Feature: Picture Upload

Feature Developer: David DeGirolamo
Date Developer Submitted: 04/11/23

Review by: Joseph Armas
Date Review Completed: 04/15/23

Major Positives

1. Database design. The ***Positive*** about this is that he created a separate table called dbo.Extensions and by doing this simulates a ***look up table*** which I think is a good additional layer of validating an extension of a file before inserting into the database.

2. Loosely implement some sort of validation in the ***Service layer*** after grabbing data from the ***Data Store.*** Implementing this check ensures the data grabbed from the ***Data Store*** is the correct value that needs to pass to the ***front end.***

3. Validations in the ***Front End*** when checking *UserID* and *pinID*. This will ensure that a pin exists and that the user is authorized to request it in the backend. In addition, he also implemented a check in the front end when uploading a picture. This will ensure the system will check for file size and extension before posting into the backend.

4. Validation in the **SecurityManager** . This ensures that if the user wants to use this feature, the system will check and proceed based on the user type. It protects the system and restricts users' access to what they are allowed to do.

5. For the ***UIDiagram.jpg*** view, I think it is great he implemented showing the picture when a user that is not the pin owner clicks on it and shows a preview of the uploaded picture. This allows other users to have a glimpse of how the area looks after reading the title and description of a pin.

Major Negatives

1. The ***low level back end design*** and ***front end views*** are missing for: **Delete an image a user posted** and **View uploaded pictures (See Design Recommendation 7 & 8)**

2. In Reference to ***UploadFileFail.png*** validation checks are only present in the front end.
   This is an issue because as said in class, the front end is not secure. Therefore, there needs to be some kind of validation checks in the backend. (***See Recommendation 9***)

3. There is no indication of a precondition in *all* the *low level diagrams* . This causes an issue because developers who do not have any knowledge on our system will be confused as to when this feature can be used. Does a user have to have an authenticated session?

4. In all overall diagrams the arrows when calling on methods from ***SecurityManger*** to ***Database*** are all synchronous and do not have a **return data type**. This can be a huge issue because all of these methods would have to wait on each other before proceeding. This will cause breakups in the systems and reach deadlocks. As a result, the user experience in the front end will not be pleasant because the user will be obstructed from using other resources. ***(See recommendation 5)***

5. In reference to ***FileUpload.png*** The ***SecurityManager, FileService, SqlDAO*** have the same method name ***UploadFile()***. Having the same name method in each of these layers can lead to confusion and also make it difficult for a developer to differentiate the purpose of each method on each of these layers. ***(See recommendation 11)***

6. The User Story titles do not match or are the same. This makes it difficult to interpret what should the system be doing and what should be displayed to the user in the front end.

Unmet Requirements

1. As an owner of an image, I am able to delete an image

This means that the image will stay on the pin or profile pic. The modification or customizable preference is now gone. What if a user accidentally posted a picture that is not supposed to be seen? Or suppose that they want to remove the picture because they do not want it to be shown anymore. **(See Design Recommendation 8)**

2. As a user, I am able to view my own uploaded images

This is crucial for end users who like to see the trackable history of the photos they have uploaded. In essence, this can also show the participation a user is with interacting with the upload picture feature. Moreover, a user is able to see the distinct comparison between before and after pictures. **(See Design Recommendation 7)**

3. A User is able to upload an image to a pin that is not theirs (See Design Recommendation 2a)

4. Low Level Diagram Success/ Failure for an Admin User

a. **Success**:
   i. Delete at least 1 or more user's images **(See Design Recommendation 10)**
   ii. View 100 images with 7 seconds of loading for each **(See Design Recommendation 3)**

b. **Failures**:
   i. Unable to delete images
   ii. Image takes more than 7 seconds to load (**See Design Recommendation 3)**
   iii. System slows due to too many images **(See Design Recommendation 3)**

This is crucial to implement, because this allows users in the admin role to have control over any inappropriate content that is uploaded on Utification. Moreover, it allows extensibility by having *Regular, Service or Reputable users* able to contact *Admin users* to fix any issues on retrieving or posting any images.

Design Recommendations

1. It's not a major issue, but I would consider **changing** the ***SecurityManager*** to possibly something like ***FileManager***. The reason why I suggest this allows other developers to easily indicate where the issue is coming from when one arises, rather than trying to search the source in the SecurityManager class that isn't related to files. Moreover, It keeps it more consistent when you have already named the service class ***FileService.***

2. In reference to ***FileUpload.png,*** Before uploading a picture to AWS S3 bucket, I would consider having another **check statement** when the data is received from the backend. Solely relying on a 200 response to post an image can possibly lead to a vulnerability by circumventing the backend and just mimicking a 200 response. Maybe consider ***checking the filename and extension*** before posting to the AWS S3 bucket.
    a. Inside the method IsPinOwner() is perfect to resolve this issue. He did not have a set diagram returning the error message "**Posting Issue. Invalid Action Performed.**" to the front end.

3. I would recommend having a **timer** inside of ***fileUpload.js*** this will ensure each of the images are properly being loaded in less than 7 seconds. Moreover, according to an admin failure case of the generic "***System slows down due to too many images***" since there is not a set metric on defining what is considered slow, having a timer in the front end will allow the system to have a reference to dictate what is slow. For example, if the timer is greater than 10 seconds then it can trigger the **BRD failure scenario** of the **system slowing down due to too many images**. For testing (**See Test Recommendation 3)**

4. When uploading a photo to a pin, I suggest **capping the length and width** of a photo to a reasonable size.
    a. **Positive**: This will possibly allow the system to ***easily render*** smaller images compared to largest framed images. Additionally possibly what can help is dynamically resizing images based on the "native" screen . This can improve the system performance in
    Based on screen size
    (https://stackoverflow.com/questions/37868027/dynamically-change-image-depending-on-screen-resolution)

Image resize css (https://cloudinary.com/guides/bulk-image-resize/bulk-image-resize-in-css-javascript-python-java-node-js-and-other-languages)

b. **Negative**: I'm not too familiar with how images work in the front end, however I do think there might be some type of ***loss in quality when resizing*** an image to a smaller size.

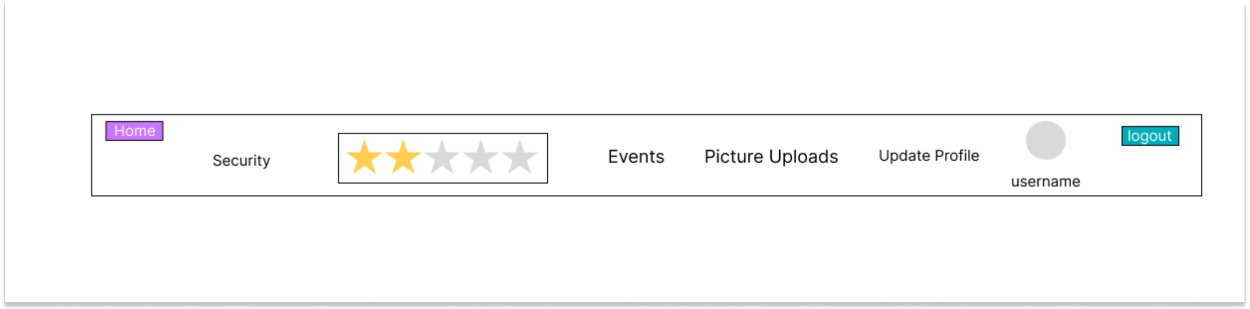
5. Since we decided to go fully asynchronous it should be dotted arrows from the entry point. Moreover, based on our current Source Code, it should be returning a Response obj.
    a. To indicate that the **method** is asynchronous it would like:
        i. Task<Response>DownFile(string filename, int userId, int pinID)
    b. A recommendation for returning a list of obj, example when "GET" from database is to use:
        i. Task<DataResponse>DownFile(string filename, int userId, int pinID)

    Allowing this will be easier in having this obj being passed around each layer to do any sort of validation, mapping or logging. In addition, when the obj reaches the front end, you're able to validate by parsing through the json obj.

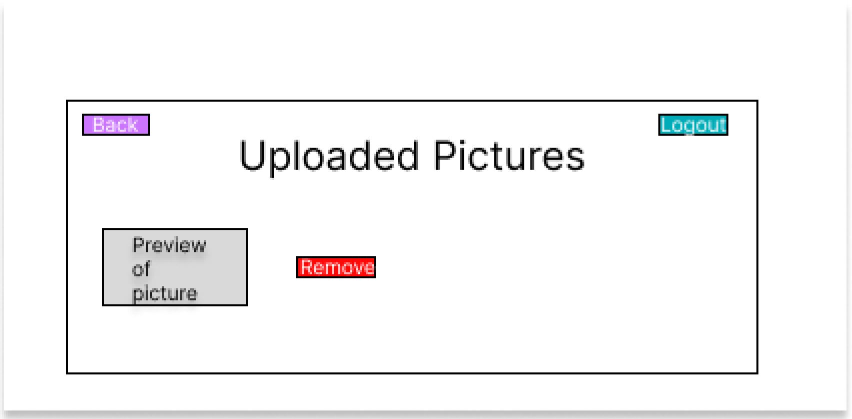
6. In Relation to **Logging,** I would recommend it happen in **FileService.** This will validate any retrieved or inserted data using SqlDAO and do any logging if needed. Logging can be used to determine the success rate in inserting data or if it is loading within the time constraint.
    a. If decided to **implement**, I would recommend passing in a **string userHash** and **UserProfile** as a parameter. It would look like:
        i. Task<Response>DownFile(string filename, int userId, int pinID, string userHash, UserProfile userProfile)

        Refer to line 232 how the logging is implemented:
        (https://github.com/JosephArmas/cecs-491A-Team-Big-Data/blob/main/SourceCode/back-end/TeamBigData.Utification.SecurityManager/SecurityManager.cs)

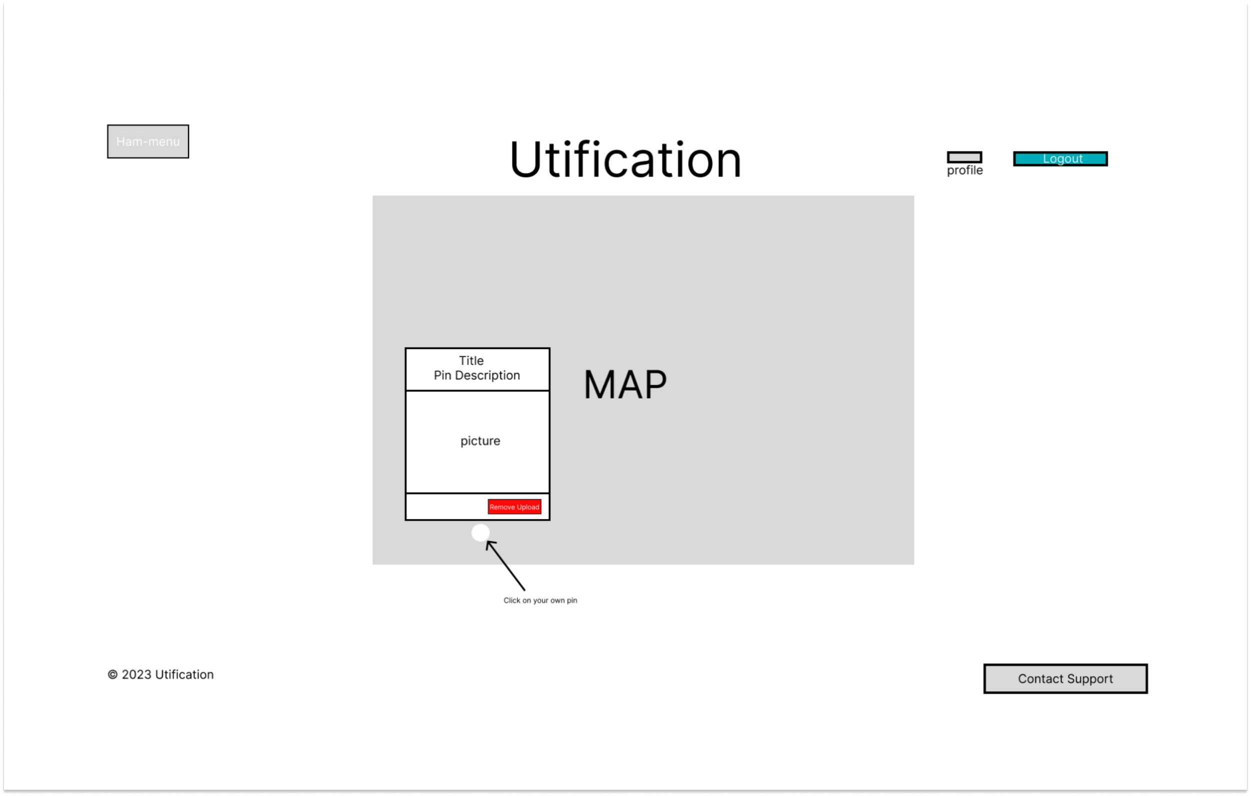
7. In Relation to as a User I am able to view my own uploaded images, possible Ui can look like this: (https://www.figma.com/file/Yn017yQFhCZIDdC8menQT/Utification-views?node-id=0-1&t=rCdEiSEbyVcZVwyn-0)



This is how it would look if a user were to click on the profile from the home page to be directed here. And If it the user were to click on Picture Uploads, it can look something like this: (https://www.figma.com/file/Yn017yQFhCZIDdC8menQT/Utification-views?node-id=0-1&t=rCdEiSEbyVcZVwyn-0)

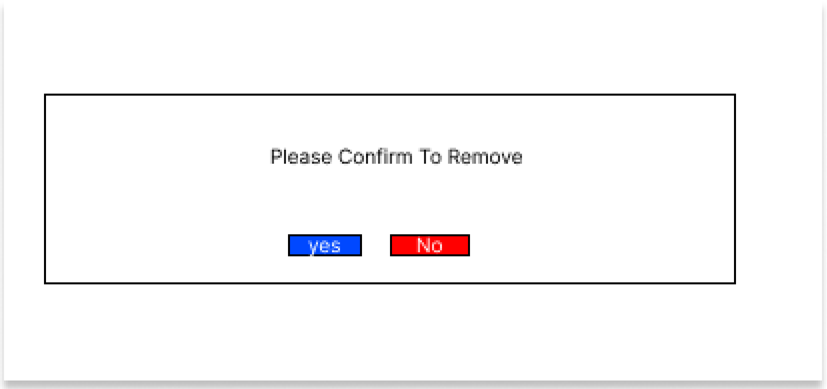


8. In relation to Deleting an image, a possible Ui can look like this: (https://www.figma.com/file/Yn017yQFhCZIDddC8menQT/Utification-views?node-id=0-1&t=rCdEiSEbyVcZVwyn-0)



This will allow the user who uploaded the photo the **option to remove** the picture uploaded.

a. Also consider when creating the table is to have a "**disable**" column in **dbo.PinPic**. The reason I say this is because if you don't want to completely delete it in the database, the disabled column can just act as a flag.
b. I think it would be a great idea to prevent any accidental clicking on remove is to prompt an alert or some kind of view to "**Confirm remove**" before actually removing.



9. In Reference to **UploadFileFail.png.** Although it has the front end code to check, it is missing the backend code of having the validation. It is needed because the front end is very insecure, hence the back end is needed to ensure security and input validation to be done in the manager layer. Moreover just reusing **FileUpload.png** is fine and satisfying different failure scenarios (**See Test Recommendation 1)**

10. This flow of control is similar to **FileUpload.png.** However, now the method name would be to delete an image. As result, it can look something like:
    Task <Response> DeleteFile(string fileName, int userID, pinID). This will allow the method to be asynchronous but also delete an image based on the owner, therefore taking in the userID, pinID would make sense to check if the relation is indeed their own pin.
    CheckIsPinOwner will stay the say, however, adding in check if the user is an admin, which will satisfy **BRD Success number 1.**

11. You can leave parameters the same, however a solution to renaming UploadFile method in FileService to **UploadNewFile**(string fileName, pinId, Userprofile cred) and the method in SqlDAO renamed to **InsertFile**(string fileName, int userID, pinID). This gives a better naming convention in determining what these methods are doing and can make it more clear to what method is causing an issue compared to having it the same name. Moreover, by doing it this way, the system follows a single responsibility in each layer. **UploadNewFile** will be responsible for uploading a new file and **InsertFile** will be responsible for inserting the file into the database.

Test Recommendations

1. One test recommendation is to do an ***end to end*** test ensuring that file is uploaded. This will ensure that the front end checks are indeed working and in the backend it is properly **inserting** into the database
    a. Moreover, in the backend you can do a ***unit tes***t in validating input. There should be a test based on the business rules for ***valid format*** and ***valid size.*** Each should be separate to satisfy Business requirement for:
        i. A User uploads an image that is not in .jpeg or .png format.
        ii. A User uploads an image larger than 9 megabytes.
    b. For the **Integration *test***, I would suggest that it is **properly inserting** into the database, this will ensure that the sql operation is properly inserting the values and show that the system is working as intended.

2. Similar to ***Test Recommendation 1***, but now for the system to ***properly retrieve*** from database
    a. Possible ***unit test*** is to retrieve the data from the database and do a ***check*** that the value is indeed correct and working as intended.
    b. In addition, retrieving data checks if the backend results are false in the Response obj, therefore, you can set the errorMessage property to "**Error. Cannot Access Images. Please Try Again.**" By doing this you will able to satisfy the **BRD failure condition** if ***A user is unable to Access their images***
        i. It would be a good idea to have a check condition inside of **fileUpload.js** because this is the action where the actual grabbing of the image from the S3 bucket occurs. It will ensure validity consistency in the backend to the frontend.

3. For testing if an ***image loads less than 7 seconds***. The *function that utilizes axios to GET*
   Should be timed the moment it is called and stopped when the image is displayed on the frontend. Also you should probably do it in the backend to keep it consistent. You can add the **stopwatch.start()** in the FileService when it calls the SQLDAO to GET from the database, after validating the retrieved data is when you can **stopwatch.stop().** By doing this, it allows the system to be consistent with the timings in the front end and back end.

a. Moreover you're able to create a base metric on what can be considered to be "***Too Slow***" based on how fast images load. For example, if it reaches 10s it can display *"image capacity limit"* which satisfies the **BRD failure condition** for **System slows due to too many images.**

Positive: You're able to keep it consistent with timing in the front end and the back end. As a result, you're able to conduct test on both front end and back end

Negative: There could be a potential time difference in the backend compared to the front end. Strictly only timing on the front end would be reasonable because that is the moment a user is trying to retrieve an image to be displayed.

4. Having an **integration test** for **deleting images** is to ensure that the system is working as intended. I would suggest doing something similar to **Test Recommendation 1** and ensure the correct alert error message is displayed. The reason why is because the operation is similar, however in this case is deleting. For **overall functionality**, It is recommended to do an **end to end test**.