GreetNGroup

Low Level Design Document

CECS 491A Sec 05 8332

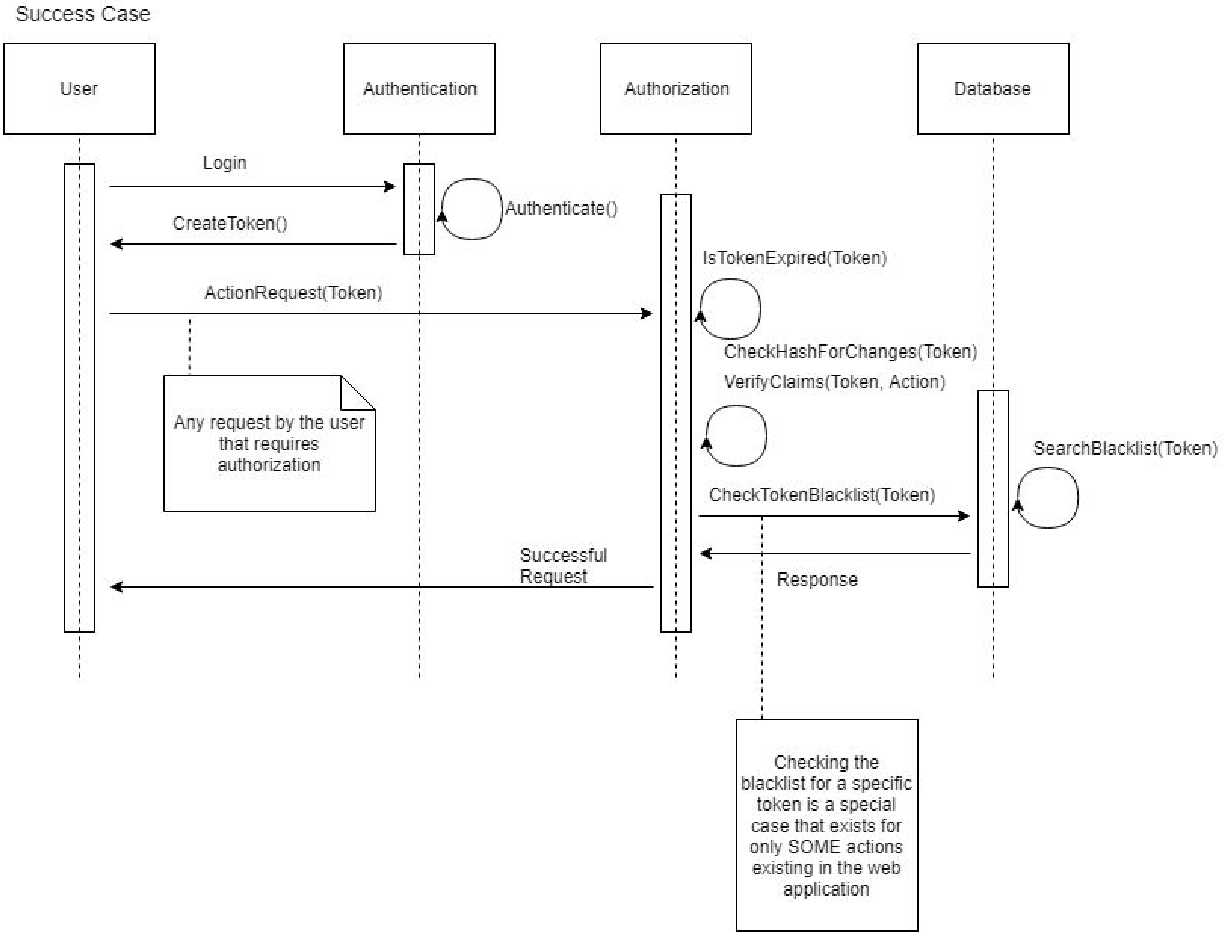
Team Gucci
Dylan Chhin 014430570
Eric Lee 014303261
Jonalyn Razon 014580772
Winn Moo 014633357 (Team Leader)

12/11/18

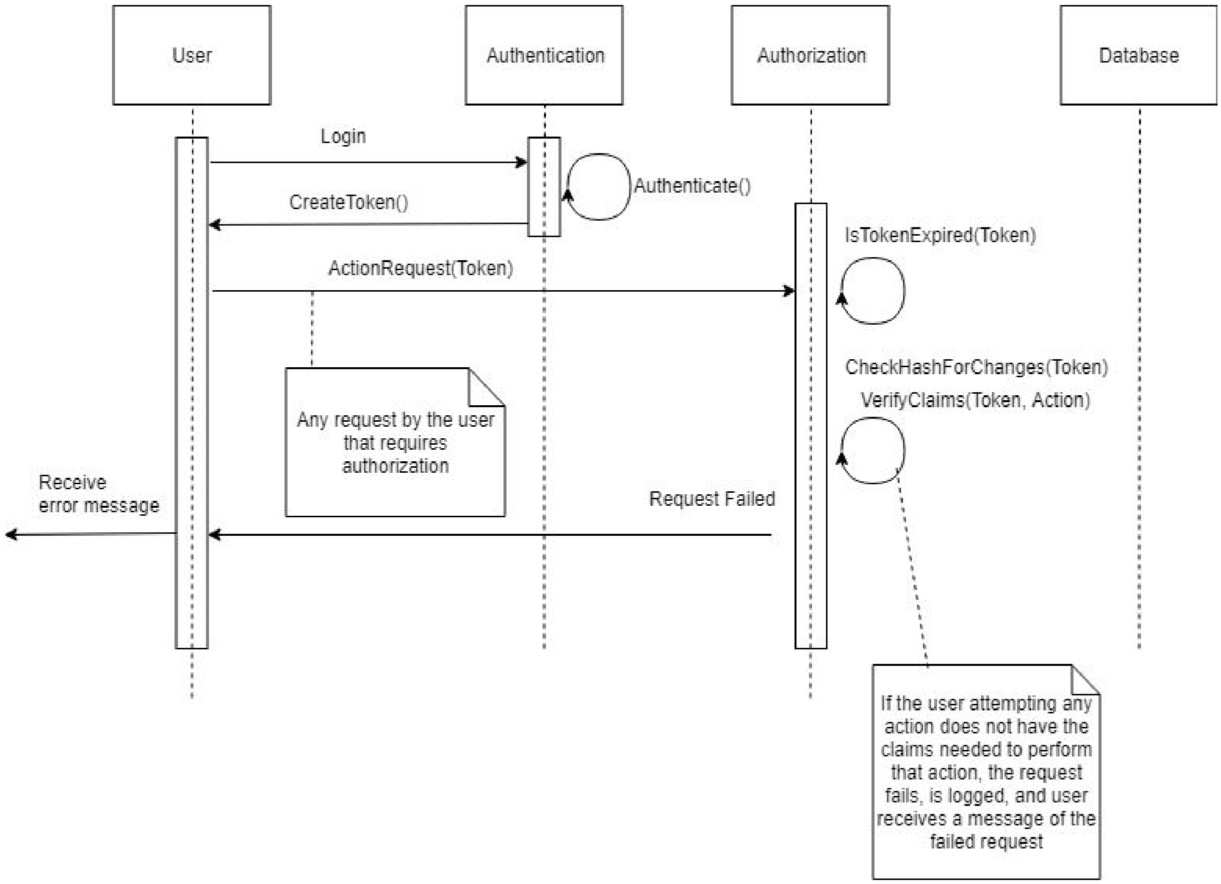
Table of Contents

Authorization

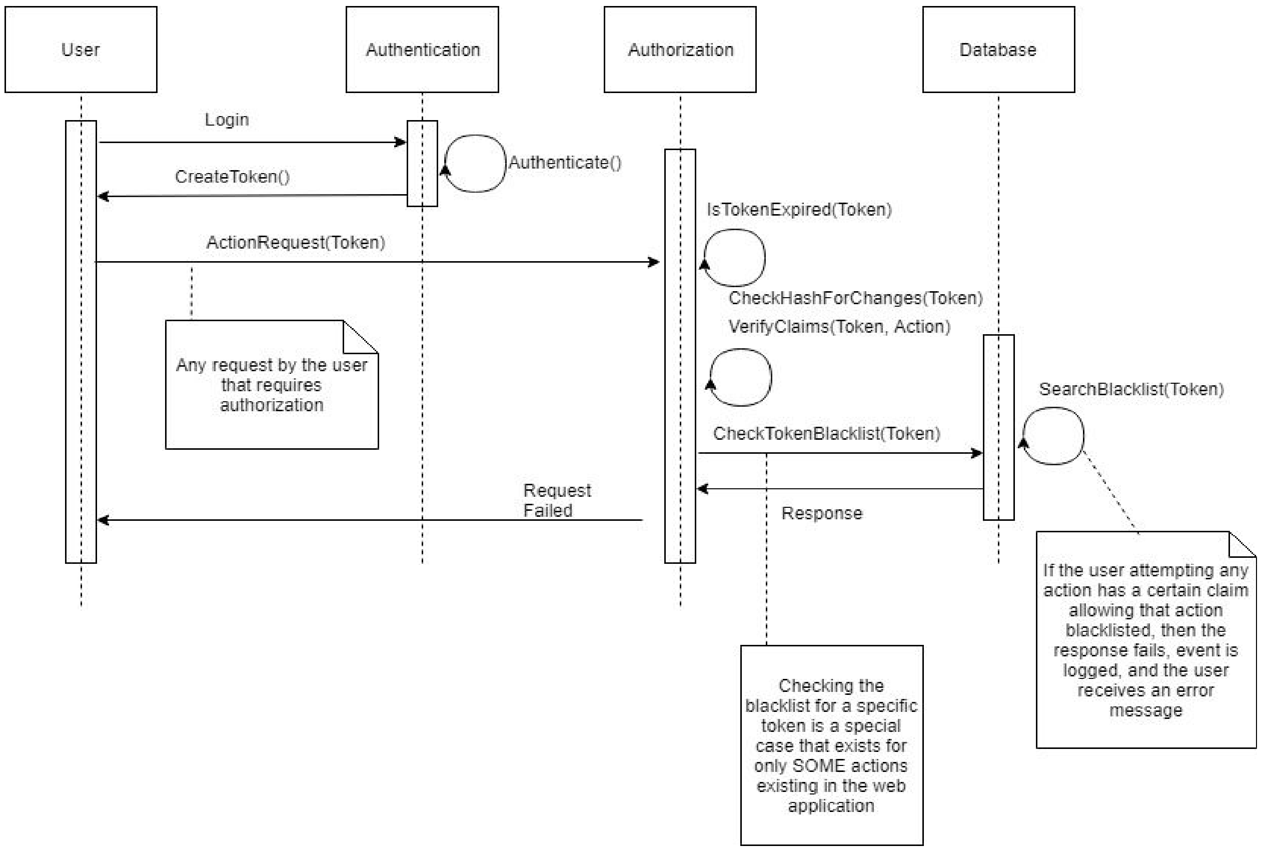
- For authorization, our system makes use of claims based authorization
- Claims are stored on a token that is given to the user when they login
    - This is done so that instead of storing claims on a database and having to constantly go into it, we can just skip going into the data access layer and check for claims quicker on the token
- Any time an action takes place, the token is passed as a parameter to be checked before granting the user authorization to carry on with the action
    - Based on whether the the token is expired, blacklisted, or contains the necessary claims for the action
- In a later implementation when tokens are to be given to users. Tokens will be hashed and sent to the user via the header. This has will be used for determining changes made on the token --if any has been done.


Success Case
User
Authentication
Authorization
Database
Login
Authenticate()
CreateToken()
IsTokenExpired(Token)
ActionRequest(Token)
CheckHashForChanges(Token)
VerifyClaims(Token, Action)
Any request by the user that requires authorization
SearchBlacklist(Token)
CheckTokenBlacklist(Token)
Successful Request
Response
Checking the blacklist for a specific token is a special case that exists for only SOME actions existing in the web application

Fail Case: Incorrect Token(s)


User
Authentication
Authorization
Database
Login
Authenticate()
CreateToken()
IsTokenExpired(Token)
ActionRequest(Token)
CheckHashForChanges(Token)
VerifyClaims(Token, Action)
Any request by the user that requires authorization
Receive error message
Request Failed
If the user attempting any action does not have the claims needed to perform that action, the request fails, is logged, and user receives a message of the failed request

Fail Case: Blacklisted Token(s) Found

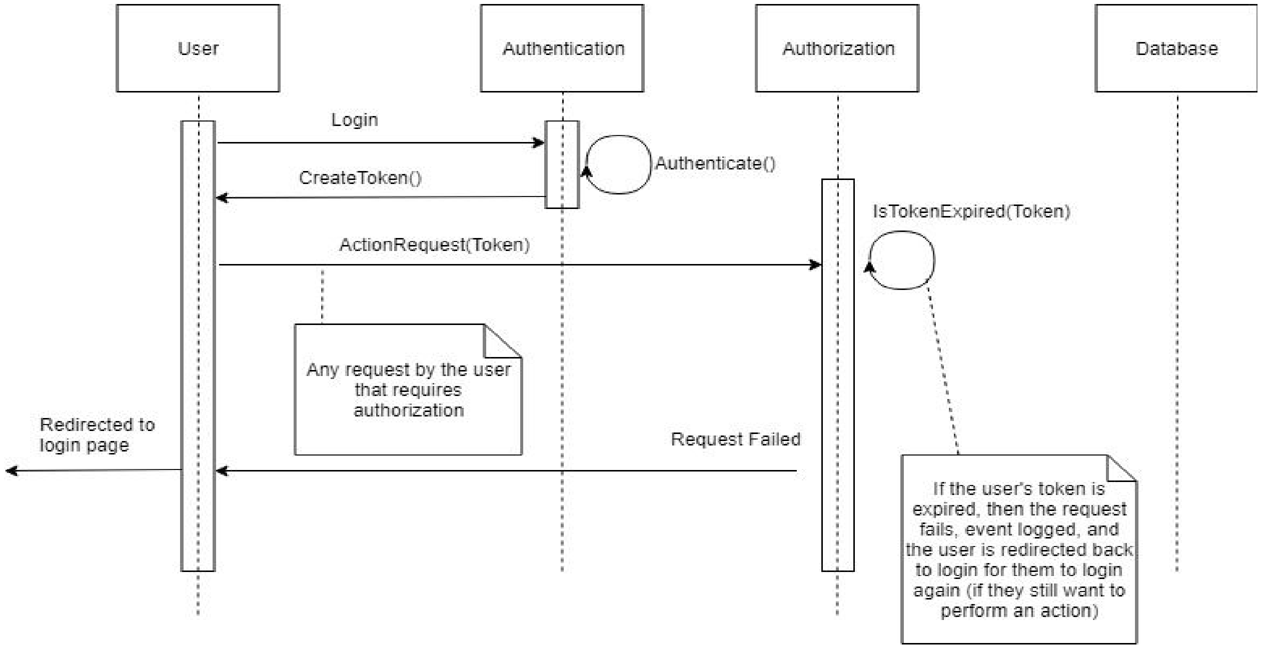


Any request by the user that requires authorization

If the user attempting any action has a certain claim allowing that action blacklisted, then the response fails, event is logged, and the user receives an error message

Checking the blacklist for a specific token is a special case that exists for only SOME actions existing in the web application

Fail Case: Expired Token



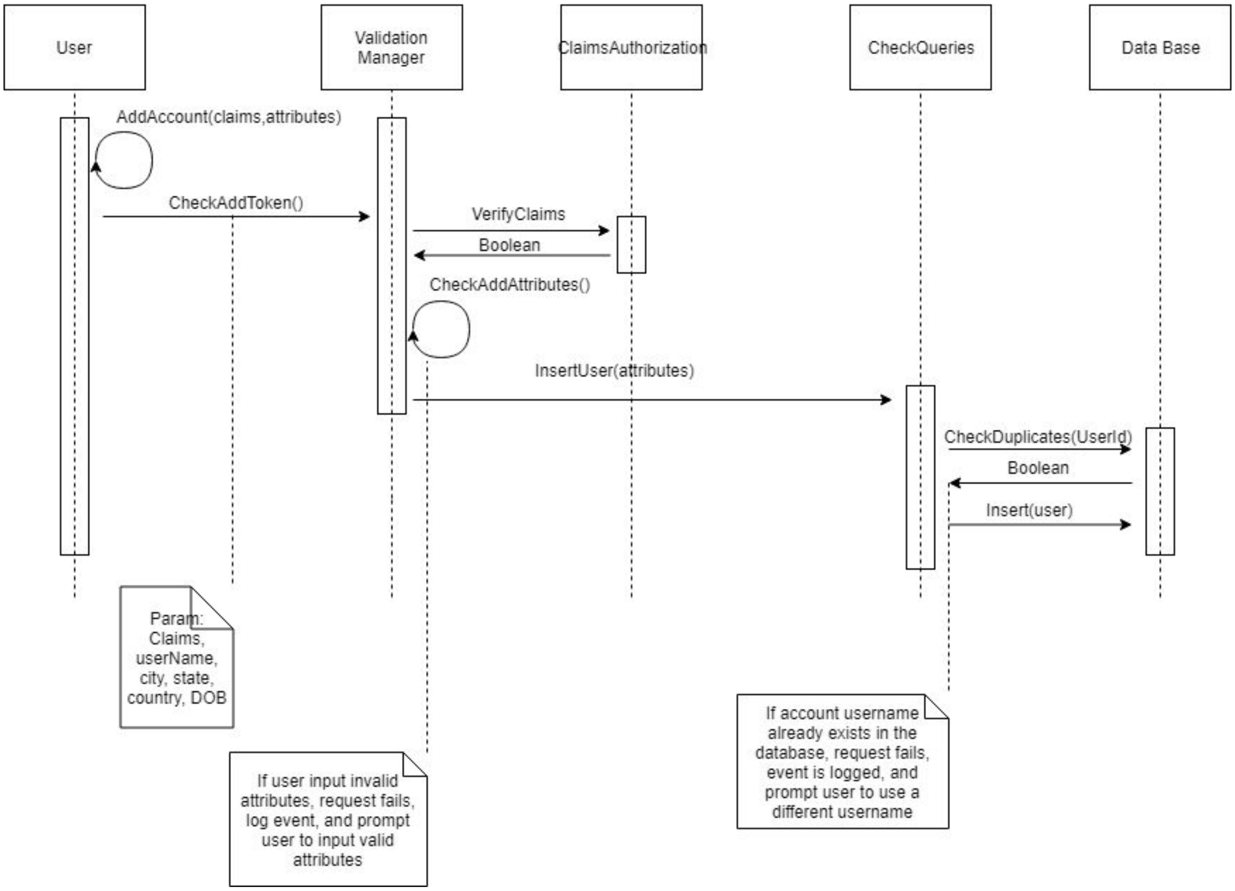
Any request by the user that requires authorization

If the user's token is expired, then the request fails, event logged, and the user is redirected back to login for them to login again (if they still want to perform an action)

User Management

- For user management, there are 3 functions that every user has access to, but their ability to use those functions relies on the user's claims
    - These functions are Add, Delete, and Update users
        - The disable/enable user accounts function has been merged into the Update function
            - A disabled or enabled account is simply marked with a boolean value in the User table
    - For example, only users who have admin or system admin claims should be able to add incoming user accounts who have registered or delete existing user accounts
        - Users who do not have these claims are able to register or request to delete their account; however, users with admin or system admin claims are the ones to actually add or remove their information in the database
- To prevent unnecessary requests to the database, checks in the authorization and data access object classes are done to catch any exceptions that would make the request invalid without needing to access the database
    - This will prevent slowing down the response time of a request since exceptions are caught before needing to do a database query

Add/Registering Account

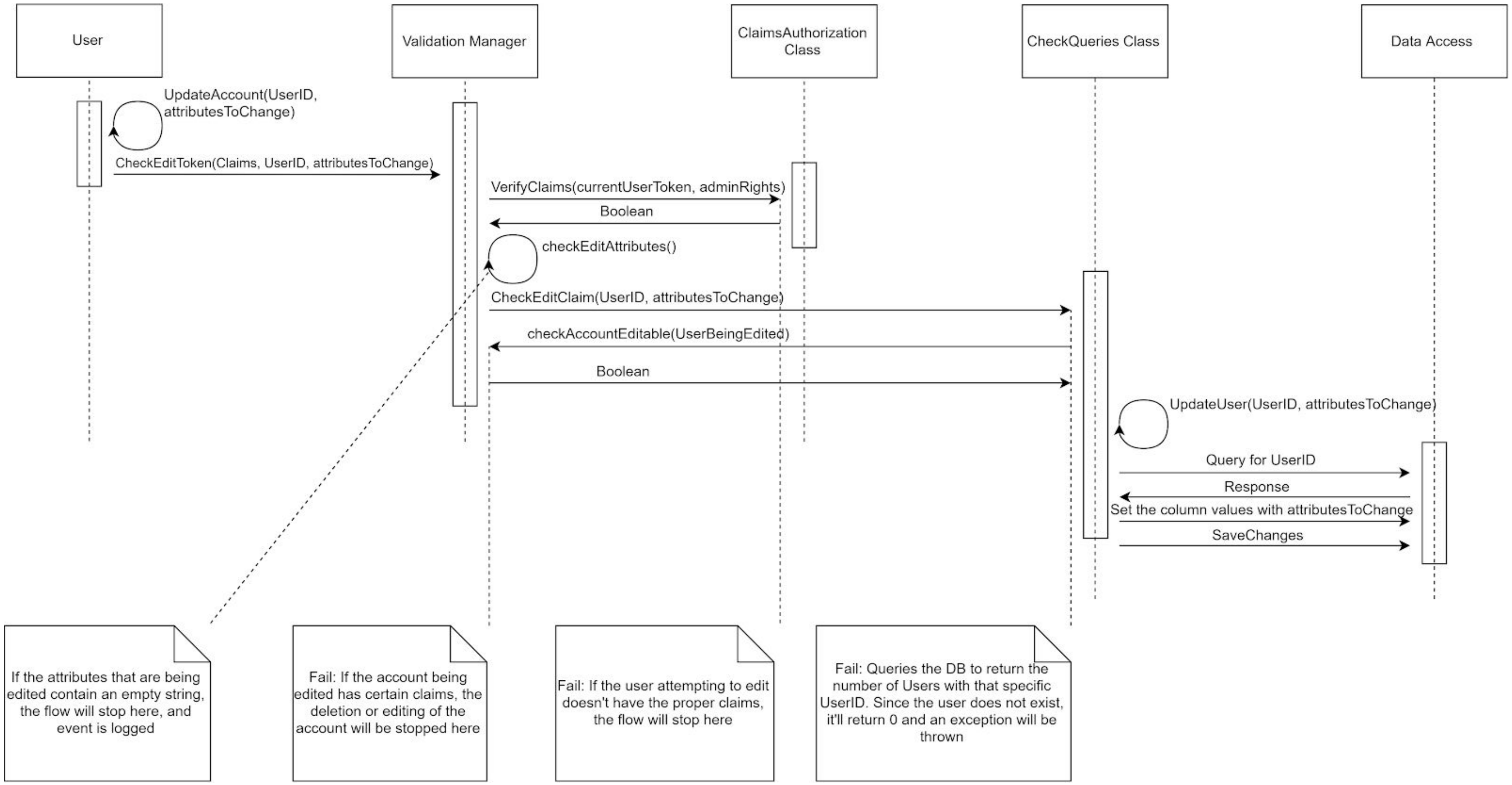


Param: Claims, userName, city, state, country, DOB

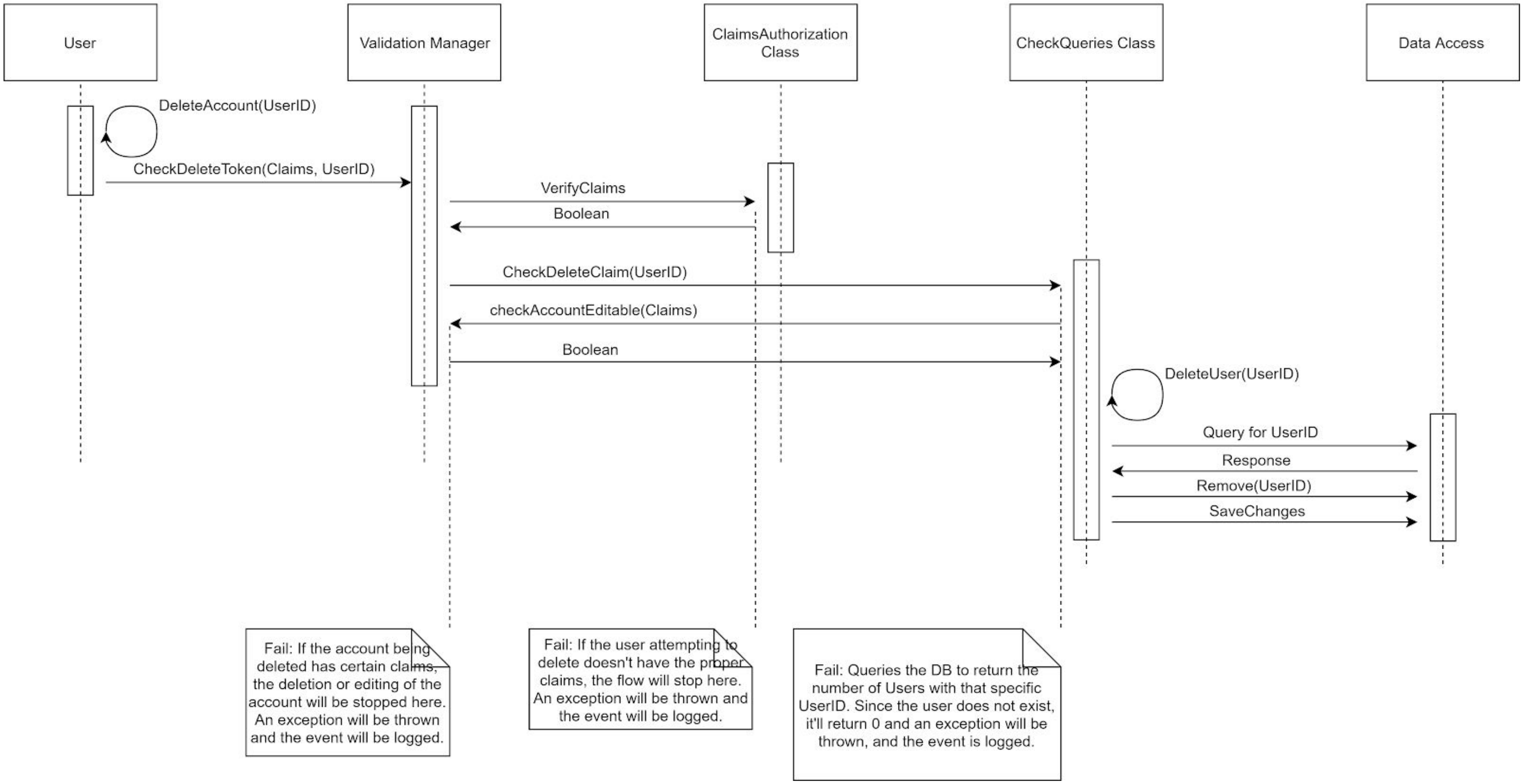
If user input invalid attributes, request fails, log event, and prompt user to input valid attributes

If account username already exists in the database, request fails, event is logged, and prompt user to use a different username

Editing User


User
Validation Manager
ClaimsAuthorization Class
CheckQueries Class
Data Access
UpdateAccount(UserID, attributesToChange)
CheckEditToken(Claims, UserID, attributesToChange)
VerifyClaims(currentUserToken, adminRights)
Boolean
checkEditAttributes()
CheckEditClaim(UserID, attributesToChange)
checkAccountEditable(UserBeingEdited)
Boolean
UpdateUser(UserID, attributesToChange)
Query for UserID
Response
Set the column values with attributesToChange
SaveChanges
If the attributes that are being edited contain an empty string, the flow will stop here, and event is logged
Fail: If the account being edited has certain claims, the deletion or editing of the account will be stopped here
Fail: If the user attempting to edit doesn't have the proper claims, the flow will stop here
Fail: Queries the DB to return the number of Users with that specific UserID. Since the user does not exist, it'll return 0 and an exception will be thrown

Deleting User



Fail: If the account being deleted has certain claims, the deletion or editing of the account will be stopped here. An exception will be thrown and the event will be logged.

Fail: If the user attempting to delete doesn't have the proper claims, the flow will stop here. An exception will be thrown and the event will be logged.

Fail: Queries the DB to return the number of Users with that specific UserID. Since the user does not exist, it'll return 0 and an exception will be thrown, and the event is logged.

Scope Creep - Password Verification

The PwnedPasswords API requires that passwords that are sent are hashed using the SHA1 algorithm

- To hash passwords, we will use the System.Security.Cryptography library
    - Specifically, we will use the SHA1CryptoServiceProvider class, as it is FIPS compliant[1][2][3]
    - Recreating the hashing algorithm on our own would be time consuming therefore simply using a library would be much more efficient
    - This namespace was created by the .net team at Microsoft, thus it is reputable
- When calling GET on the endpoint, you can supply either the full hash or just the first 5 characters
    - For the sake of security, we will send the first 5 characters

When we make requests onto the the PwnedPasswords API endpoints, they will all be over HTTP. The response from the API will be a list of hashes that match the first 5 characters of the password we hashed. To see if the user provided password has been hacked, we need to see if the rest of the 35 characters is in the list. At the end of the hash is the number of times that the password has appeared in the password list. What we do with this information is as follows:

- If the password appears once, we consider the password pwned. However, should business rules change, the number of occurrences that would be considered pwned can be changed through changing the value of the constant ‘`minimumPasswordOccurences`’

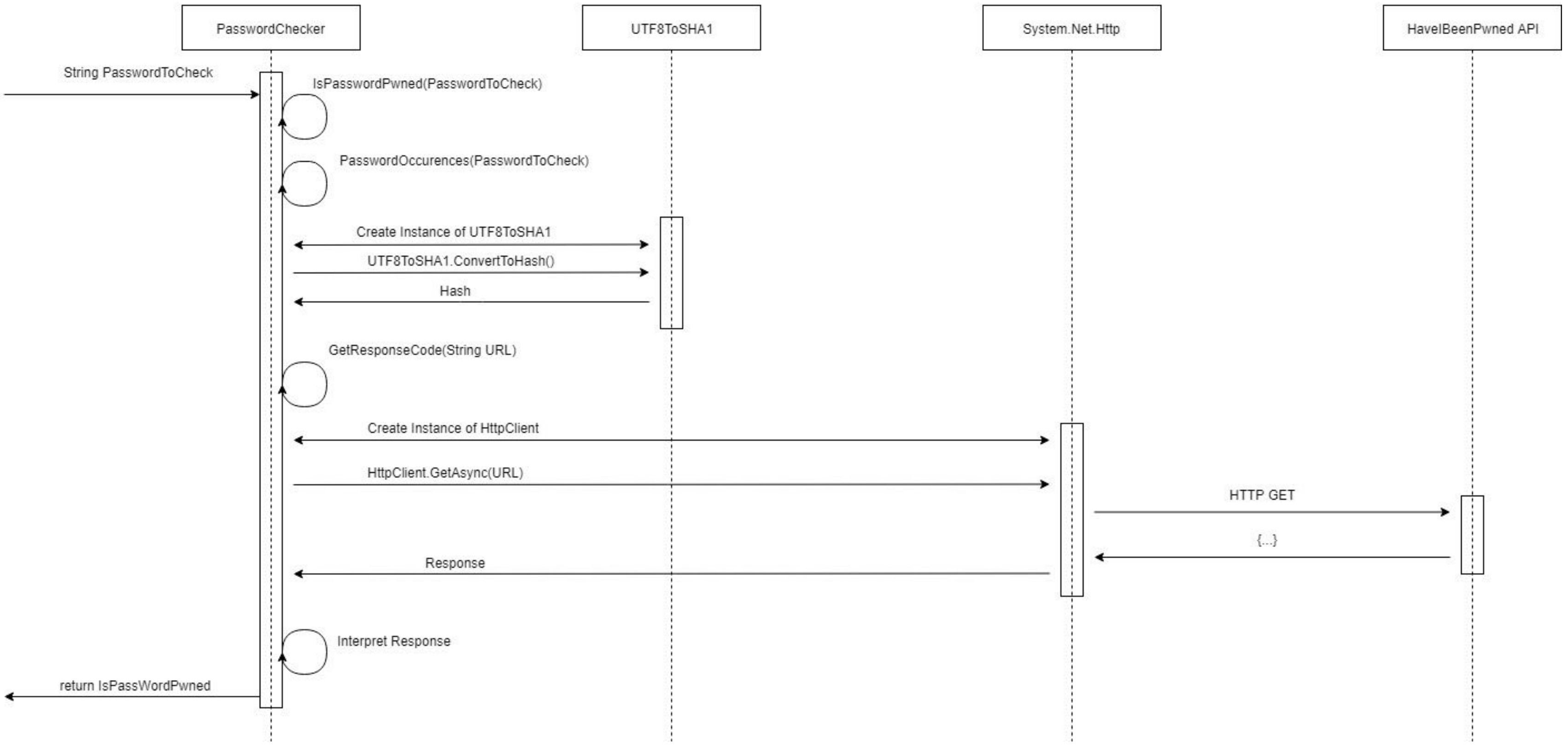
- If the password appears more than once, the `IsPasswordPwned` will return true
- If the password doesn’t appear, the `IsPasswordPwned` will return false
- The `IsPasswordPwned` function makes use of the `PasswordOccurences` function to reduce code reuse.
    - In the UI, `PasswordOccurences` will be used to show how many times the password has shown up in leaked password lists
    - Therefore, we decided it would be best to have two functions that do 2 separate tasks
        - But, >90% of the code would have been the same in the 2 functions, so we made `IsPasswordPwned` make use of `PasswordOccurences`
- Any errors encountered in both hashing or requests to the API will be logged using the Trace object found in the System.Diagnostics library

[1] http://blog.aggregatedintelligence.com/2007/10/fips-validated-cryptographic-algorithms.html

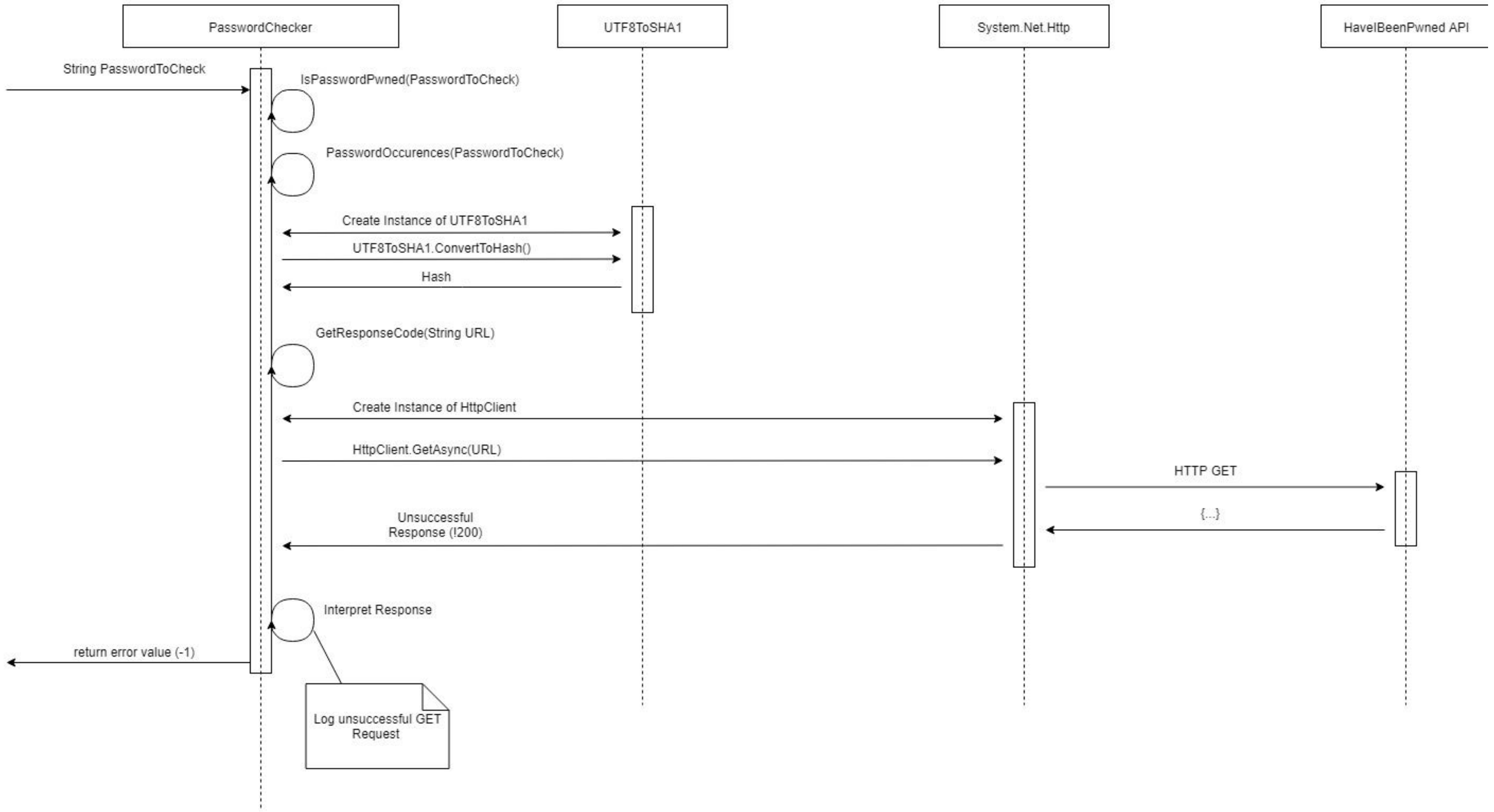
[2] http://codeissue.com/issues/i34dda6deaad90a/difference-between-sha1-sha1cryptoserviceprovider-sha1managed-and-sha1cng

[3] https://docs.microsoft.com/en-us/windows/security/threat-protection/fips-140-validation#id0ewfac

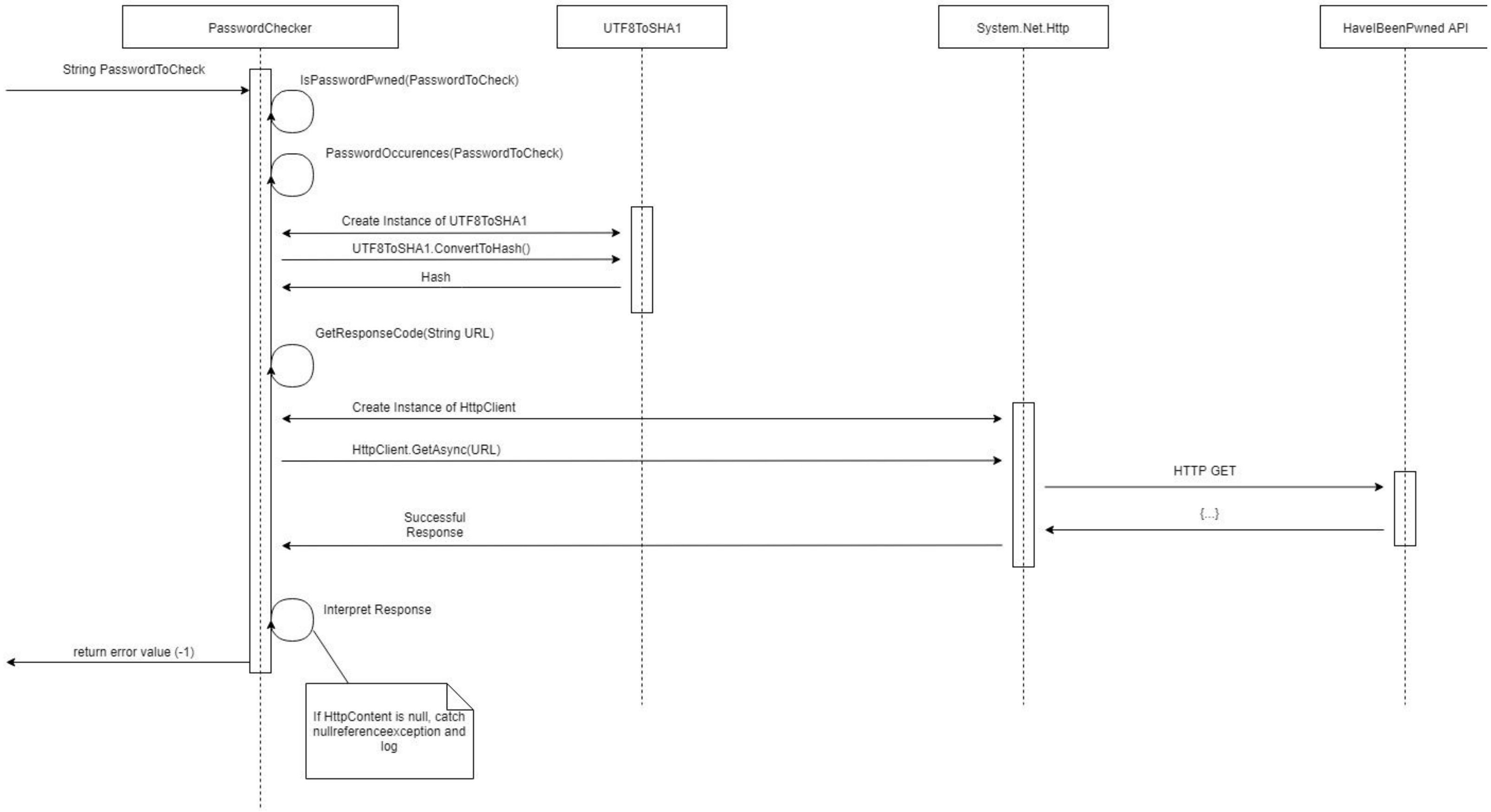
Success Case


PasswordChecker
UTF8ToSHA1
System.Net.Http
HaveIBeenPwned API
String PasswordToCheck
IsPasswordPwned(PasswordToCheck)
PasswordOccurences(PasswordToCheck)
Create Instance of UTF8ToSHA1
UTF8ToSHA1.ConvertToHash()
Hash
GetResponseCode(String URL)
Create Instance of HttpClient
HttpClient.GetAsync(URL)
HTTP GET
{...}
Response
Interpret Response
return IsPassWordPwned

Fail Case: Unsuccessful API
GET request



Fail Case: Null HttpContent


PasswordChecker
UTF8ToSHA1
System.Net.Http
HaveIBeenPwned API
String PasswordToCheck
IsPasswordPwned(PasswordToCheck)
PasswordOccurences(PasswordToCheck)
Create Instance of UTF8ToSHA1
UTF8ToSHA1.ConvertToHash()
Hash
GetResponseCode(String URL)
Create Instance of HttpClient
HttpClient.GetAsync(URL)
HTTP GET
{...}
Successful Response
Interpret Response
return error value (-1)
If HttpContent is null, catch nullreferenceexception and log