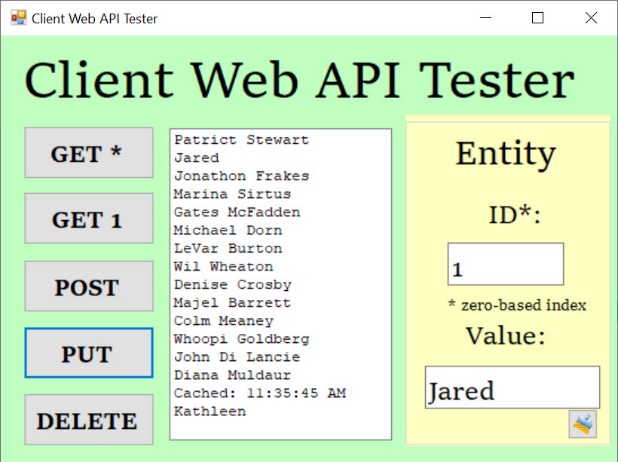
# ASP.Net Web API Memory Cache Service Application + Client Tester CRUD (Get, Post, Put, Delete) Windows Form Application

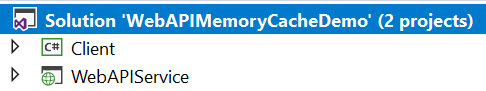
## 

## About

This project presents a Visual Studio solution including a simple demo ASP.Net Web API 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). In addition to demonstrating standard CRUD capabilities, the Web API service implements a .Net Memory Cache (MemoryCache). The client “tester” application also allows the user to verify that the memory cache is implemented correctly and expiring per the set policy. Discussion of the memory cache implementation may help the reader with tips to understand, correctly implement, and verify the cache is expiring. Lastly, the project is shown in the demo section with a video and screen captures.



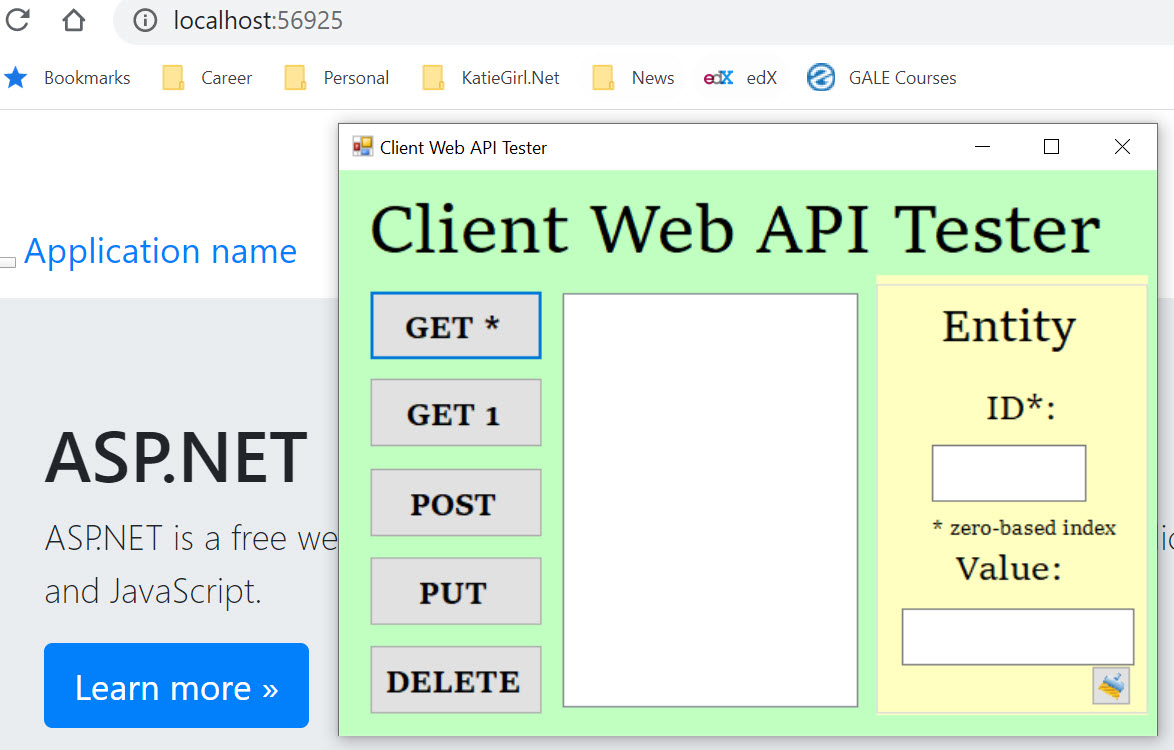
## Architecture



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



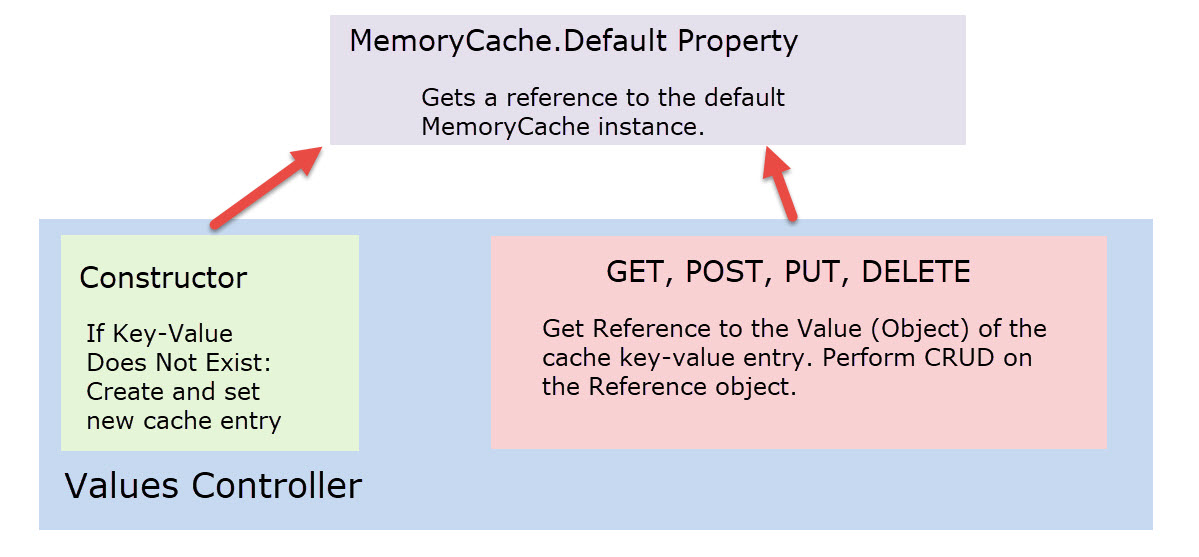
The ASP.Net Web Application template was used to create a simple Web API service and was hosted on my local development machine using IIS Express launched from Visual Studio. The API functions will be discussed after the Memory Cache approach. The code is available on GitHub [here].

### Memory Cache “MemoryCache” .Net Discussion and Notes

Before I discuss the implementation code, I wanted to mention some tips and words of wisdom that I have learned working with the .Net MemoryCache in my Web API implementation [doc link: <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.caching.memorycache?view=netframework-4.8>] that may help you in your project.

1. When the client calls the Web API for any request, the service creates a unique controller object instance to service each of the client’s requests (similar to PerCall in WCF). The class will be setup for instance objects unless you otherwise mark it static. My Web API implementation, discussion, and notes assumes instances not a Singleton service.
2. Because each Web API call spins up a new instance, the Values controller constructor is called every time the client calls the Web API (see note above). This affects the approach of implementing a memory cache.
3. “MemoryCache.Default” is the default instance of the cache and can be shared by all client callers to the service [see doc: <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.caching.memorycache.default?view=netframework-4.8>]. So, in order to utilize the Memory Cache, the controller should create an object that references this default instance object. Here is a code example…
4. Each client call to the Web API spins up a new Values controller object instance with its own cache object (newly created each call) that references the default “shared” cache.
5. The Values controller constructor executes first before any API call (example: Get). We can verify if there is an existing key-value pair (like a Dictionary) in the default cache upon the constructor load.
6. If the default cache does not contain the specified key (string) and value (can be object) for our inquiry, we first create one (key) and add it to the default cache with its value.
7. We can add the key-value pair to the default cache with a policy object (Expiration) using the cache.Set method [see doc here: <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.caching.memorycache.set?view=netframework-4.8>].
8. Once the policy is set in the constructor to create the MemoryCache object…. And you’ve created the object… you cannot retrieve the policy object. There is not much you can do with a MemoryCache object to retrieve information about its previously set policy or when it’s going to expire after it has been constructed with a policy object (I think this would be a good feature enhancement to add a policy object as a property to the MemoryCache). I studied the current Microsoft (.Net Framework) docs to come to this conclusion, and if I’m wrong, please reach out to me and explain why and how you can retrieve the policy information from a MemoryCache object.
9. Once the cache key-value pair is constructed with a policy, it is stuck with that policy until either it expires (the policy expiration property) or the key-value pair is modified with a new policy or modified without a specified policy object (defaults to non-expiring or when the App pool needs to recycle).
10. Using the indexer method to set or modify your cache key-value … very important…. The expiration policy will be changed to an InfiniteAbsoluteExpiration [see reference doc here: <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.caching.memorycache.item?view=netframework-4.8>]
11. Using the cache.Set method to set or update the key-value in the cache allows for including a policy object that controls the expiration. Remember note # 8 above, means you have to create a brand-new policy object with your expiration settings. When you use the cache.Set method you are effectively resetting the policy of the existing key-value entry in the cache.

#### My MemoryCache Implementation Approach



My approach for implementing a Memory Cache was to create a cache object for each Web API call that points to the default cache (single instance). At the start of each client call to the Web API it first calls the Values Controller (discussed in the next section) constructor before the specific request method (ex: Get). In the controller constructor, it checks to see if the cache object (pointing to the default instance) has a specific key-value pair: a key of “People” and a value (reference object) of List<string> representing a list of characters from Star Trek TNG. If the cache does not contain the key “People” it will first create a default List<string> of characters from Star Trek TNG and then set the cache with the key-value pair.

Next the specific Web API request method is called (GET, POST, PUT, or DELETE) in the Values controller. The CRUD actions are done on the cache object by reference. This means that the CRUD operates do not directly modify (indexer) or set (cache.Set) the key-value cache entry. This allows the original key-value entry in the cache to retain its original policy and expiration settings (see notes above). Instead, each CRUD application copies a reference to the key-value entry in the cache (a List<string> is a reference type) and does the CRUD operations on the object reference. Thus, there is no need to utilize the indexer or cache.Set methods of adding/updating the cache key-value entry directly. As previously mentioned, the reason why we do not want to modify the cache key-value entry directly (either with indexer or cache.Set) is because we would be overwriting the original policy and expiration that was set when it was originally created.

### Values Controller

This values controller is the generic default Web API controller implementation and I did not rename or do any fancy 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 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.

[insert code here]

#### Get (Collection)

The Get (Collection) retrieves the key-value entry from the cache and then returns an IEnumerable collection to the caller.

[insert code here]

#### Get (Entity Id)

The Get (Single Entity) first retrieves the key-value entry 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.

[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.

[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.

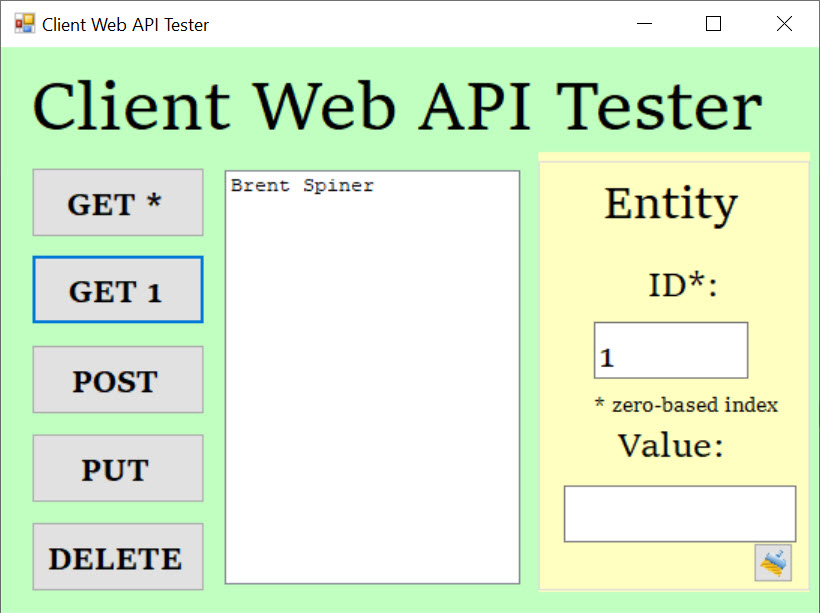
[insert code here]

#### 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.

[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 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.

### 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
* Path to Web API Service

#### Methods

##### Main

This main method is the entry point of the client tester application. It sets up the HttpClient object with the Web API Values Controller address path.

##### Events

There are multiple button click events to handle the user requests. 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.

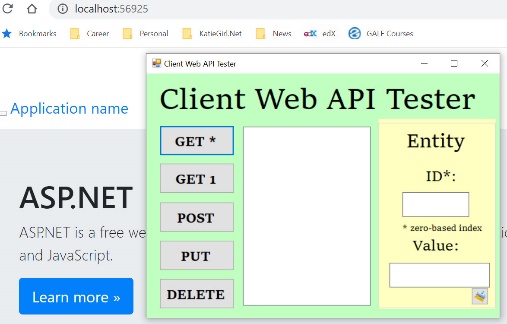
## Demo

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

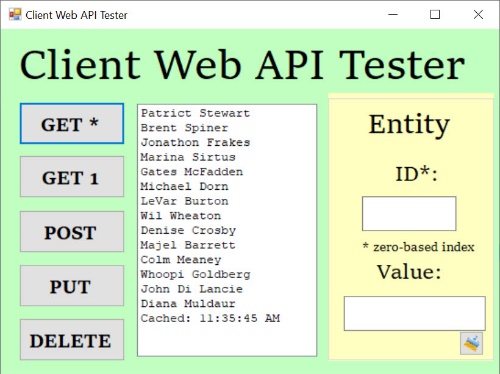
[insert YouTube video]

Screen captures are shown below for various user activities:

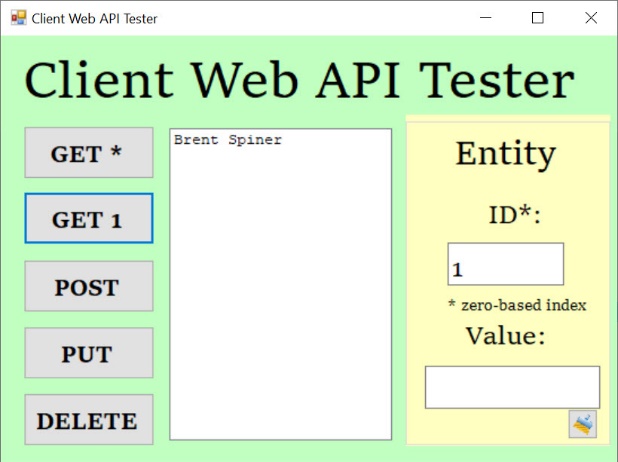
1. After all projects are loaded from Visual Studio



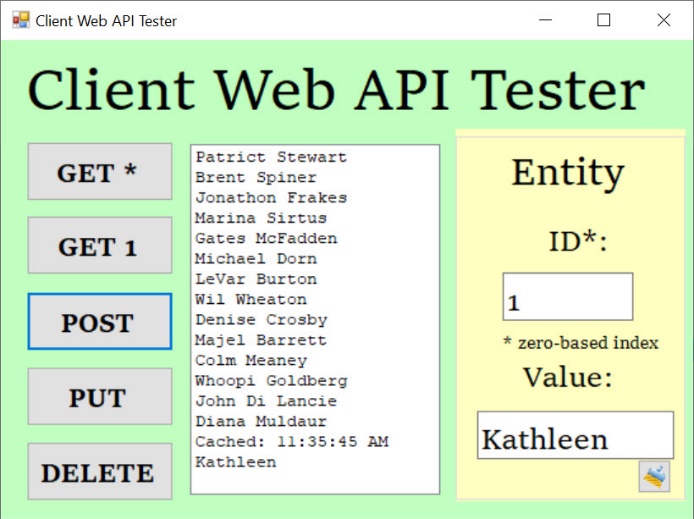
1. After Get(\*)



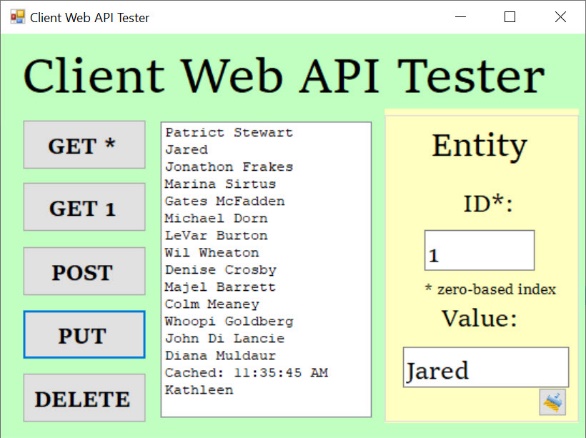
1. After Get(1) with the index specified



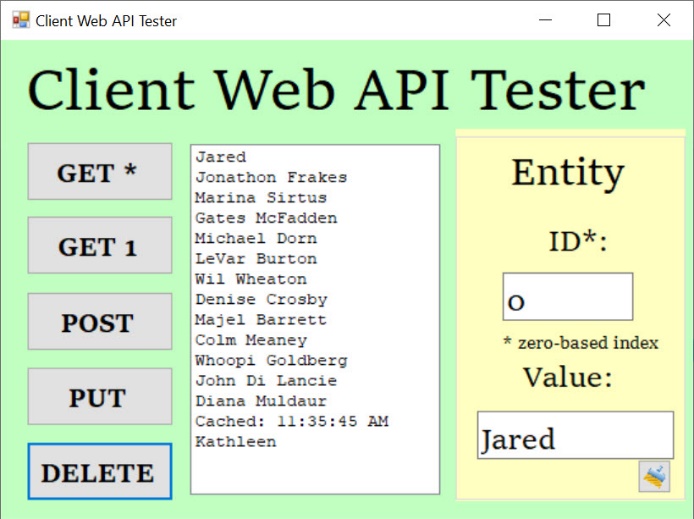
1. After Post with the Value specified



1. After Put with the Index and Value specified.



1. After DELETE with the Index specified



## Code

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