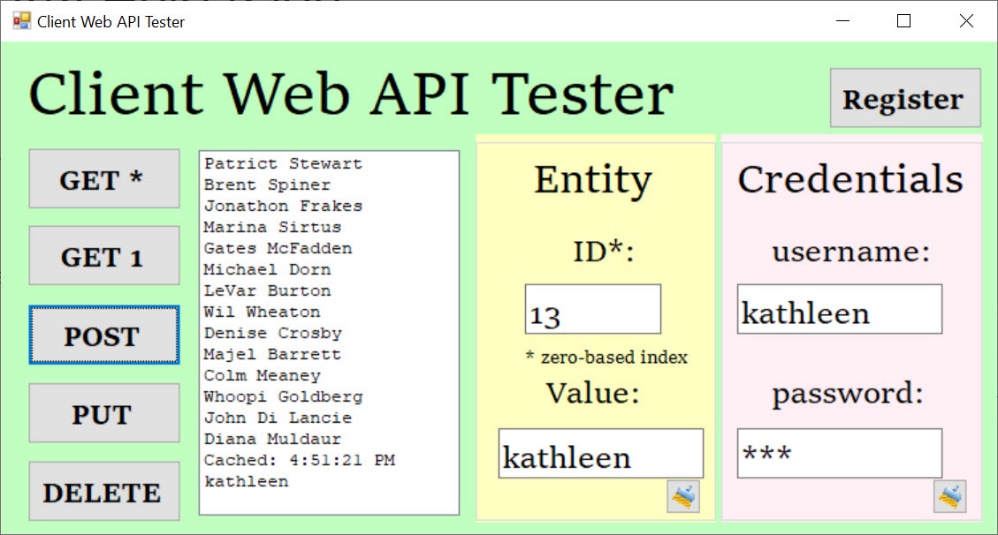
# ASP.Net Web API Basic Authentication Demo Application + Client Tester CRUD (Get, Post, Put, Delete) Windows Form Application

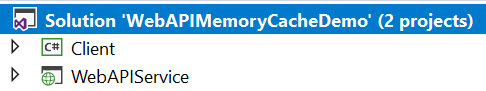
## 

## About

This project presents a Visual Studio solution including a simple demo ASP.Net Web API Basic Authentication Service Application and a “Tester” Client (Windows Form Application) that allows the user to test the Web API with CRUD operations (GET, POST, PUT, DELETE) that service design requires authentication (except the Get). In addition to demonstrating standard CRUD capabilities, the Web API service implements a .Net Memory Cache (MemoryCache) and custom username and password validator. Passwords are stored securely using Password-Based Key Derivation Function PBKD cryptology. The client “tester” windows form application is not intended as a UX/UI demo but used to test and verify that the backend authentication service allows the user to register with a username and password and verify the Web API CRUD functions based on basic authentication scheme. Lastly, the project is shown in the demo section with a video and screen captures. Note: Every Web API Authentication service should use secure transport. For brevity, this demo project does not implement or discuss the complicated detail nature of SSL/TLS.



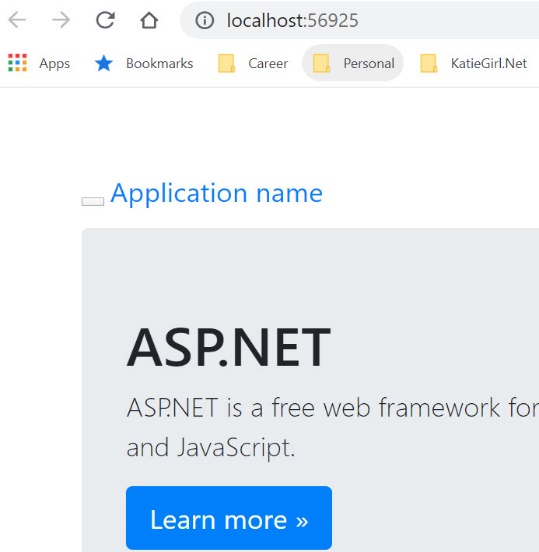
## Architecture



The demo project consists of these simple component topics:

* ASP.Net Web API Service (Hosted by IIS Express) Application Project “WebAPIService”
* Client “Tester to Service” Windows Form Application Project “Client”

## ASP.Net Web API Service Application



The ASP.Net Web Application template was used to create a simple Web API service and was hosted on my local development machine using IIS Express launched from Visual Studio. The API functions will be discussed after other topics. The project code is available on GitHub [here].

Topics:

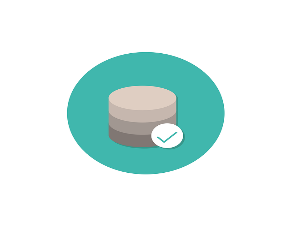
* User Modeling
* Memory Cache “MemoryCache” .Net Implementation
* PBDK (Password-Based Key Derivation) Secure Password Cryptology
* Custom Username and Password Validator
* Authentication Handler
* Registration Controller
* Values Controller

### User Modeling



The application *Models* folder contains a class called *User* that represents a “registered” user. The User model is very similar to the User model discussed and used in a previous demo project article [link here <https://portfolio.katiegirl.net/2020/02/12/wcf-authentication-service-application-custom-validator-secure-password-storage-error-handler-certificates-client-tester-windows-form-application/>] except it does not store a callback reference. The User model contains properties for cryptology and also has a string override for the object to list the username property of the User object. Please reference the previous demo project article referenced above for detailed discussion regarding the User model. The class code is available on GitHub [here].

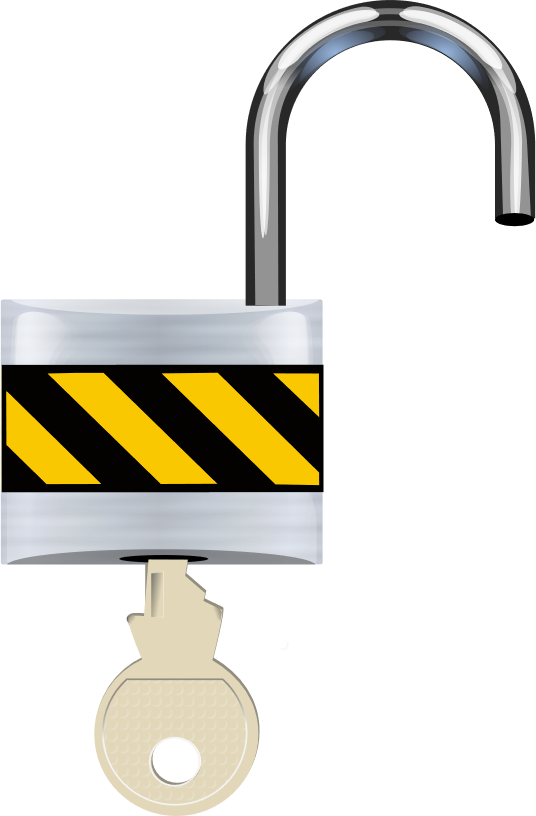
### Memory Cache “MemoryCache” .Net Implementation



Since this is not a backend database demo, I implemented a .Net “MemoryCache” to simulate calls to the backend database store where user registration and application data information would be stored and accessed in production-ready applications. This is implemented in both the *Registration* and *Values* controller and also in the *Authentication* message handler.

I previously created a Web API Demo project specially to demonstrate an ASP.Net Web API Memory Cache implementation. The project article [here <https://portfolio.katiegirl.net/2020/01/09/asp-net-web-api-memory-cache-service-application-client-tester-crud-get-post-put-delete-windows-form-application/>] is a great resource for understanding what a MemoryCache is and how-to implement this in a Web API with my experience tips and words of wisdom. In this project demo, I will not discuss in detail regarding my MemoryCache implementation. Please reference the previous demo for reference on how to implement a MemoryCache in ASP.Net Web API.

### PBDK (Password-Based Key Derivation) Secure Password Cryptology



The *ServiceCryptology* class implements Password-Based Key Derivation Function *PBKD* for password management. I created a previous demo project and blog article [link here <https://portfolio.katiegirl.net/2020/02/12/wcf-authentication-service-application-custom-validator-secure-password-storage-error-handler-certificates-client-tester-windows-form-application/>] that discusses the nature and implementation of PBDK cryptology. I used the same cryptology design and code implementation in this project demo. Please reference the previous demo for reference on how to implement PBKD Cryptology. The class code is available on GitHub [here].

### Custom Username and Password Validator



The *UserNamePasswordAuthenticator* class is a custom username and password validator. It only validates against previous registered users and if a user is not currently registered it does not validate the caller to the service as being authenticated. The class obtains the list of registered users from the MemoryCache to simulate an expensive backend database server call to query the username and password credential store. The validator then utilizes cryptology *PBDK* (see prior section) to check the input username and password against the credential store of *RegisteredUsers* in the MemoryCache. After a successful comparison using the PBDK, the validator passes the success state back to the *Authentication Handler* (see next section) so that it can then attach identity credentials. The class code is available on GitHub [here].

[insert code here]

### Authentication Handler



The *Authentication Handler* is a message handler that processes the user authentication to the Web API application. The class code is available on GitHub [here]. The implementation details are detailed below.

#### Configuration

This custom message handler should be added first to the processing pipeline so that the user is authenticated early on. I added this handler to the W*ebApiConfig* file registration under *App\_Start*.

[insert code]

#### SendAsync Method

This is TPL (*task parallel library*) message handler for authentication. Method inputs include the incoming *http request message* object from the client and a *cancellation token* that allows cancellation. This method is called as the application "chains" through its event handlers and the pipeline. The cancellation token allows monitoring and potential cancellation of the request although this particular demo does not monitor and process cancellation requests. The *SendAsync* method main function is to process the incoming http request message object, then call the base message handler to continue the processing pipeline. Finally, the method processes the final *outgoing http response message*.

[insert code here]

#### ProcessRequestMessage Method

This method takes the input “*HttpRequestMessage*” incoming request object and processes the http request authorization headers. This method extracts the username and password, then validates them against a credential store of *existing* username and password using a custom username and password validator. If they are valid, it builds and sets a *ClaimsPrincipal* of identity. This process allows the application to provide credentials early on in the pipeline processing before any of the Web API controller methods are called. The Web API application is now able to extract from the thread, the context and authentication principal identity information. This allows the service application to know which groups the *current user* belongs to and their permissions. The Web API controller can now filter out *authenticated users* at the method level with simple attribute such as using the [*Authorize*] attribute.

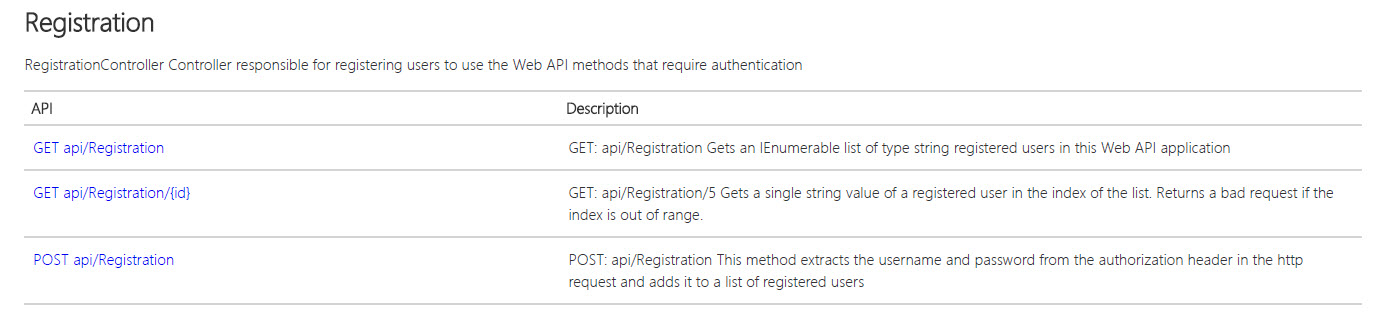
[insert code here]

#### ProcessResponseMessage Method

This method takes the input and final *outgoing http response* object to the client. The method checks to see if the response status code is *unauthorized* and adds further information to the response to indicate the specific authentication scheme that is required. If the response is unauthorized, then the process pipeline falls here eventually.

[insert code here]

### Registration Controller



This *Registration* controller is a generic default Web API controller implementation responsible for registering users to the service application. I did modify and customize the GET and POST methods to simulate CRUD operations that could be done on a backend database. The Memory Cache simulates “conserving” expensive resource calls to the backend database by keeping the contents in memory for a specified duration. The Registration API is described in the image above. Users must first register to the application service before they are authorized to do certain requests on the regular service data (POST, PUT, and DELETE). The code is available on GitHub [here].

#### Constructor

The *Registration* constructor gets called each time the client does a Web API request. A controller instance is created after the construction for the Web API actions. The constructor provides an opportunity to check the existence of the *key-value pair* in the MemoryCache and if it does not exist, call a backend database to repopulate it. In this demo, I am simulating the database call by reinitializing a default List of *User* objects called *RegisteredUsers* with two test users and also placed it in the MemoryCache with a longer expiration policy.

[insert code here]

#### Get (Collection)

The Get (Collection) retrieves the *RegisteredUsers* key-value entry from the cache and then returns an IEnumerable collection of User objects (as strings) to the caller. Note: The method requires no prior user registration or authentication to access the service.

[insert code here]

#### Get (Entity Id)

The Get (Single Entity) first retrieves the *RegisteredUsers* key-value entry from the cache as an IEnumerable collection, then returns the User object (as a string) at that index location in the collection. It first checks to see if the caller’s index is in range before returning the entity. If the specified index is out of range, it will return a BadRequest response code to the caller. Note: The method requires no prior user registration or authentication to access the service.

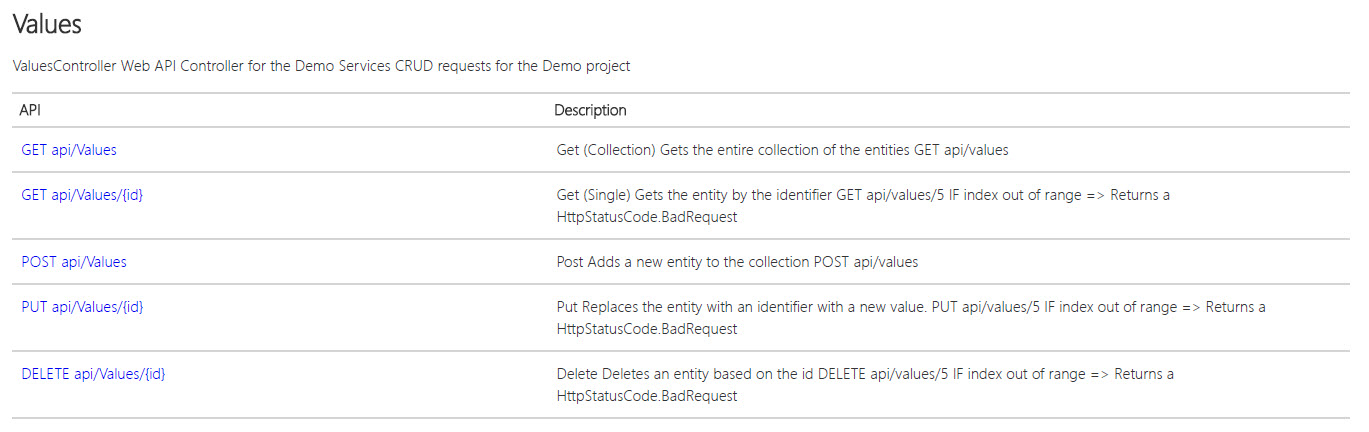
[insert code here]

#### Post

The method first retrieves the cache from a key *RegisteredUsers* into a List object in the method. Because the MemoryCache object reference is set to the local object, all actions done on the local *List<User>* object takes effect in the key-value entry in the cache. The method then extracts the *authorization header* from the current http request object to obtain the username and password. If the username does not already exist in the current List<User> of *RegisteredUsers*, the Post method creates a new *User* object and simply adds this object as new entry to the List of *RegisteredUsers*. In case of pre-existing registered users and other issues, the http response is formed with appropriate status code and reason. Note: The method requires no prior user registration or authentication to access the service.

[insert code here]

### Values Controller



This *Values* controller is the generic default Web API controller implementation and I did not rename or do any custom routing other than use the default routing. This is a basic demo with focus on showing the *Memory Cache* and client implementations. However, I did modify and customize the GET, POST, PUT, and DELETE methods to simulate CRUD operations that could be done on a backend database. The Memory Cache simulates “conserving” expensive resource calls to the backend database by keeping the contents in memory for a specified duration. The Values API is described in the image above. The code is available on GitHub [here].

#### Constructor

The *Values* constructor gets called each time the client does a Web API request. A controller instance is created after the construction for the Web API actions. The constructor provides an opportunity to check the existence of the *key-value pair* in the MemoryCache and if it does not exist, call a backend database to repopulate it. In this demo, I am simulating the database call by reinitializing a *default List* of string objects called “People” and then placing the key-value item in the MemoryCache object. I also initialized a default List of *User* objects called *RegisteredUsers* with two test users and also placed it in the MemoryCache with a longer expiration policy.

[insert code here]

#### Get (Collection)

The Get (Collection) retrieves the key-value entry *People* from the cache and then returns an IEnumerable collection to the caller. Note: The method requires no prior user registration or authentication to access the service.

[insert code here]

#### Get (Entity Id)

The Get (Single Entity) first retrieves the key-value entry *People* from the cache as an IEnumerable collection, then returns the object value at that index location in the collection. It first checks to see if the caller’s index is in range before returning the entity. If the specified index is out of range, it will return a BadRequest response code to the caller. Note: The method requires no prior user registration or authentication to access the service.

[insert code here]

#### Post

The method first retrieves the cache value (List object) from a key *People* into a List object in the method. Because the object reference is set to the local object, all actions done on the local List<string> object takes effect in the key-value entry in the cache. Post simply adds a new entry to the List<string> object. The [*Authorize*] attribute was added to require a user be pre-registered (see *Registration Controller API*) and authenticated (see *Authentication Handler*) before having the authority to perform this function.

[insert code here]

#### Put

The method first retrieves the cache value (List object) from a key *People* into a List object in the method. Because the object reference is set to the local object, all actions done on the local List<string> object takes effect in the key-value entry in the cache. Put simply replaces an entry to the List<string> object at the specified index. However, the method first checks to determine if the index is in range. If out of range, it will return a BadRequest response code to the caller. The [*Authorize*] attribute was added to require a user be pre-registered (see *Registration Controller API*) and authenticated (see *Authentication Handler*) before having the authority to perform this function.

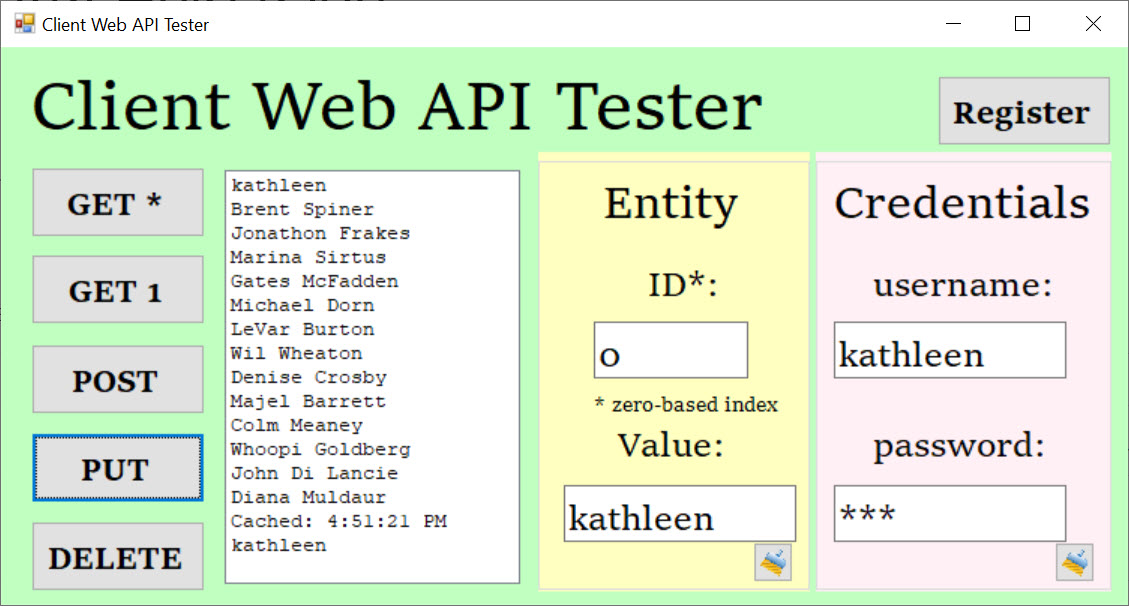
[insert code here]

#### Delete

The method first retrieves the cache value (List object) from a key *People* into a List object in the method. Because the object reference is set to the local object, all actions done on the local List<string> object takes effect in the key-value entry in the cache. Delete simply deletes an entry in the List<string> object at the specified index. However, the method first checks to determine if the index is in range. If out of range, it will return a BadRequest response code to the caller. The [*Authorize*] attribute was added to require a user be pre-registered (see *Registration Controller API*) and authenticated (see *Authentication Handler*) before having the authority to perform this function.

[insert code here]

## Client “Tester to Web API Service” Windows Form Application



The client “tester to service” is a simple windows form application project (*Client*) in the same solution that connects to the Web API by constructing a .Net HttpClient object and setting the path to the Web API service. The client program has a useful GUI to aid the user to test the GET, POST, PUT, and DELETE calls to the Web API with parameters specified in the data entry boxes. The client also allows the user to enter credentials and register on the service. After successful registration, the user will be able to successfully complete the POST, PUT, and DELETE requests only if they include the same credentials at the time of their request. The project code is available on GitHub [here].

### Features

The client application was meant to only quickly demonstrate consumption of the Web API service, so this client has limited scope of features. Some of the client application features are:

#### Simple Design

The user interface is simplified to do basic demo, testing, verification, and showing results in the GUI. After executing a Get(\*) or any request other than Get(1), the user can select an item in the list and the index data entry box will populate with the correct entity index. The user can also manually enter an index, but the list value will not be selected. The user can select an item from the list, let the index box auto-update, and then perform PUT, DELETE, or GET(1) Web API requests. Note: The GET(\*) is called after every PUT, POST, and DELETE request to update and refresh the contents of the local list shown in the GUI. The user can also easily POST a new entity by entering a string value into the data entry box and clicking on POST. The user can also select an existing item in the list, the index will show, and then type an entry into the Value text box to PUT or replace an entry. Likewise, the user can select an entry in the list box, the index selection will update, and then DELETE the entity.

#### Results in the GUI

After each Web API call (except Get (1)), the GUI List Box is populated with the current contents of the controller’s cache key-value entry. This would simulate a database query on the service and seeing the results of the query in the client GUI. For the Get(1) request, the List Box is only populated with the entity at the index that was requested by the user. Note: The GET(\*) is called after every PUT, POST, DELETE request to update and refresh the contents of the local list shown in the GUI.

#### Feedback of Bad Web API Responses

If the service returns any response other than “successful” the problem is caught in the client and alerts the user with a Message Box dialog with the code and error.

#### MemoryCache Verification

The service places a string object indicating the current DateTime that the MemoryCache key-value entry was created. This information is returned to the client in the same List<string> object with the remainder of the “People” items. This allows the user to verify that the MemoryCache entry is expiring on schedule and refreshing its contents with the default values per the design. The user can verify the MemoryCache is working in the Client by selecting the Get(\*) button after the policy expiration interval to verify the DateTime is changed.

#### User Registration

The user credentials are parsed from the data entry text boxes and placed into the authorization header of the client default http request header. Upon clicking the button, the client sends a request to the Registration service on the Web API with the credentials embedded in its request.

### Program Code

The code behind file is for the client tester and manages the application in one file. The code is available on GitHub [here].

#### Fields

Private fields allow the client to work with the following:

* Http Client for Web Requests
* Paths to Web API Services (Registration and Values)

#### Methods

##### Events

Multiple button click events handle the user requests including registration. Some events may first validate user text input from the GUI, then call a helper method to further assemble the HttpClient object for the GET, POST, PUT, or DELETE requests.

##### Helper Methods for Web API

These methods run after a button click request, work to further setup the HttpClient object, communicate with the Web API service, and process the results asynchronously. If there is a response other than successful, it throws an exception that is handled further upstream with a user dialog alerting the user of the Web API issue during testing.

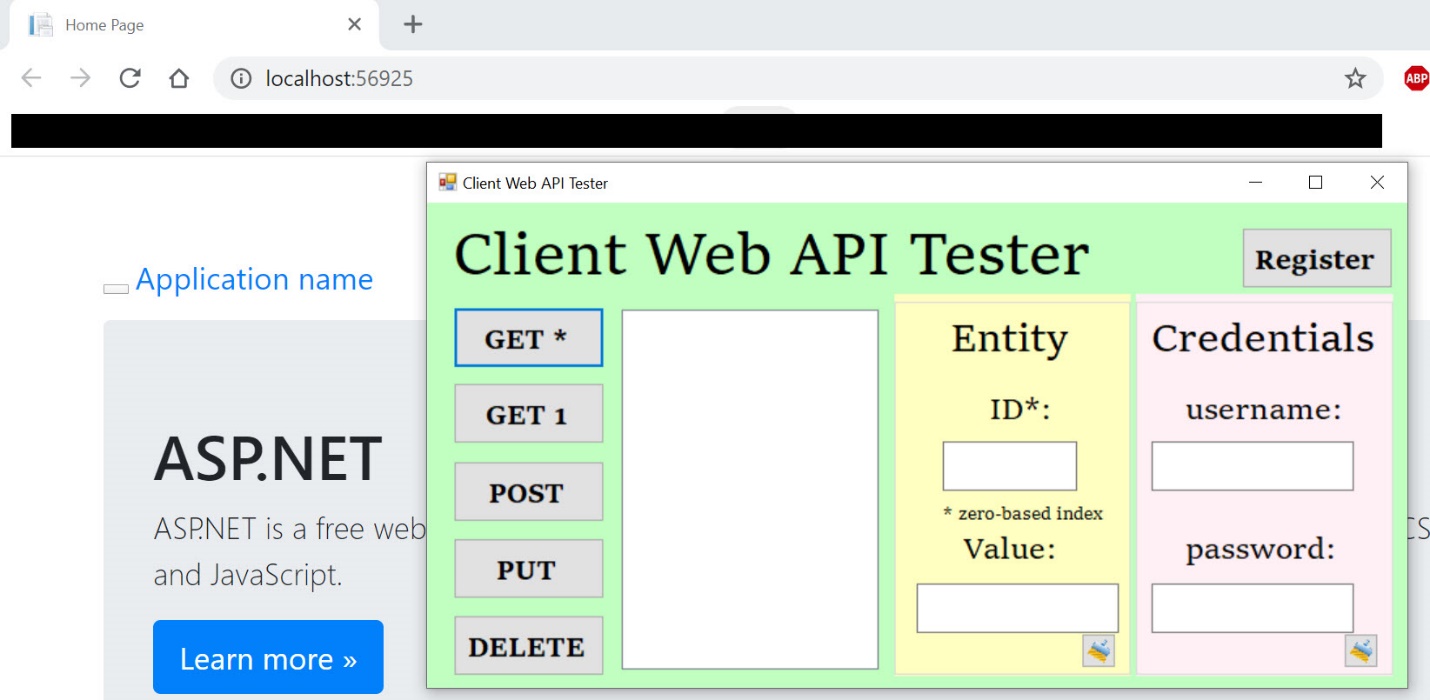
## Demo

A video demonstration of the project is available on YouTube here:

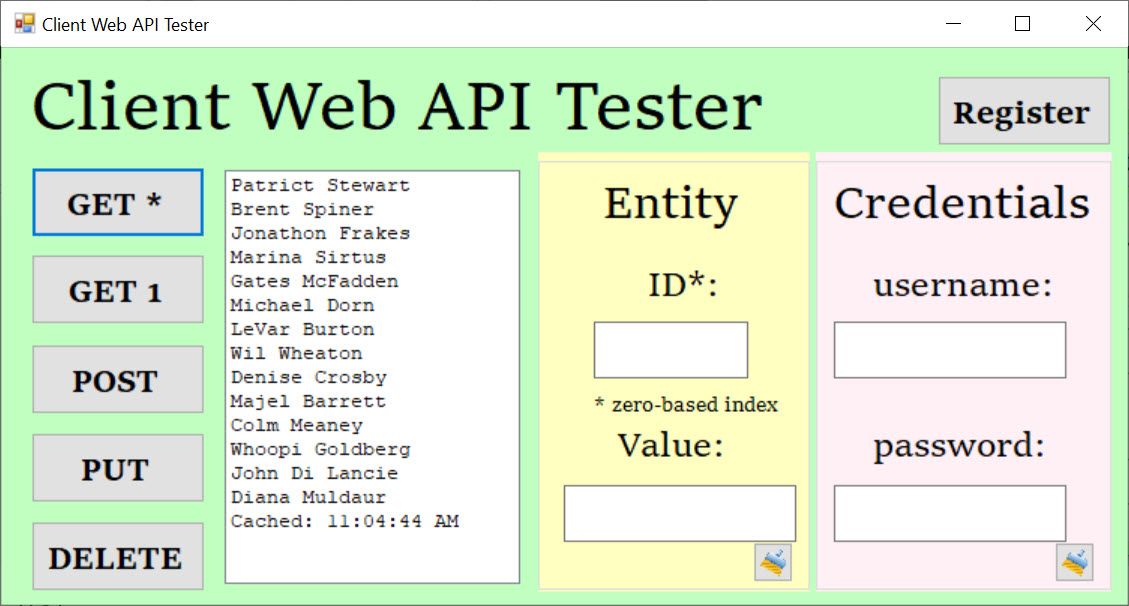
[insert YouTube video]

Screen captures are shown below for various user activities:

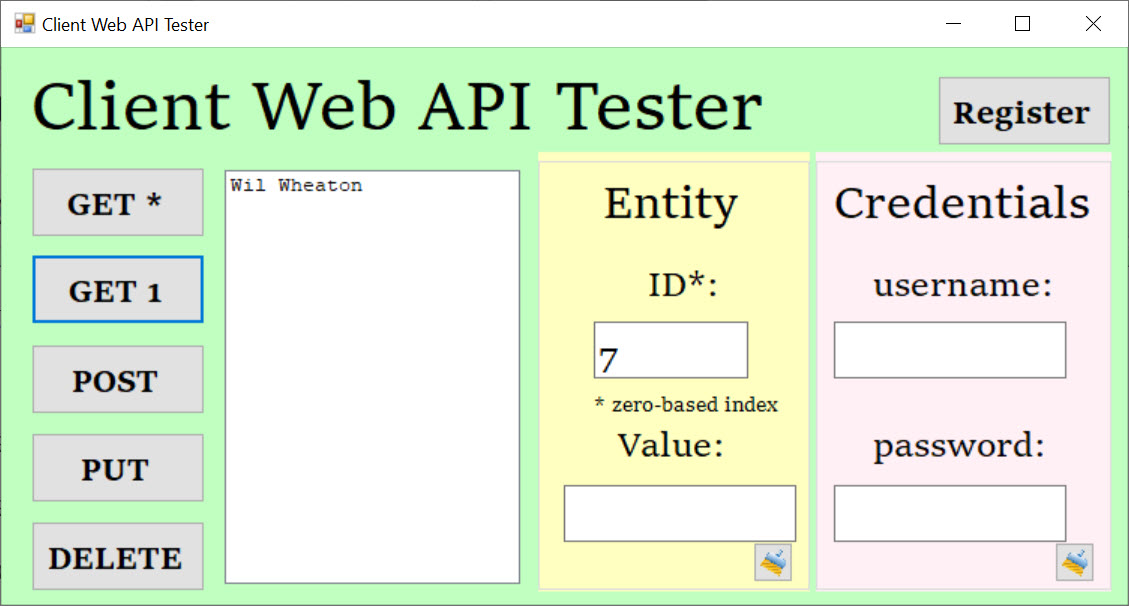
1. After all projects are loaded from Visual Studio



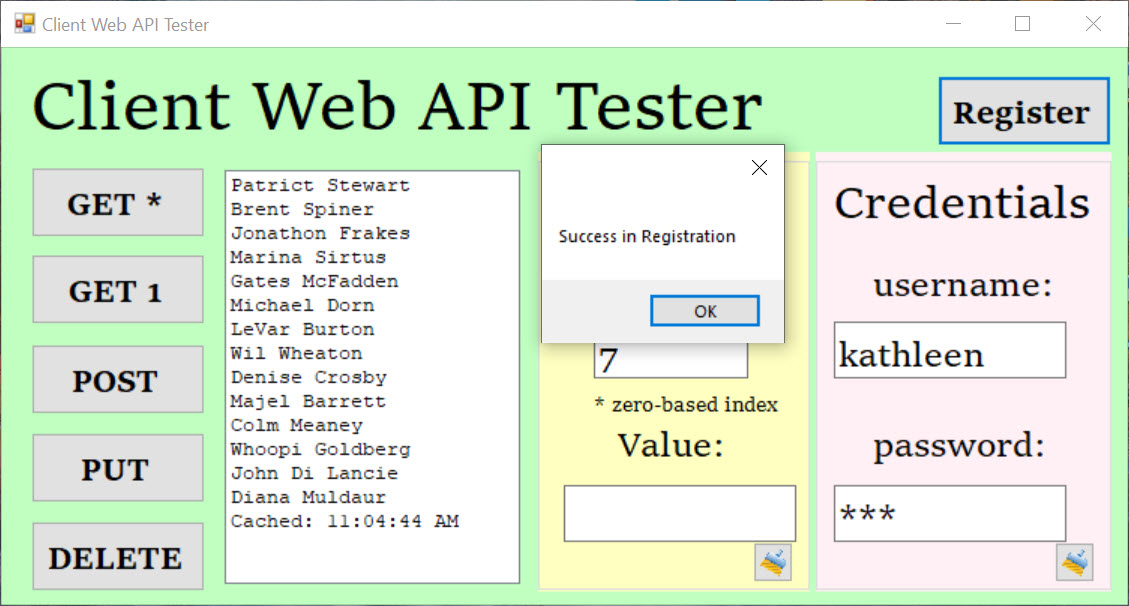
1. After Get(\*)



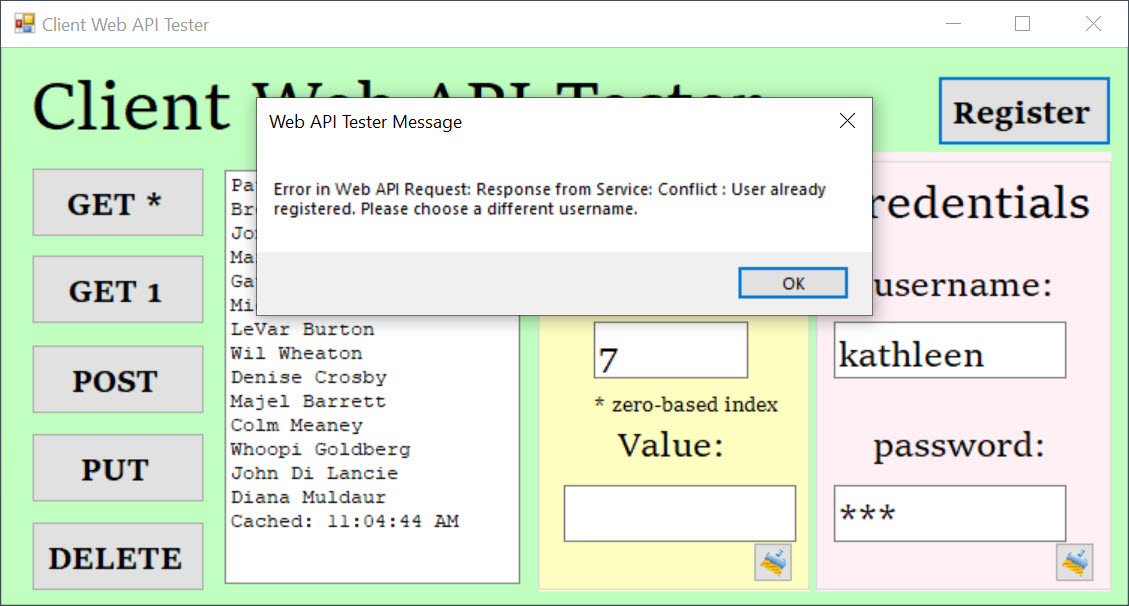
1. After Get(1) with the index specified



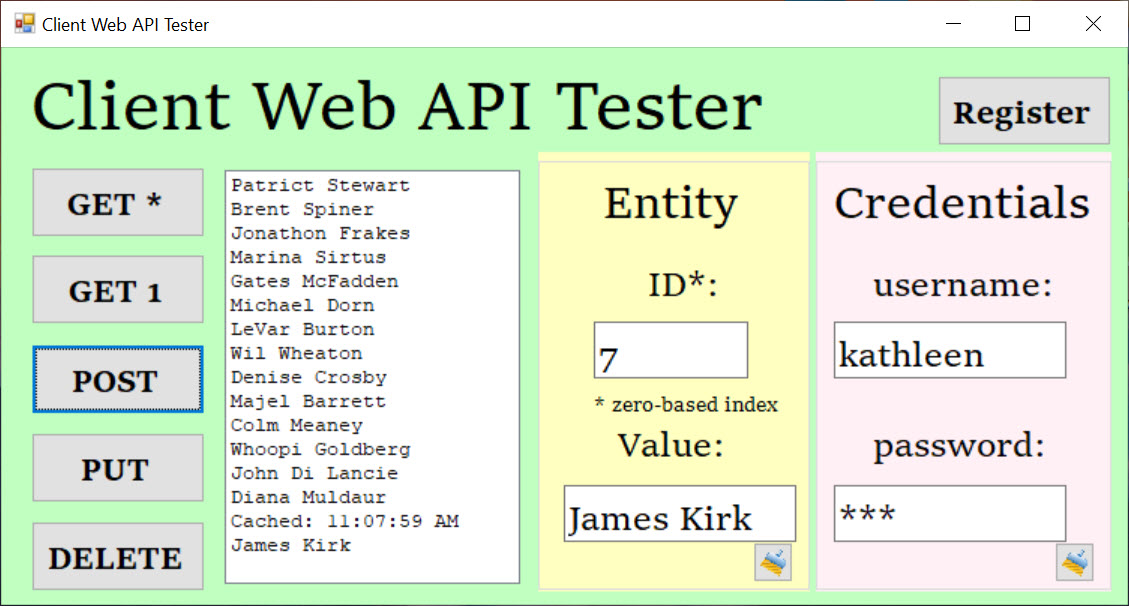
1. Registration – Successful



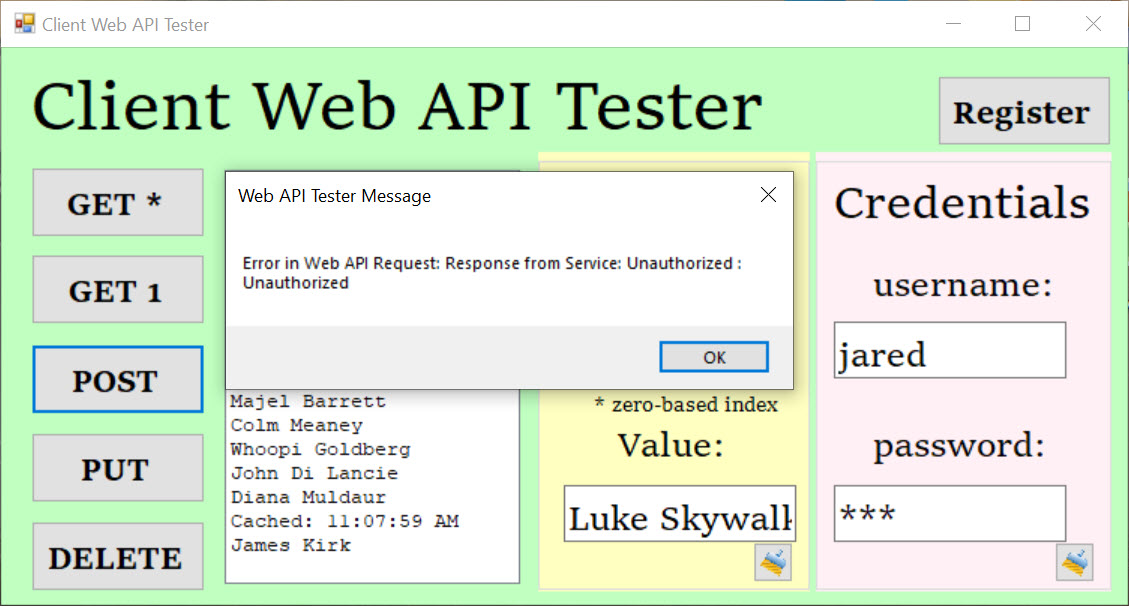
1. Registration – Invalid



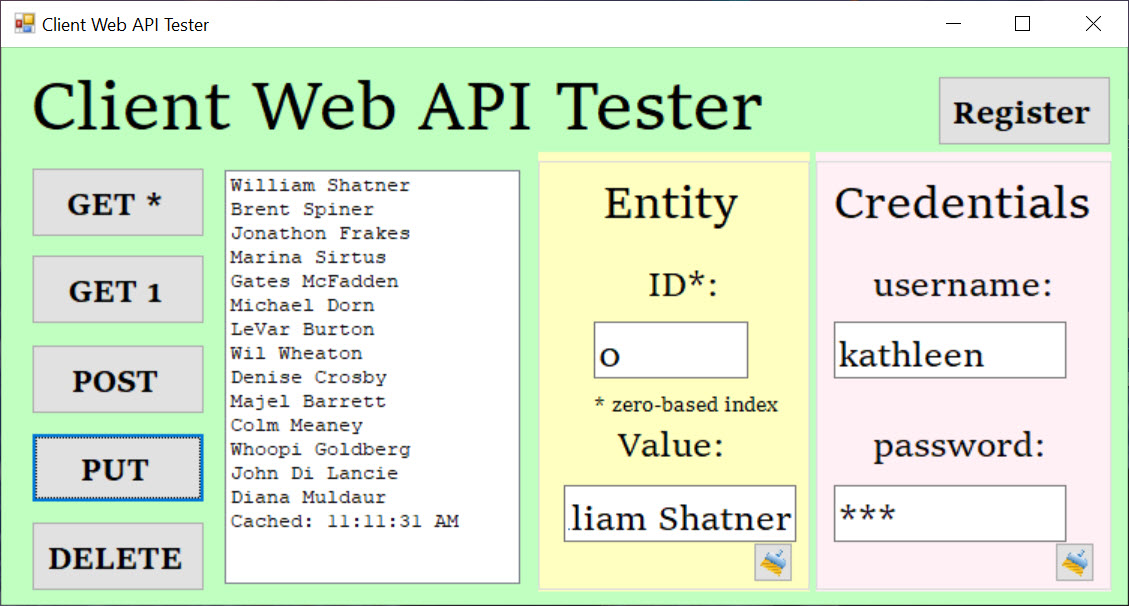
1. After Post with the Value specified, Authorized (Authenticated) User



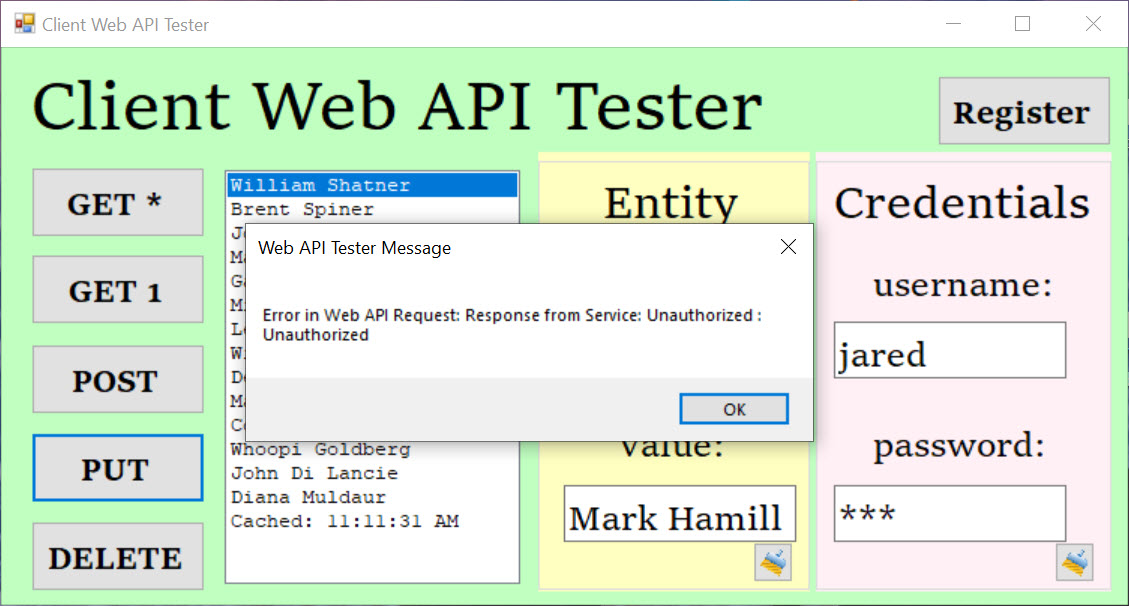
1. Post with the Value specified, Unauthorized (Authenticated) User



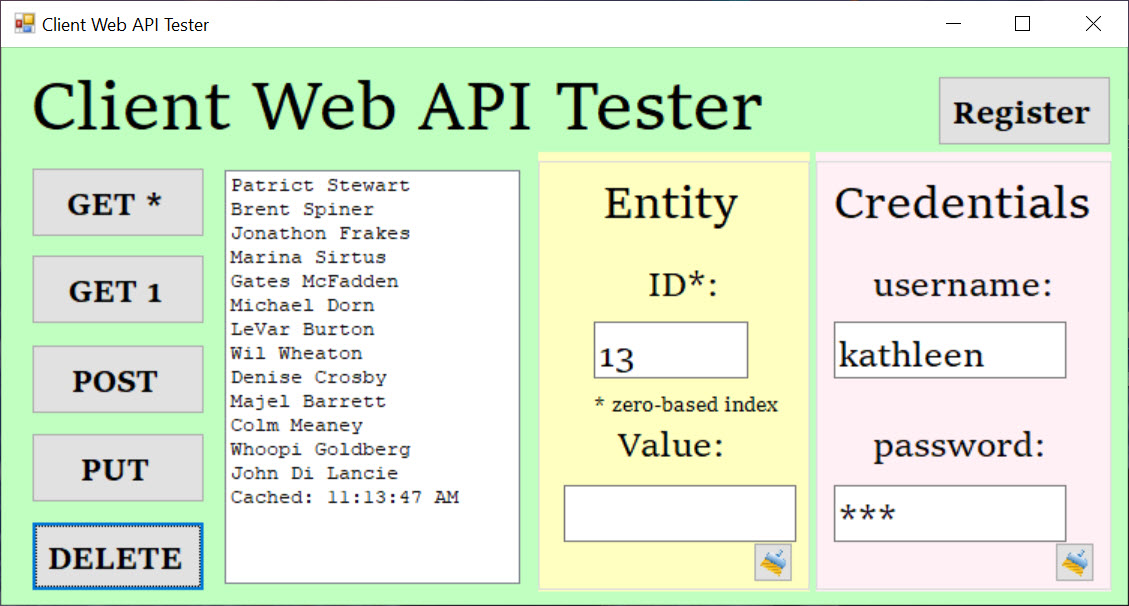
1. After Put with the Index and Value specified, Authorized (Authenticated) User



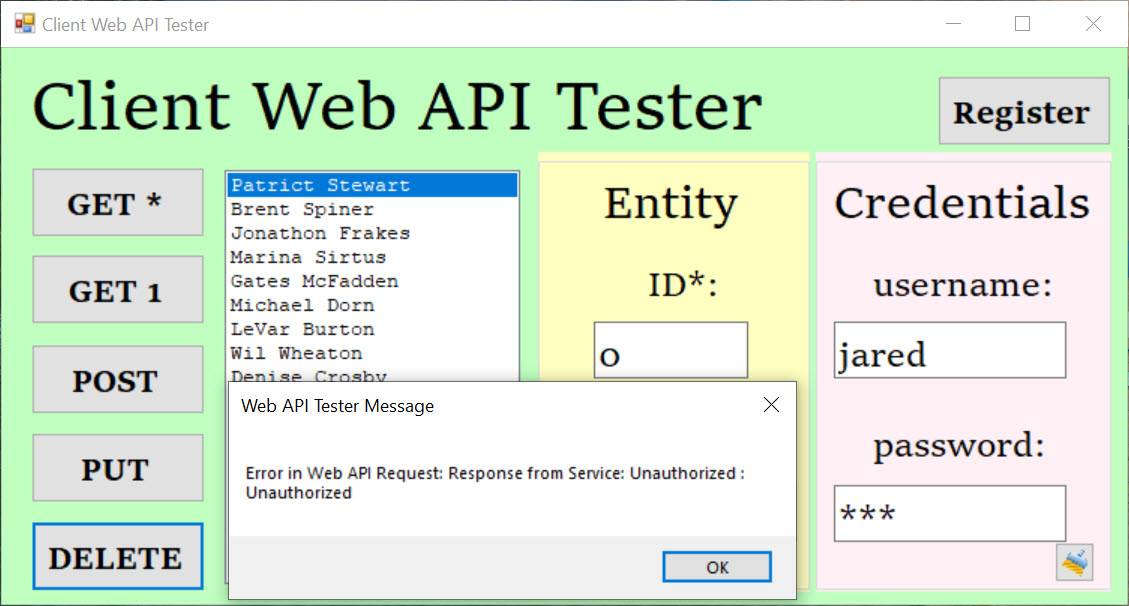
1. Put with the Index and Value specified, Unauthorized (Authenticated) User



1. After DELETE with the Index specified, Authorized (Authenticated) User



1. After DELETE with the Index specified, Unauthorized (Authenticated) User



## Code

The entire project code repository is available on GitHub [here].