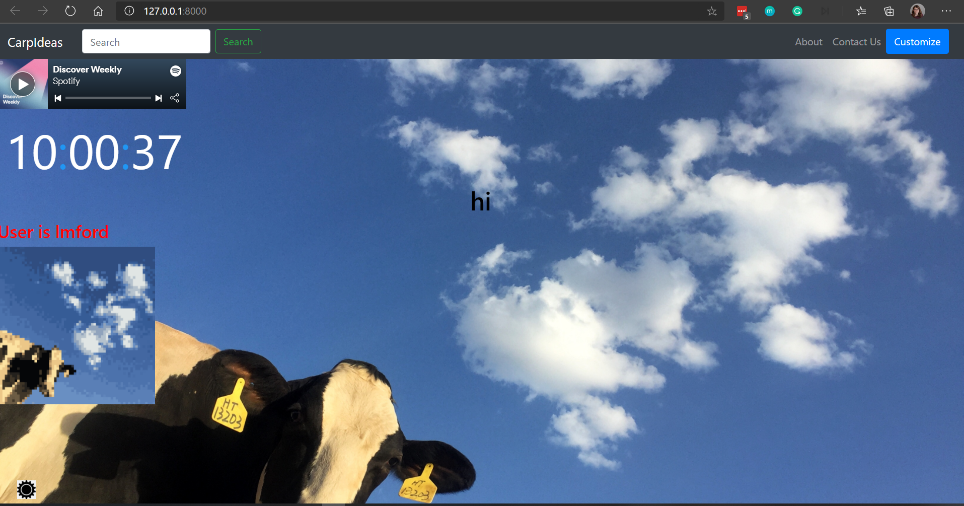
## **Design**

At first, I had implemented this using Numerical Linear Algebra. Then, my team and I realized that this homebrew pixelization function gave a rather ugly result, giving this pixelation:



After that, I decided to implement code from GitHub called PyPXL (found here: <https://git.lattuga.net/goat/pypxl>)

Which gives me this pixelization (shown in the small box under the red letters):



I have since then modified the code to make the pixelization take up the whole screen, like so:



Here is a short video showing how the pixelating Image button works.



## **Implementation**

### Create URL path for Pixelating an Image

To do this, I had to follow a similar procedure to making the home app, but this time I did not make a new app to do this. Instead, I added another view to the Home/views.py and put the reference to that into a url:



### Pixelating Functions

Then, for the new view, I made a series of functions for pixelating the image.

First, I will show my old pixelating function:



Now, here is my current pixelating function that uses the code from github and a couple other packages, such as Scikit-Image for converting a URL into a PNG file. Also, to use the PyPXL code, I had to learn about subprocesses, which is pythons way to do multithreading.



This function saves the pixelated image to a file that can be accessed. Here is a short function for accessing that image from the file:



### Creating the View

To call either of these functions, I need a view that will be called from the URL. Here is the view that can be modified to call either of the two functions. I also made it so that it can just grab the pixelated image from the file because the pixelating function is a bit heavy, and takes usually at least a minute to run.



### Connecting it to the Pixelating Image Button

Here is a button that is hooked up to the URL:



# **User Story:** As a user, I want to be able to download the image to my computer so that I can access the picture without accessing the website.

## Confirmation

|  |
| --- |
| create a download button on the user's home page |
| test whether download button downloads on click and to where |

## Design

Currently, it uses the same structure as the pixelate button, bringing you to another html page where the image download would happen. This means I made an html page for the downloading image, a url to link to it, and a button to click that brings you to the URL from the home page. In the future, I’d like to rework this so that the download happens in home.html and the button click starts the download.

Here is a small video showing how it currently works:



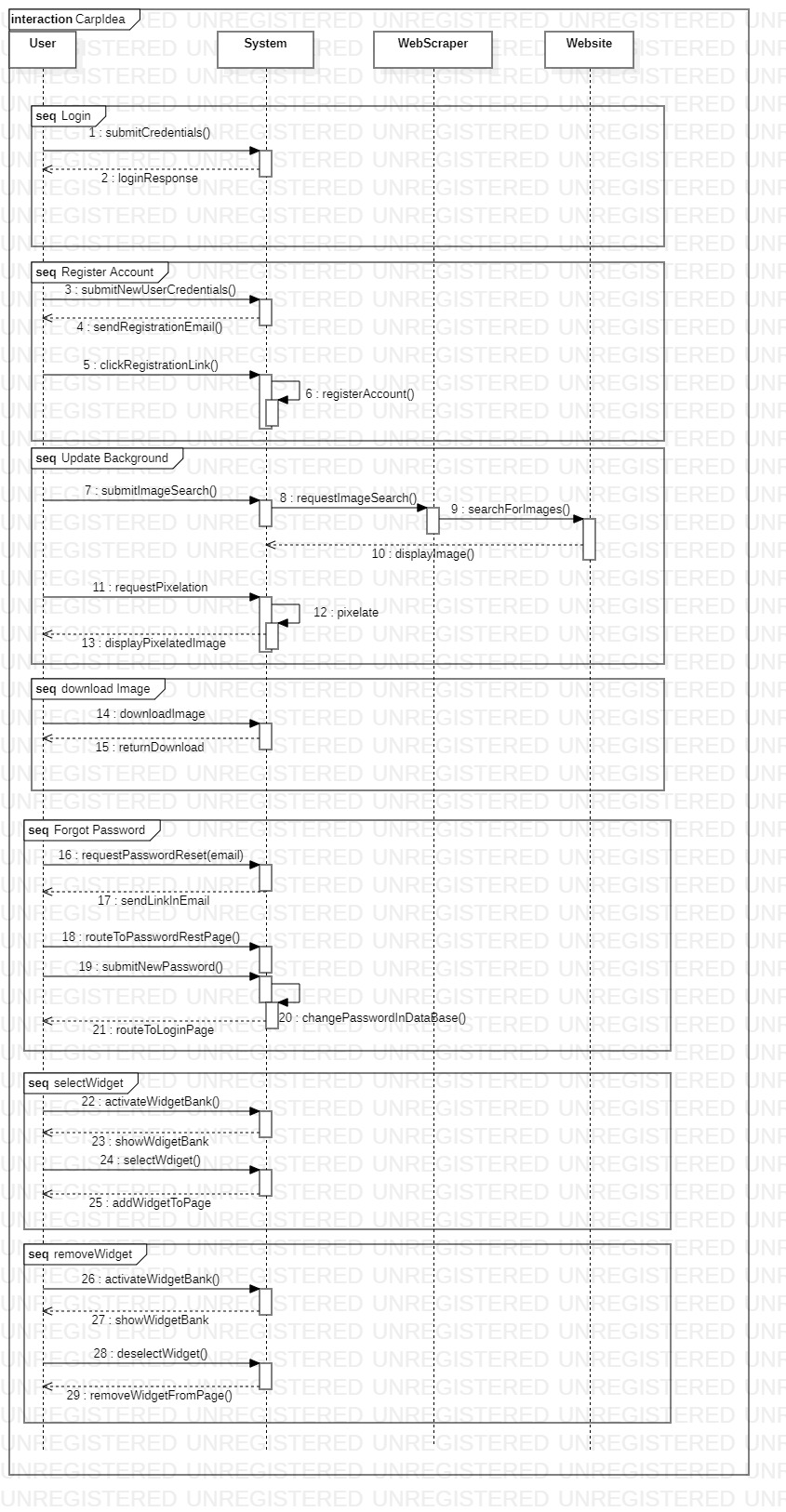
## Implementation

Withholding code that looks nearly identical to the pixelating Image setup (like the url path, the html page, and the button code), I will show you what is different: the download\_view from home/views.py

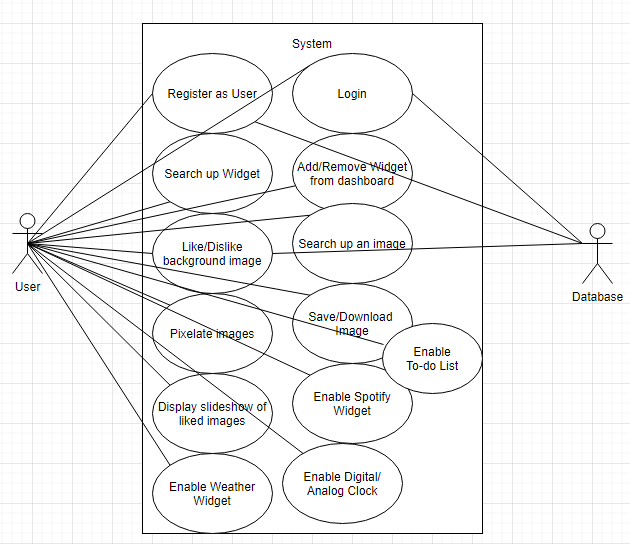


# **Design**

Besides our ER diagram that was shown above, here is a sequence diagram displaying pixelating an Image and Downloading an Image:



Here is also a use case diagram that shows similarly:



# **Testing**

*(List all unit testing, system testing, and acceptance testing implement. List a testing plan to demonstrate that your code works)*

We have generally done functional testing and have implemented this by using the administration page that comes with Django. With it, we can add test objects to the database to see if our server interacts with them in the way we want it to. We can also use the administration page to make sure objects that that client adds by way of the html pages are added to the database. Besides this, Django apparently has its own way of testing itself, but we have not had time to research this and implement it. Learning and implementing that is on our list of things to get done.

# **Significance of code**

*(Describe challenges overcome in implementing your code. Describe its significance to the project)*

In general, Django has a pretty steep learning curve that we are constantly running into. For instance, working with the database has a lot of layers of abstraction over it, which, coming from MySQL Workbench, is quite different. This issue was especially prevalent when trying to pass functions to button clicks, which I still am not sure if I am doing correctly.

Besides this, For my pixelating function, I was constantly running into roadblocks involved with the versions of the packages I was using, the fact that I was initially using a 32-bit Python which means I was running into memory errors, and packages generally not wanting to run. I ran into such issues because I am using a GitHub project, PyPXL (found here: <https://git.lattuga.net/goat/pypxl>), and that means I didn’t quite know how it was using its packages and which ones it really needed. This took a lot of digging and researching. After upgrading to a 64-bit version of Python, which got rid of my memory issues, I learned that not all of my packages work with Python 3.9, so I had to downgrade back to 3.8 but this time get a 64-bit version. Working with multithreading alongside this was also very difficult.

Our virtual environment also often does not have the right packages after pulling from our Git Repository because, especially myself, we are often adding more modules to make our program run the way we want it to. Because of this, I have to update our requirements.txt a lot using this command:

pip freeze > requirements.txt

My last greatest challenge was using Django and GitHub together. Apparently, a lot of things should be ignored by GitHub within Django, but we didn’t know this initially so I had to figure out how to update the .gitignore page after people had already made many commits that included stuff we don’t need. I now know how to do this, but I had to ask for help from someone in the industry who is used to using GitHub.

If applicable, describe any extra contributions to your team not listed above

I was often in charge of updating our readme.txt, .gitignore, and our requirements.txt. Besides this, I would often instruct my groupmates on how to set up their virtual environment and push and pull from GitHub while using Branches.