

SOFT 338 – Assignment 2

Distributed API Documentation

1.0 Introduction

The chosen topic area surrounding the distributed API for this assignment was events. An Event object is comprised of an eight character long identifier (a reference), a name, address, postcode, date and time.

An application has been implemented that provides an easy way to test the distributed API by providing a combo box that is populated with CRUD operations and a list view to visualise the changes made to objects that are retrieved from the local database. This will be referred to throughout the report as the “client application”. The output format for the responses written to the output stream is JSON.

There are several sections that make up the documentation for this assignment, which includes both the server-side distributed API, as well as the client application. This report features UML class diagrams, a section describing the distributed API's calls and a programmatic walkthrough of both the distributed API and how to use the client application for retrieving, deleting, updating and creating a new instance of an Event.

With reference to the criteria for the assignment that relates to a third-party API call, it is worth mentioning that a resource has been dedicated to “extracting” and outputting the result of a query to the Google Places API, which returns information relating to addresses. The “consuming” criteria is covered by an optional parameter of the distributed API's CreateEvent method for an object; more information for this can be found in section 3.

Table of Contents

SOFT 338 – Assignment 2	1
1.0 Introduction	1
2.0 Overview and design of distributed-API and client-side application.....	3
2.1 Distributed API	3
2.1 Client-side application	4
3.0 Using the API: reference section.	5
3.1 Error handling and status codes.....	5
3.1 Resource: Event	6
3.1 Resource: search_places.....	11
4.0 Distributed API Walkthrough.....	12
5.0 Testing the distributed API.....	17
5.1 Client-side application - overview.	17
5.2 Testing walkthrough (inserting data, using the client-side application and visualising changes).....	18
5.3 Postman.....	19
5.0 Bibliography	20

2.0 Overview and design of distributed-API and client-side application.

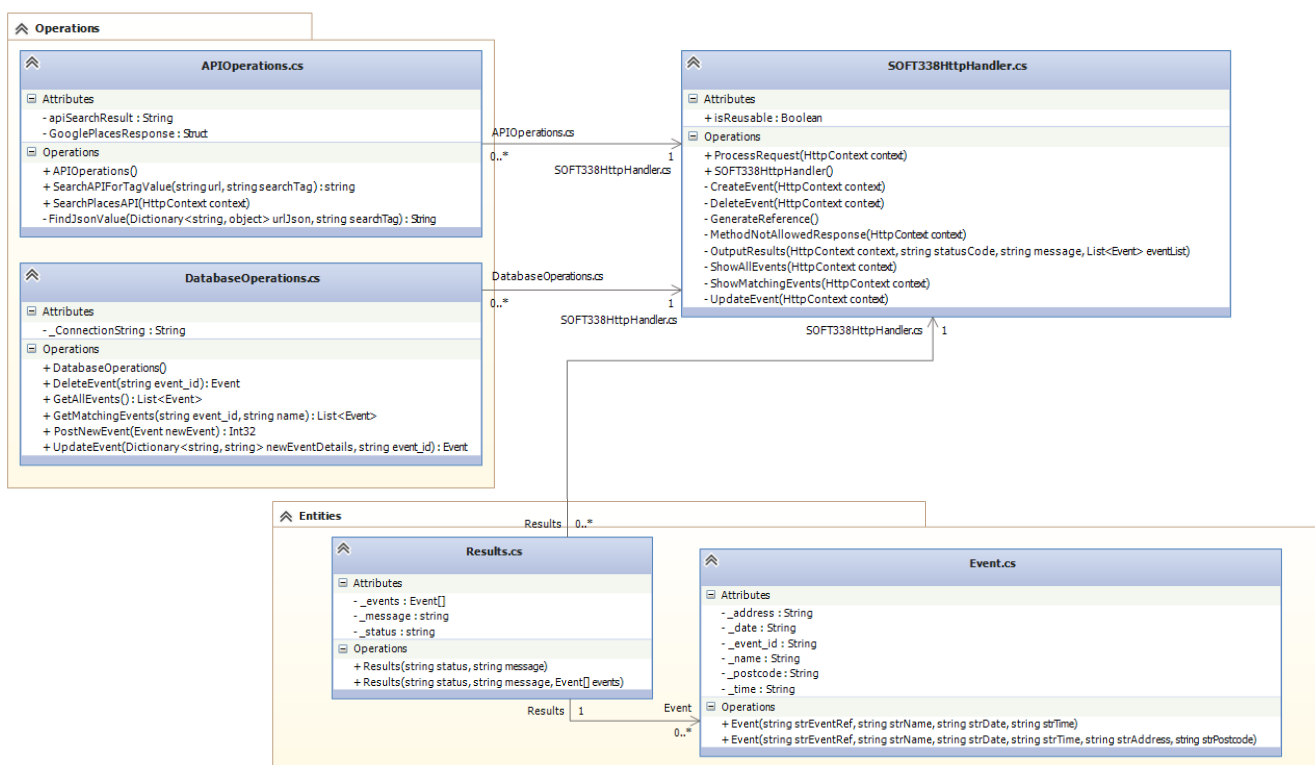
This section describes the thought processes and methods used to construct both the distributed API and the client application that was constructed to test it.

2.1 Distributed API

There are two resources for this assignment submission, which are events and search_places; the first allows for the responses of CRUD related operations to be written to the output stream and the second extracts information from a third-party API. When a request is sent to the distributed API, a method is called depending on the type of resource and query string provided.

The responses for CRUD operations are written to the output stream using an instance of the class Results, which contains an array of Events, a status and finally a message that informs the client of what would be needed to fix an encountered problem – for example, feedback about a missing parameter.

There are two different constructors for a Results object, and one of the two is purely used to output statuses and messages. This is explained further in the section that describes the functionality of the distributed API (section 4); the methods, different types of statuses and messages.



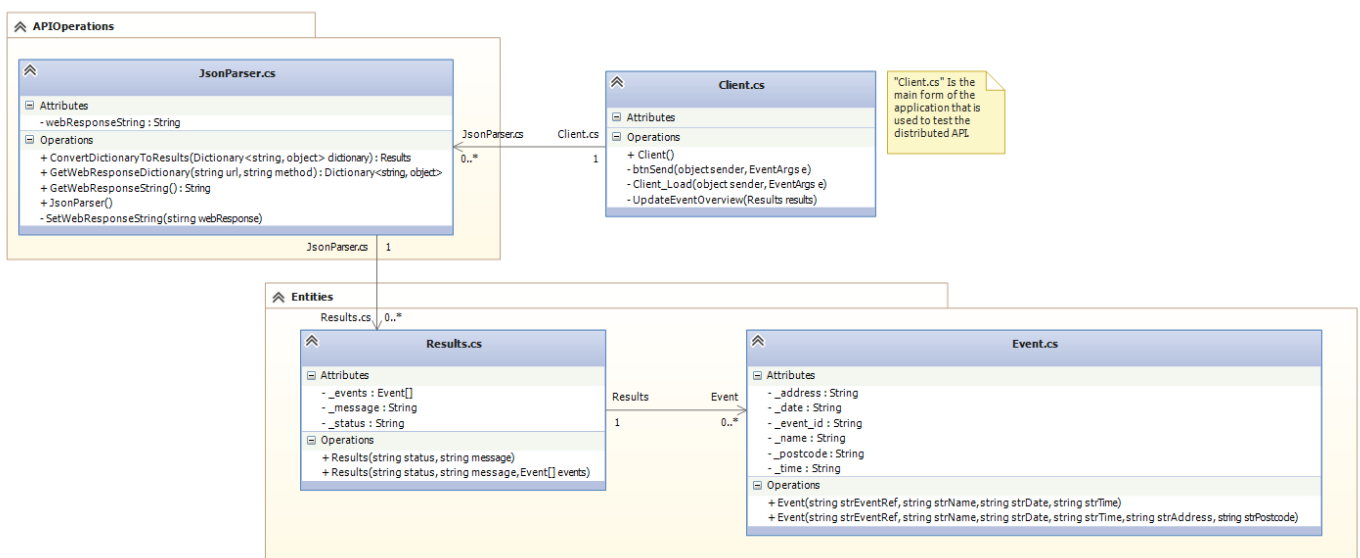
2.1 Client-side application

A simple application has been made in c# which is comprised of a single form that has certain features that postman provides (such as creating HTTP responses) but includes specifically the ability to illustrate the effect that a specific API call has on one or more of the Events stored in the local database. An example of this would be deleting an Event, where the event_id is provided and the request is sent to the server. Once it has been completed the Event will be removed from the list view.

To add to this, the application has been designed to ensure that all available API calls are available from a combo box which results in only the query string requiring manual input.

Once a URL has been constructed and the query string has been added to it, an API call is sent and the response is collected as a dictionary of objects which are identified by string keys. When this response is parsed into a format that matches the constructor for an instance of the Results class, it can be used to provide feedback in the sense that it is added to a specific form object such as a list view or a text box.

It is worth noting that all CRUD related operations only comprise a single instance of an API call. Therefore (as an example) when deleting an Event and viewing the changes made on the list box, the client application sends two requests; one to delete the specific Event, and another to retrieve all of the Events for the purpose of updating the list view to show the changes that have been made.



3.0 Using the API: reference section.

This section will serve as a guide to those that want to learn how to use the distributed API. It will describe the different types of calls that are currently available which provide CRUD and external API functionality for events using the local database allocated for the SOFT338 module. The description for each call will include the URL structure, required and optional parameters and examples that can be used or adopted.

As a side note: If the client application is used to test the calls documented in this section, then knowing the structure of the URL is not a requirement, since the structure is programmatically generated.

If Postman is being used however, the URL structure is as follows:

```
http://xserve.uopnet.plymouth.ac.uk/modules/soft338/astevenson/ + call + query string
```

For example, to retrieve all events using Postman that match the name "Movie Night":

```
http://xserve.uopnet.plymouth.ac.uk/modules/soft338/astevenson/events/show?name=Movie Night
```

If needed, please refer to section 5.2 for a walkthrough on how to use the client application for creating, retrieving, deleting and updating a new Event.

3.1 Error handling and status codes

Statuses are returned as strings (much like the Google Places API) and contain a brief description about the outcome of the API call. If there is an error, the message that accompanies it can provide information about how to resolve it.

The following table lists the different types of statuses and - in the case where an error is encountered - some of the reasons behind why an error may occur, together with the likely reason behind it:

Status	Description
OK	There has been no error. The server has understood and interpreted the request and has completed it successfully.
INVALID_REQUEST	There are several reasons for why this status may be returned: 1) If a required parameter is missing for a query string. 2) No parameters are given where at least one needs to be specified. 3) The format of the date is incorrect. 4) The method used for the request is incompatible with the type of operation. For example, POST needs to be used to create a new event.
NO_RESULTS_FOUND	The request has been completed successfully, but there were no results that matched the query, or there are currently no events in the local database.

3.1 Resource: Event

3.1.1 /events

DESCRIPTION: Retrieves all events from the local database.

URL STRUCTURE: <http://xserve.uopnet.plymouth.ac.uk/modules/soft338/astevenson/events>

METHOD: GET

PARAMETERS: No parameters are required.

RETURNS: All events stored in the local database.

Example JSON response:

```
{
  - "events": [
    - {
      "address": "Fairmead Ave, Benfleet, Essex SS7 2UJ, UK",
      "date": "10\8\2015",
      "event_id": "loNSHYir",
      "name": "Adams Birthday",
      "postcode": "SS72UJ",
      "time": "13:00"
    },
    - {
      "address": "12 Hawks Park, Saltash, Cornwall , UK",
      "date": "18\5\2016",
      "event_id": "QqMHjave",
      "name": "Movie Night",
      "postcode": "PL124SP",
      "time": "18:00"
    }
  ],
  "message": "Retrieved results successfully",
  "status": "OK"
}
```

3.1.1 /events/show

DESCRIPTION: Retrieves all Events that match a specified name or event_id.

URL STRUCTURE: `http://xserve.uopnet.plymouth.ac.uk/modules/soft338/astevenson/events/show`

METHOD: GET

PARAMETERS: **(Optional) event_id:** The eight character identifier for the Event.

(Optional) name: The name of the Event.

RETURNS: All Events that matches either the event_id or name.

Example URL:

`http://xserve.uopnet.plymouth.ac.uk/modules/soft338/astevenson/events/show?name=Movie Night`

Example JSON Response:

```
{
  - "events": [
    - {
      "address": "12 Hawks Park, Saltash, Cornwall , UK",
      "date": "18/5/2016",
      "event_id": "QqMHjave",
      "name": "Movie Night",
      "postcode": "PL124SP",
      "time": "18:00"
    }
  ],
  "message": "Events retrieved successfully",
  "status": "OK"
}
```

3.1.1 /events/create

DESCRIPTION: Creates a new instance of an Event.

URL STRUCTURE: `http://xserve.uopnet.plymouth.ac.uk/modules/soft338/astevenson/events/create`

METHOD: POST

PARAMETERS:

- (Required) name:** The name of the event.
- (Required) date:** The date in the format: dd/m/yyyy.
- (Required) time:** The time of the event.
- (Optional) address:** The address of the event.
- (Optional) postcode:** The postcode of the event.
- (Optional) find_address:** Uses an external API (Google Places) to automatically find additional information for the address field. To use this feature, this value needs to be set to "yes".

RETURNS: The information of the new event; a status and message confirming that it has been added to the local database.

Example URL:

```
http://xserve.uopnet.plymouth.ac.uk/modules/soft338/astevenson/events/create?name=Movie Night&time=19:00&date=20/02/2015&address=12 hawks park&postcode=pl124sp&find_address=yes
```

Example JSON Response for /events/create:

```
{
  - "events": [
    - {
      "address": "12 Hawks Park, Saltash, Cornwall , UK",
      "date": "20/2/2015",
      "event_id": "P4z0qzj9",
      "name": "Movie Night",
      "postcode": "PL124SP",
      "time": "19:00"
    }
  ],
  "message": "Event has been created successfully",
  "status": "OK"
}
```


3.1.1 /events/delete

DESCRIPTION: Deletes an event from the local database.

URL STRUCTURE: `http://xserve.uopnet.plymouth.ac.uk/modules/soft338/astevenson/events/delete`

METHOD: DELETE

PARAMETERS: **(Required) event_id:** The eight character long identifier for the event.

RETURNS: The information of the event that has been deleted in addition to a confirmation message and status.

Example URL:

`http://xserve.uopnet.plymouth.ac.uk/modules/soft338/astevenson/events/delete?event_id=Oi4Lab7X`

Example JSON Response:

```
{
  "events": null,
  "message": "The event has been successfully deleted. The details were: name = 'Movie Night', date = '20/2/2015', time = '19:00', address = '12 Hawks Park, Saltash, Cornwall , UK', postcode = 'PL124SP'",
  "status": "OK"
}
```

3.1.1 /events/update

DESCRIPTION: Updates an event in the local database with one or more new details.

URL STRUCTURE: `http://xserve.uopnet.plymouth.ac.uk/modules/soft338/astevenson/events/update`

METHOD: PUT

PARAMETERS:

- (Required) event_id:** The eight character long identifier for the event.
- (Optional) name:** The new name of the event.
- (Optional) time:** The new time of the event.
- (Optional) date:** The new date of the event.
- (Optional) address:** The new address of the event.
- (Optional) postcode:** The new postcode of the event.

RETURNS: The newly edited event along with the message and status.

Example URL:

`http://xserve.uopnet.plymouth.ac.uk/modules/soft338/astevenson/events/update?event_id=Oi4Lab7X&name=Changed Movie Night`

Example JSON Response:

```
{
  - "events": [
    - {
      "address": "12 Hawks Park, Saltash, Cornwall , UK",
      "date": "20\2\2015",
      "event_id": "Oi4Lab7X",
      "name": "Changed Movie Night",
      "postcode": "PL124SP",
      "time": "19:00"
    }
  ],
  "message": "Event has been updated successfully",
  "status": "OK"
}
```

3.1 Resource: search_places

3.1.1 /search_places

DESCRIPTION: Provides calls to the Google Places API. This is used to collect information relating to events. For the event/create call, this can optionally be used to help construct the address field.

URL STRUCTURE: `http://xserve.uopnet.plymouth.ac.uk/modules/soft338/astevenson/search_places`

METHOD: GET

PARAMETERS:

- (Required) query:** The searching string that will be used to retrieve places from the external API.
- (Optional) tag:** Retrieves the value of the tag specified (for example, formatted_address).

RETURNS: Either the array of address results or the specific value of the tag which has been specified.

Example URL:

`http://xserve.uopnet.plymouth.ac.uk/modules/soft338/astevenson/search_places?query=hawks park pl12&tag=formatted_address`

Example JSON Response:

```
{
  "message": "Collected the result string for the external API call.",
  "results": "Hawks Park, Saltash, Cornwall PL12, UK",
  "status": "OK"
}
```

4.0 Distributed API Walkthrough

This walkthrough has been designed to describe and illustrate the programmatic flow of the distributed API that has been created and submitted for this assignment. The UML class diagram in section 2 can be used to navigate the application.

The chosen example for this walkthrough is the creation of a new Event. It should be thought that the programmatic flow for independent API calls are very similar in terms of the navigation between classes. For more information on the different calls available, please refer to section 3.

The programmatic starting point for this walkthrough is when a request is made to the server. When a call is made to the API, the web configuration file determines what handler will be used based on the type of resource that is detected. The HTTP verb used in conjunction with the specified URL structure determines what the output is; the method for stripping down the request's path has been taken from the module practical's, but alternatively URL templates could have been used. The method "ProcessRequest" is used for this purpose (In the class SOFT338HttpHandler).

If a resource is used but the pathway does not match any case statement, the default behaviour of the application is to create an instance of a "Results" object that is written to the output stream to inform the client of the error.

```
public void ProcessRequest(HttpContext context)
{
    // Retrieve and shorten the path
    string strPath = context.Request.Path.Replace("/modules/soft338/astevenson/", "");

    switch (strPath.ToLower())
    {
```

(More code in-between)

```
        case "events/create":
            switch (context.Request.HttpMethod.ToLower())
            {
                // Accepted method: post
                case "post": CreateEvent(context); break;

                // Anything besides accepted method, respond with a 405 status code
                default: MethodNotAllowedResponse(context, "post", context.Request.HttpMethod.ToLower()); break;
            }

            break;
```

The method "MethodNowAllowedResponse" is invoked when the HTTP verb used is not accepted. If this is not an issue, the application proceeds to the next step which (in this case) is invoking the "CreateEvent" method which is designed to add a new instance of an event into the local database.

The new event is only added after validating the query strings and ensuring that the date is in the correct format. The time variable has no stipulation with regards to formatting because the client could provide an input value that would be acceptable in several different contexts (for example, 3pm and 15:00).

If validation detects an error then the method “OutputResults” is invoked, which is modular in the sense that it is invoked by all of the other CRUD related methods for the same reason; to provide feedback to the client.

```

/// <summary>
/// This method creates an output stream which provides details to the client about the Events, status codes and messages
/// for the Http Response.
/// </summary>
/// <param name="context">The specific http context</param>
/// <param name="statusCode">The status of the response. For example "OK".</param>
/// <param name="message">A message to provide feedback to the client</param>
/// <param name="eventList">The array of events that will be written to the output stream.</param>
private void OutputResults(HttpContext context, string statusCode, string message, List<Event> eventList)
{
    // Create a serialiser object of type "Results".
    // This is comprised of a list of Events, a status code (for example OK in cases where an operations has been
    // completed successfully) and finally a message for the client.
    DataContractJsonSerializer jsonData = new DataContractJsonSerializer(typeof(Results));

    // Create the output stream - which will provide the result in a json format
    Stream outputStream = context.Response.OutputStream;

    // Construct an instance of a Results Object (Status Code, Message, (optional) List<Event>)
    Results results;

    // Determine the type
    if (eventList.Count > 0)
        results = new Results(statusCode, message, eventList.ToArray());
    else
        results = new Results(statusCode, message);

    // Write the response to the output stream
    jsonData.WriteObject(outputStream, results);
}

```

Once validation has been checked for the “CreateEvent” method, a reference is generated for the Event. This is the eight character long identifier that is required for the update and delete calls to the distributed API and uniquely identifies each event. It is unlikely that there will be several instances of objects that have the same generated identifier, but not impossible. Future alterations can be made to the application to account for this, such as increasing the size of the identifier (at the cost of how easy it is to use) or providing an alternate parameter for the DeleteEvent method that serves as an additional “WHERE” operator for the database related command.

The reference (or event_id) is generated using the “GenerateReference” method:

```

/// <summary>
/// Generate a random eight character long string, which will be used as the reference
/// for the Event or Member
/// </summary>
/// <returns>An eight character long string that is the identifier for the event</returns>
private string GenerateReference()
{
    // Create the new identifier for the event/member ( a reference )
    // Source: http://stackoverflow.com/questions/1344221/how-can-i-generate-random-alphanumeric-strings-in-c
    // The characters that will be used to form the key identifier
    var characters = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";

    // The container for the randomly generated string; the element size of the array defines the length
    char[] randomChar = new char[8];
    var random = new Random();

    for (int i = 0; i < randomChar.Length; i++)
    {
        // Until we reach the end of the length of stringChars,
        // append to the string a random character from the characters variable
        randomChar[i] = characters[random.Next(characters.Length)];
    }

    string reference = new String(randomChar);

    return reference;
}

```

Once the identifier has been generated for the new event, the query string “find_address” is checked to determine if the Google Places API will be used to contribute to the construction of the “address” variable which is one of the parameters of a new “Event” object. If this option is used, a URL structure is determined and will be used to search for the new value of the address variable. The new address value will be the string value of the tag “formatted_address” which can be collected from the response of an external call to a third party API.

```
if (useGooglePlaces.ToLower() == "yes")
{
    // use the third party api search to "consume" the response
    // first, store the url string that will be sent to and used by the methods contained
    // within the "APIOperations" class.
    string url = "https://maps.googleapis.com/maps/api/place/textsearch/json?query=" + address + " " +
        postcode + "&key=AIzaSyDU3ZDN-wKRktsn7SgUwOrLFxe-GG1u1hc";

    // Retrieve the json response from the webclient request to Google Places
    // It returns (in this case) the response value as a string, for the
    // tag "formatted_address". However, it can be changed to retrieve the value
    // from any tag (as long as it is there in the json response).
    APIOperations operations = new APIOperations();
    address = operations.SearchAPIForTagValue(url, "formatted_address");

    // Check to see if the postcode is contained in the response, if it is, then replace it
    // with nothing
    // first remove the spaces for the postcode
    // and remember that strings are immutable in c#. So the left value assigning is required
    postcode = postcode.Replace(" ", "");

    // and make it upper case and add a space in the right place
    postcode = postcode.ToUpper();

    int length = postcode.Length;
    string refinedPostcode = postcode.Substring(0, 4) + " " + postcode.Substring(4, length - 4);

    // remove the postcode from the returned "formatted_address" tag response
    if (address.ToUpper().Contains(refinedPostcode.ToUpper()))
        address = address.Replace(refinedPostcode.ToUpper(), ""); // strings are immutable, so requires reassigning
}
```

To specifically collect the value of the new address, the method “SearchAPIForTagValue” is invoked which belongs to the class “APIOperations”.

```
private string apiSearchResult = ""; // for searching using a third party API.

///<summary>
/// This method is primarily used to return the value of an external API call that corresponds to a specified tag.
/// For example, formatted_address may contain habitual information, and this method returns the first responses ArrayList
/// value for that specified tag. Admittely, this could be improved by returning an array of sorts that contains multiple
/// values, but validation can ensure that the correct address (in this example) can be found and used.
///</summary>
/// <param name="url">The url that will return json information.
/// This requires that the content-type of the current request is application/json</param>
/// <param name="searchTag">The name of the JSON Tag that is being searched</param>
/// <returns>The value of the searched tag</returns>
public string SearchAPIForTagValue(string url, string searchTag)
{
    apiSearchResult = ""; // reset search variables - if this method is used more than once by an instance of this class
    string searchResult = "";

    //
    // Collect and parse the response from the third party API
    //
    var webClient = new System.Net.WebClient();
    var jsonString = webClient.DownloadString(url);

    if (jsonString != null)
    {
        JavaScriptSerializer serialiser = new JavaScriptSerializer();
        Dictionary<string, object> jsonDictionary = serialiser.Deserialize<Dictionary<string, object>>(jsonString);

        //
        // Collect the result of the search
        //
        searchResult = FindJsonValue(jsonDictionary, searchTag);
    }

    return searchResult;
}
```

An instance of a “WebClient” is created which downloads the response from Google Places and contains it within a string. A Dictionary is constructed that is a compilation of different objects which are identified by different string values (or keys). In this particular instance,

these string identifiers contain objects that are most commonly strings, ArrayLists and other dictionaries.

In order to collect the value of the specified tag that has been passed as a parameter, the "FindJsonValue" method is invoked. The method is designed to look at each key within a dictionary and determine what logic to perform based on the type of object found. If the tag value that is being searched is found (In this instance it would be "formatted_address"), the value of the string is stored and is returned, which indicates a success. Multiple instances of this method are most likely to be active at once, because it is invoked whenever another dictionary is found.

A private global string variable has been constructed which is used to store the value of the tag. It is important for it to be global because each time a dictionary is found another instance of this method is called. If the variable was local, there would be more than one return value which would be blank and the final return value would determine the result of the entire operation.

When these methods have completed, if the value is found it is returned and the third party API call to change the value of the address variable to that of the value of the tag "formatted_address" is completed.

An instance of an "Event" object is created and instantiated with the query string and address values. A "DatabaseOperations" object is created that contains all of the methods that use a connection string to establish a connection to the local database for the purpose of performing CRUD related operations.

```
// use the event object to write to the database
DatabaseOperations.PostNewEvent(e);

// Output the new event
// Create an array of events and make it consist of the
// Event that was added to the local database
eventList.Add(e);

// Write the results to the output stream
OutputResults(context, "OK", "Event has been created successfully", eventList);

// change the status code to OK
context.Response.StatusCode = (Int32)HttpStatusCode.OK;
```

Finally, the "PostNewEvent" method is invoked and adds a new row to the database using the values of the event that have been passed by reference as a parameter. The new event is then added to the list of events that was created at the start of the "CreateEvent" method and the "Results" are written to the output stream.

It should be presumed that the programmatic logic that underpins the other CRUD operations that have not been covered in this walkthrough are similar in terms of the flow between different methods. In each case the query strings are validated and any other pre-emptive errors are accounted for.

As a brief overview, some of the major differences between the different calls are as follows:

Updating an event (“UpdateEvent” in “SOFT338HttpHandler”) ensures that the required event identifier parameter is present, along with one or more optional parameters that serve the purpose of changing specific values for an “Event” object. A for loop is used that concatenates a string with one or more SET clauses – with reference to the amount of changes that need to be made.

The search_places resource follows the same logic as above, except in this instance a private structure is used and instantiated with the exact response from “Google Places”, a message and a status; refer to the “APIOperations” class to see this.

5.0 Testing the distributed API.

5.1 Client-side application - overview.

An application has been designed and created for the purpose of making the testing process easier. The application (SOFT338APIClient) requires that Visual Studio 2012 is installed and should run on both 32 and 64 bit operating systems.

In order to test out a specific API call, all that is required is that a specific selection is made from the “URL” combo box and that the query string is provided.

As an example, to retrieve all events that match the name “Movie Night”:

- 1) Select events/show from the “URL” combo box.
- 2) Add the query string “?name=Movie Night” to the query string text box.
- 3) Press the send button.

Output:

SOFT338 API Client

This is a desktop application that makes use of the distributed API of the module, that (in this case) is for managing events. This is not extremely complicated, but is used to demonstrate the principles behind the four main http verbs (get, post, delete and put) in addition to an external call to the Google Places API.

Parameters

Http verb: GET

Url: events/show

Query String: ?name=Movie Night

Response Events

event_id	name	address	postcode	date	time
Oi4Lab7X	Movie Night	12 Hawks P...	PL124SP	20/2/2015	19:00

Response Details

Status: OK

Message: Events retrieved successfully

Unedited Response (Json): {"events":[{"address":"12 Hawks Park, Saltash, Cornwall, UK","date":"20/2/2015","event_id":"Oi4Lab7X","name":"Movie Night"}]}

In each case where a call to the API is made that involves using a HTTP verb that is not “GET”, an additional call is made to retrieve the list of events, for the purpose of easily visualising the changes that have been made.

From a programmatic point of view, the method “GetWebResponseDictionary” is invoked and a new instance of a WebClient is created that uses a different method of retrieving information dependant on the verb chosen. In all cases however, the response is collected as a string and deserialised into a dictionary of strings and objects. This method can be found by navigating to the “JsonParser” class.

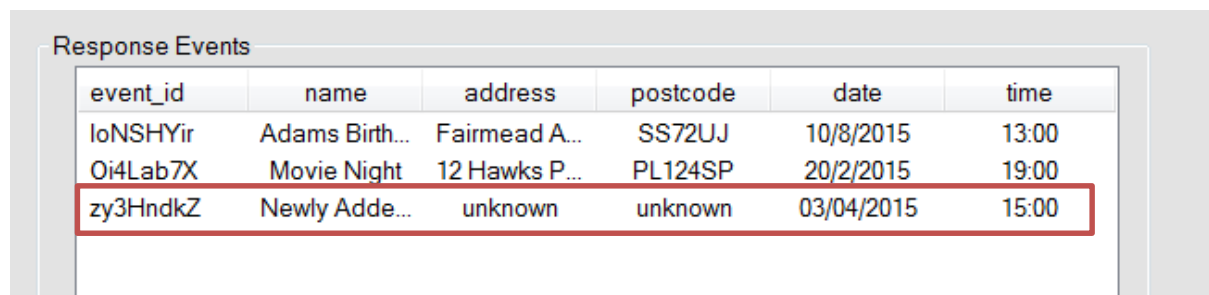
Once the string has been collected and deserialised, a new “Results” object is made by cycling through the dictionary and creating new instances of “Events”, the message and finally the status. The method for this is called “ConvertDictionaryToResults”.

5.2 Testing walkthrough (inserting data, using the client-side application and visualising changes)

This sub-section will describe how to create, retrieve, update and delete a specific instance of an event. It has been written to clearly illustrate that the aims of the assignment have been met – with reference to the sections that relate to CRUD.

Creating a new Event:

- 1) When the SOFT338APIClient loads, all events that are currently in the local database should be visible on the right hand side of the application in a list view. If this is not the case, then selecting “events” from the URL combo box may resolve the issue.
- 2) Select events/create from the combo box.
- 3) Change the HTTP verb to “POST” - as we are adding a new Event.
- 4) Add in the query string “?name=Newly Added Event&date=03/04/2015&time=15:00” and press the “send” button. This should add a new Event to the list. Adding an address and a postcode is optional and can be added to the query string if the willingness is there to do so.
- 5) The new Event should be visible from the list view.



event_id	name	address	postcode	date	time
IoNSHYir	Adams Birth...	Fairmead A...	SS72UJ	10/8/2015	13:00
Oi4Lab7X	Movie Night	12 Hawks P...	PL124SP	20/2/2015	19:00
zy3HndkZ	Newly Adde...	unknown	unknown	03/04/2015	15:00

Updating the new Event:

- 1) Select the URL “events/update”.
- 2) Change the HTTP verb to “PUT” – as we are updating.
- 3) Click and copy the event_id of the new “Event” from the list view on the right.
- 4) Add the query string: “?event_id=*copied event_id*&name=Changed&time=20:00”
- 5) Press the “send” button.

Response Events					
event_id	name	address	postcode	date	time
IoNSHYir	Adams Birth...	Fairmead A...	SS72UJ	10/8/2015	13:00
Oi4Lab7X	Movie Night	12 Hawks P...	PL124SP	20/2/2015	19:00
zy3HndkZ	Changed	unknown	unknown	03/04/2015	20:00

Only retrieve the new Event:

- 1) Click on the event_id from the list view that belongs to the newly added Event.
- 2) Change the HTTP verb to "GET".
- 3) Add the query string "?event_id=*copied event_id*".
- 4) Press the "send" button.
- 5) Only the "Event" that matches the event_id should be retrieved. Optionally this can be extended to include names – refer to section 3.

Deleting the new Event:

- 1) Click on the event_id from the list view that belongs to the newly added Event.
- 2) Change the HTTP verb to "DELETE".
- 3) Add the query string "?event_id=*copied event_id*"
- 4) Press the "send" button.
- 5) The "Event" should be removed from the list view.

Searching Google Places API:

- 1) Change the HTTP verb to "GET"
- 2) Enter the query string "?query=12 hawks park pl12&tag=formatted_address"
- 3) Press the "send" button.
- 4) Look at the "Unedited Response (JSON)" section for feedback.

5.3 Postman.

All examples present in section 5.1.2 can be used on Postman by following the same steps. However it may be harder to see the changes that are being made to the database. The search_places resource admittedly can be more aesthetically pleasing when using Postman, since the response is formatted in a nice way.

Word count: 3589

5.0 Bibliography

Dropbox. (2014) *Core API*. [Online]. Available from:

<https://www.dropbox.com/developers/core/docs>. [Accessed 20th March 2015].

Fielding, R. (2000) *Chapter 5: Representational State Transfer (REST)*. [Online]. Available from: http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. [Accessed 17th February 2015].

Li, H. (2014) *Security Research at the University of Virginia*. [Online]. Available from: <http://www.jeffersonswheel.org/category/social-networks>. [Accessed 17th February 2015].

Scribner, K. Seely, S. (2009) *Effective REST Services via .NET: For .NET Framework 3.5*. Boston: Addison Wesley.

StackOverflow. (2013) *How can I generate random alphanumeric strings in c#*. [Online]. Available from: <http://stackoverflow.com/questions/1344221/how-can-i-generate-random-alphanumeric-strings-in-c>. [Accessed 20th March 2015].

Microsoft. (2015) *Dictionary<TKey, TValue> Class*. [Online]. Available from: [https://msdn.microsoft.com/en-us/library/xfhwa508\(v=vs.100\).aspx](https://msdn.microsoft.com/en-us/library/xfhwa508(v=vs.100).aspx). [Accessed 3rd April 2015].

Vera, T. (2015) *JavascriptSerializer Example – Parsing JSON*. [Online]. Available from: <http://www.tomasvera.com/programming/using-javascriptserializer-to-parse-json-objects/>. [Accessed 20th March 2015].