# Alife: An ASU Event Searching Website

**Honors Thesis**

**Barrett, The Honors College**

**Mengqi Wu**

**Thesis Director: Dr. Yinong Chen**

**Second Committee Member: Dr. Xuerong Feng**

## Abstract

Alife is an event searching and event publishing website written in C# using the MVC software design pattern. Alife aims to offer a platform for student organizations to publish their events while enabling ASU students to browse, search, and filter events based on date, location, keywords, and category tags. Alife can also retrieve events information from the official ASU Event website, parse the keywords of the events and assign category tags to them. Alife project explores many concepts of Distributed Service-Oriented software development, such as server-side development, MVC architecture, client-side development, database integration, web service development and consuming.

# Table of Contents

# 1. Introduction

Students can certainly obtain comprehensive professional knowledge in a four-year university. Yet, successful college life should not be bounded by the overwhelming academic works but consists of various social activities and events. Through the active participation of these events, students can expand their knowledge scope, build their social skills, and accumulate leadership traits by interacting and learning from their peers. Therefore, student engagement activities play an essential in every Sun Devils' college life.

However, newcomers who are new to this big community may find the events information hard to grasp. In essence, school-sponsored events and student organization events are usually published on different websites; Many student organizations send out their events information via emails only to whom previously subscribed; Existing school websites display only core events information but lacks customized control for students to subscribe their favorite event categories.

Therefore, the primary objective for the Alife project is to create an ASU event searching website that integrates both school-sponsored events and student organization events to provide unified event publication, event categorization, keyword extraction, customized user-profiles, and user controls. The secondary objective of this project is to gain experience in service-oriented computing and distributed software development through hands-on practice.

Both objectives are accomplished. Alife website provides two types of user roles, the student account type, and the organization account type. Once registered, users can view and search events by categories, locations, and times. Each event can be displayed in detail

containing category tags, keywords, descriptions, times, and locations on an interactive map. The users can also browse through the organizations' page to view detailed information about each organization. The organization accounts are permitted to create new organizations and publish their events. In addition, Alife can parse events information published on the official ASU Event website, extract their keywords, and assign category tags automatically.

This project explores various aspects of Distributed Service-Oriented Computing such as RESTful service development and consuming, web application development, database management, service and web application hosting. The development process generally follows the Agile principle: the entire project was divided into several small deliverables, such as creating a mockup website, server-side development, client-side development, web service development, etc. Recurrent meetings with thesis committee members are set up approximatively every two weeks to evaluate deliverable progress, identify improvement needed, and adjust the requirement of future deliverables.

This project used the Model-View-Controller (MVC) as the main software design pattern. It utilized various tools, such as Bootstrap, jQuery DataTable, Html Agility Pack, to complement the development process. Detailed discussion on requirement analysis, development process, system design, application implementation, project testing, and technologies used is included in the later section.

## 2. Related Work

ASU Digital Repository is a great resource to view related honor thesis projects done by the previous alumnus. Existing works related to event searching application is analyzed to understand the potential scope of this project.

The *Fiddlevent* was an event searching website created by Christopher Thornton [1] in 2013 as his Honors Thesis Project. This website is written in Ruby on Rails using the MVC software design pattern. The goal for this website is to enable any person to go online and find local events that interest him, and enable merchants to post their events online [1]. Event searching is achieved through the users providing their current location, events location coordinate, and filter set is integrated to allow narrowing down the search. The *Fiddlevent* implemented role-based access control. Three types of users are supported on this website: the administrator who can define different filters available to customers and merchants, the merchant who can publish events and define the location and filter set of the event, and the customer who can search events. The innovation of this website is that it supports different user types on registration, it uses predefined filter tags to optimize the searching result, and it allows the users to mark events as favorite.

However, there are several limitations exist in his approach. Firstly, the event searching function on this website only supports location search. In this way, the users can only search for events near a specified area then use filters to narrow down the search, instead of allowing the users to search events based on keywords in the names and details of the events. Secondly, this website only allows the users to add individual events to their favorite list but

does not allow the users to add a whole event category to the list. Thirdly, the *Fiddlevent* relies solely on registered merchants to populate its event database. It does not retrieve events information from external sources; the lack of third-party data may impair its user experience since there is no existing data to start with.

# 3. Technical Specification

## 3.1 Requirement Analysis

Before setting up the initial project requirement, an analysis was conducted on existing solutions for ASU students to search for events. *ASU Events* (asuevents.asu.edu), as shown in Figure 1, is the official ASU website for school-sponsored event publishing, it contains school events from all ASU campuses. Various filters, such as date, location, cost, and topic, are implemented to optimize the searching result. However, there are many limitations on this website. Firstly, this website display events by each date, and it treats each reoccurring event as different events and displays them separately, which results in the duplication of event information. Secondly, the user interaction is limited to only search and view events, and there is no interface for organizations to publish their event and provides no functions for the users to add events to a subscription list. As an improvement to the existing solution, Alife should possess the following features as essential requirements:

- Functionality Requirement
    - Role-based access control allowing the registration and log in for three types of users: the administrator, the organization owners, and the students
    - The administrator should be able to manage all published events, add new

category tags, and update the event database.

- The organization owners should be able to publish new events, manage events they published, update their organization profile page.
- The student users should be able to view all events published on Alife and the official ASU Event website.
- The users should be able to filter events by event date, location, and categories. They should also be able to search for events by keywords.
- All events data retrieved from the official ASU Event website should be parsed to get the keywords list based on the event details.
- The website should have a tagging system to assign category tags to each event. The users should be able to subscribe to category tags.

- Interface Requirement
  - The website should have an intuitive structure for the users to navigate through different web pages.
  - Events information should be displayed in a proper format, and event locations should be displayed on an interactive map.

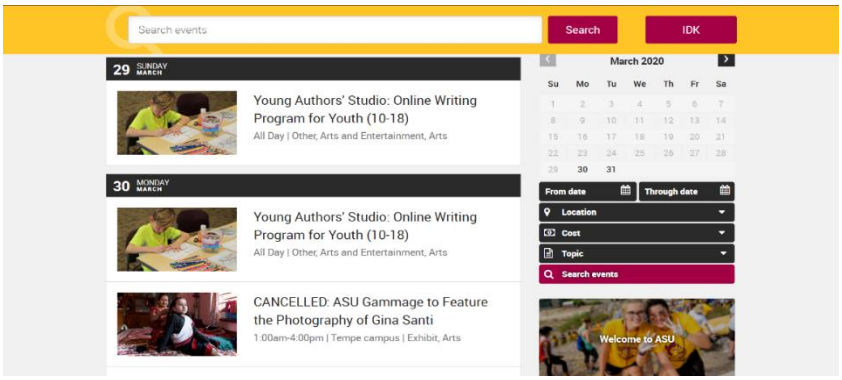


Figure 1: The official ASU Event website

### 3.2 Development Process

The development process generally follows the Agile software development principle. Alife project is divided into several deliverables such as requirement analysis, prototype website design, Alife application development, Web Event Parser Service development. In each meeting with the thesis committee members, the progress of deliverables is examined, and the requirements of the future deliverables are adjusted accordingly. After developing each functional component, such as event searching function, event publishing function, and event parser service, unit testing is performed to ensure the correctness before moving on to the next functional component.

### 3.3 System Design

The Alife project consists of the main web application and a Web Event Parser Service. This project used ASP.NET MVC as the primary software architecture. MVC stands for Model-View-Controller, this design pattern was conceived in the late 1970s by Trygve Reenskaug at Xerox PARC, as part of the Smalltalk [2]. MVC pattern separates the web component into three logical layers: The Model, also called the business layer, contains the application models such as domain models and view models, these models together represent the data and the logic of the application. The View, also called the display layer, contains the web pages that render the data and present it to the users. The Controller, also called the input layer, handles the user inputs, and forms a connection between the View and the Model. There are other alternative frameworks available to create web applications, such as ASP.NET WebForm, which has drag-and-drop features, providing the developers an easy way

to create small-scaled event-driven web pages. Although the MVC pattern is harder to grasp, the separation of view and control makes it more flexible and scalable for complex multipage web applications, which makes it a better choice for this project.

### 3.1.1 Alife Web Application

1. Models

The models are like the classes in object-oriented programming; they store the variable names and types for different classes that are used across the application. Domain models store the data members for each object. For example, The Event Model, see figure 2, contains variables to represent an event, such as event ID, name, location, date, time, detail, tags, keywords, and related organization. Domain models should be defined in a higher priority during the development process because it not only contains all the attributes of a class but also used to generate data tables in a SQL database using the code-first approach. Details of code-first implementation using Entity Framework are covered in the Implementation section. Domain models can also be used to pass object data to a view page; this allows the View page to render the data in the run-time dynamically. View models are particularly useful if multiple domain models need to be passed to the view. Detailed UML diagram for domain models used in Alife can be found in Appendices.

```
public class Event
{
    public int Id { get; set; }

    [Required] // Data Annotation
    public string Ename { get; set; }

    [Required]
    public string Location { get; set; }

    [Required]
    [DataType(DataType.DateTime)]
    public DateTime Time { get; set; }

    [Required]
    [Range(1, 48)]
    public int Duration { get; set; }

    [Required]
    public string Detail { get; set; }

    [Required]
    public int OrganizationsId { get; set; }

    public string Tags { get; set; }

    public string Keywords { get; set; }
}
```

Figure 2 : Event model with data

2. Views

Each View is a .cshtml file that renders the web page controls, such as buttons and links, and presents the data. These Views are usually linked to particular actions (or methods) in a Controller class. The View page uses the Razor View Engine, which can render server-side C# code along with HTML elements. This powerful feature allows the data of the domain object to be displayed dynamically. The example of using Razor View is shown in the Implementation section. Alife application has many View pages; each provides different functions, such as displaying all events, publishing new events, view event details, etc. These View pages not only present data but also contain client-side code written in JavaScript to enhance the user experience.

3. Controllers

Controllers work like scaffolds of a building; they connect every View and Model and perform logical actions, such as processing user inputs, accessing the database, consuming web services. Besides the predefined Controllers generated by the MVC template, there are three Controller classes added to the project: Event Controller, Organization Controller, and Users Controller, each of them contain actions to manipulate related domain objects. For example, the Event Controller has actions to display all events, create a new event, show the details of a given event, manage all events, etc. Additional API Controllers are added for each Controller class, these API Controllers, derived from System.Web.Http.ApiController class, are used to handle HTTP requests to particular actions. This practice is beneficial if an action of a web application needs to be called by another web application or if a mobile version of

the application will be developed later. Some of the AJAX calls in Alife application are performed by calling API Controllers.

4. Database

Alife utilizes the SQL Server Database provided by Microsoft in Visual Studio to store events, organization as well as user data. The event table is linked with the organization table using organization ID as the foreign key. The users have to provide a valid organization ID when creating a new event; this practice enforces the referential integrity of the database. The following tables are created using the code-first approach: see Figure 3.

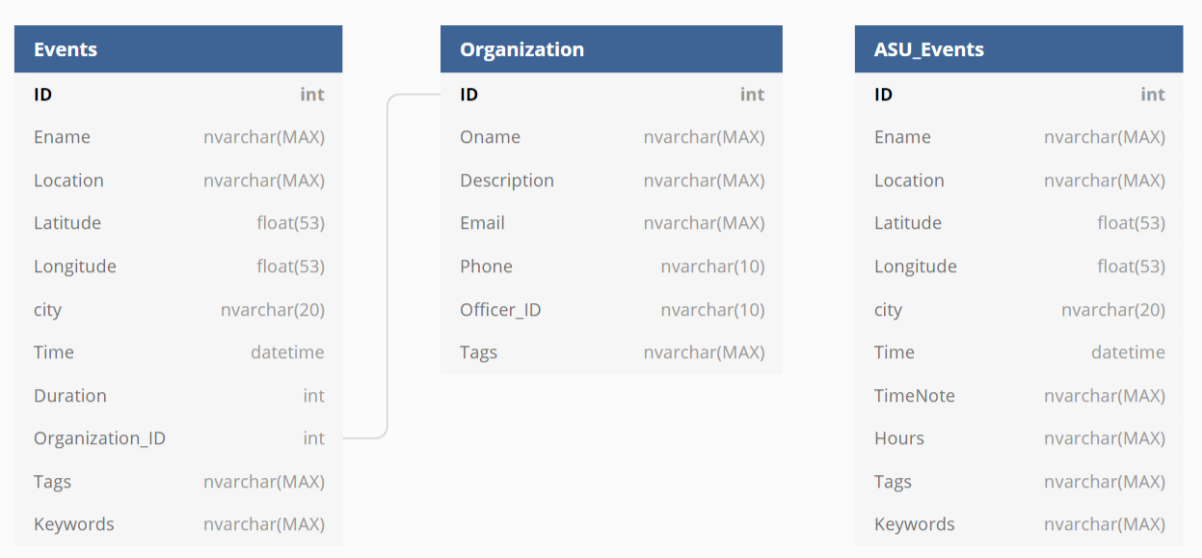


Figure 3 : Entity-Relationship Diagram

3.1.2 Web Event Parser Service

This service, developed using ASP.NET MVC Web API architecture, parses the events information on the official ASU Event website and performs various essential operations such as location coordinate calculation, keyword extraction, category tag assignment.

5. Models

The Models in the Web Event Parser consists of two categories: Models that are used in the event parsing process, such as Event_ASU Model, EventParser Model, Location Parser

Model, and Models that are used in the keyword extraction process, such as Corpus Mode.

6. Controllers

All Controllers in this service are implemented as API Controllers using RESTful convention. This approach ensures this service can be consumed as a RESTful service in the Alife application. Two Controller classes are used in this service: the ASU_Event_Parser Controller contains operations used to retrieve event data from the official ASU Event website, such as getting events list and paring event tags; the TF_IDF Controller contains operations that extract keywords using TF_IDF algorithm.

### 3.4 Implementation

This section is not meant to be a chronological log of the whole implementation process but rather an explanation of each functional component of the Alife application. The strategies for implementing each component are explained as detail as possible, and code samples are shown in Figures and Appendices.

#### 3.4.1 View Events: Passing Data from Controller to Views

In order to dynamically display events data, domain objects need to be passed to the View pages from the Controllers. First, the actions corresponding to the Views are modified, the Return statements are changed from *return View();* to *return View(object);* See Figure 4 and Figure 5

```
public class Event
{
    public int ID { get; set; }
    public string Event_Name { get; set; }
    public DateTime Event_Date { get; set; }
}
```

Figure 4. Sample Event Class

```
public ActionResult Index()
{
    Event NewEvent = new Event() {
        Event_Name = "Sample Name 1",
        Event_Date = DateTime.Now,
        ID = 1
    }; // create an event object and initialize its data.
    return View(NewEvent); // pass object to view
}
```

Figure 5. Index Action in Sample Controller

Then in the View page, the Model can be declared at the beginning of the page and used as a local object by Razor View Engine. In this way, the dynamic object data can be combined with static HTML elements. For example, in Figure 6, the object variable such as Event_Name, Event_Date are placed inside an HTML heading element. The values of these variables are determined in the run-time.

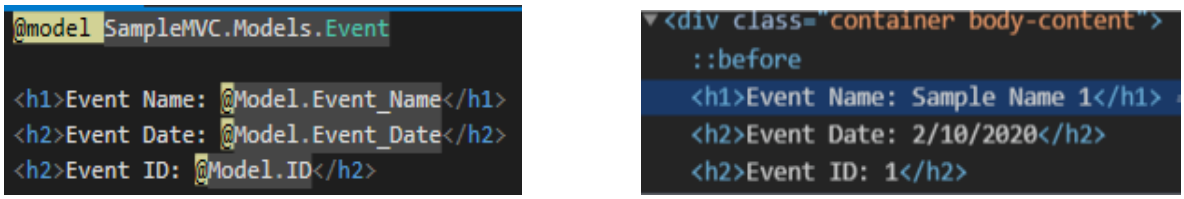


Figure 6. Accessing event object using Razor View and the resulting HTML page after application running

View Pages in Alife project are developed using the method discussed above. The Event Index page reads a list of events and displays each of them using a *for* loop, inside which the HTML elements are created to show event name, organization name, and a "Get Detail" button. After clicking the button, the users will be redirected to the Event Detail page where the detailed information, such as event date, time, location, description, keywords, and tags, are presented. The Event Detail page also utilizes Google Map API to display an interactive Google Map showing the event location. The map is generated by loading the latitude and longitude data from the event object. See Figure 7.

```
<script>
    // Initialize and add the map
    function initMap() {
        // The location of the event
        var eventLocation = { lat: @Model.Lat, lng: @Model.Lon }; // get Lat and Lon data from event model.

        // The map, centered at event location
        var map = new google.maps.Map(
            document.getElementById('map'), { zoom: 18, center: eventLocation });

        // The marker, positioned at event location
        var marker = new google.maps.Marker({ position: uluru, map: map });
    }
</script>
```

Figure 7. JavaScript code for generating Google Map

Each category tags is displayed as a button. Event handlers, written in JavaScript, are implemented to handle the button click event. If the users click on the tag button, the event handler calls an internal API to add the category tag to the users' tag subscription list. Users can always go to their profile page to manage the tag list. Each tag on the tag subscription list redirects the users to browse the related events.

### 3.4.2 Save Events: Setting Up Database Using Code-First Approach

Entity Framework is used in this project to connect the application to the SQL Server Database. It is an open-source Object-Relational Mapper developed by Microsoft to map the Dataset in a relational database into domain objects in the application [3]. The use of Entity Framework enables developers to access the data tables without writing database scripts. There are two types of workflows when utilizing Entity Framework: code-first and database-first. In database-first workflow, developers design and create data tables in the database then the Entity Framework can generate domain models based on the database schema. While in code-first workflow, domain models are created first then the Entity Framework will generate data tables accordingly. See Figure 8

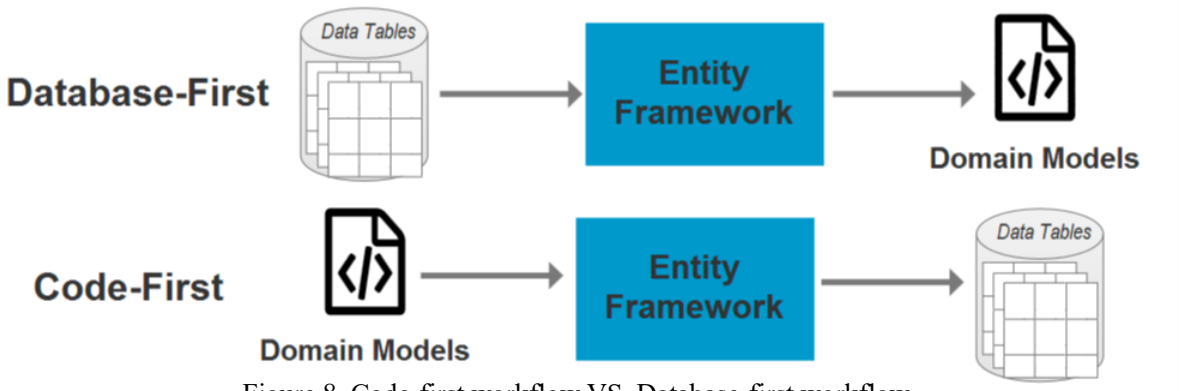


Figure 8. Code-first workflow VS. Database-first workflow

Alife project chooses the code-first workflow over the database-first for the following

reasons. Firstly, the code-first approach is more productive, and it resembles more to the convention of the Object-Oriented programming, where object classes are created first before implementing other functions. Creating the domain models and let the Entity Framework generate the data table accordingly is much faster than creating data tables first in the Table Designer tool. Secondly, code-first workflow makes it easier to make changes; for example, if additional variables need to be added to the domain models, only the change in code is needed, and the data tables can be updated by the Entity Framework. Thirdly, the code-first approach allows the full version control of the database; every modification is recorded as a database migration. The recording of all migrations provides a full history of changes applied to the database and enables the developer to restore the database to any migration at any time.

In order to enable code-first migrations, the command: *enable-migrations* need to be executed in the Package Manager Console. Then the *DbSet* needs to be added into the *ApplicationDbContext* class to reference a domain model. See Figure 9.

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    public DbSet<Event> Events { get; set; }
```

Figure 9. Code-first workflow VS. Database-first workflow

After this, we can execute the command: *add-migration* in the Package Manager Console to create a new migration. Finally, we execute the command: *update-database* to apply

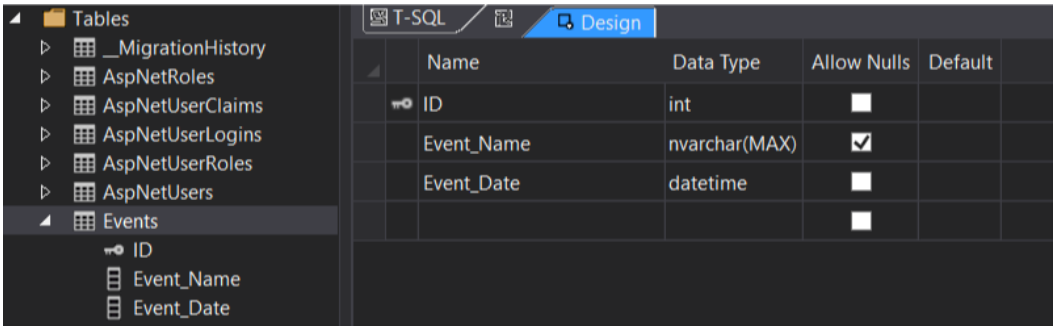


Figure 10. Data table created by code-first migration.

changes to the database. See Figure 10.

### 3.4.3 Publish Events: Using forms to Get and Store User Inputs

Forms are implemented in the View pages to get user input when publishing new events or creating new organizations. Instead of hardcoding HTML elements, HTTP Helper classes are commonly used in Alife to generate labels and input text fields in Razor View Engine. This method works closely with the domain objects we passed to the View, thus easier to implement. An example is shown in Figure 11.

```
<div class="form-group">
    @Html.LabelFor(m => m.Event.Ename)
    @Html.TextBoxFor(m => m.Event.Ename, new { @class = "form-control" })
    @Html.ValidationMessageFor(m => m.Event.Ename)
</div>
```

Figure 11. Sample form group using HTTP Helper Class

After the users submit the form, a new event object is passed to the corresponding Action of a Controller class using the HTTP POST method. In the Action method, an ApplicationDbContext object is created to add the new event object to the database using *Add()* and *SaveChange()* methods. Proper error handling should be implemented to catch the error when saving the object to the database. See Figure 12 for the sample code.

```
[HttpPost]
[ValidateAntiForgeryToken] // prevent cross-site attack
public ActionResult Create(Event NewEvent) // New Object returned by Form View
{
    var _context = new ApplicationDbContext();
    _context.Events.Add(NewEvent); // add NewEvent model to DB context.
    try
    {
        _context.SaveChanges(); // update database.

    }
    catch (DbEntityValidationException e)
    {
        Console.WriteLine(e);
    }
    return View();
}
```

Figure 12. Sample code for saving new object to database

### 3.4.4 Search Events: Querying Event Objects in Database

Alife allows users to search for events and organizations by providing keywords. The keywords entered by the users are passed to a search Action as a string. In the search Action, the keywords string is split into a string list, then each element in this string list is used as a query string to the event table and organization table. In each iteration of the query, LINQ is used to perform the query on the ApplicationDbContext object; the latter translates LINQ to SQL command and query the database. The query results are added into HashSets to avoid duplication, after the entire query process, this Action returns a View Model, containing lists of events and organizations. See Appendices for sample code.

### 3.4.5 Filter Events: Client-side Scripting

Filters in Alife project consists of checkboxes, radio buttons, and dropdown list. The filter functionality is achieved by client-side development using the jQuery library in JavaScript. The alternative method to achieve filter functionality is implementing filter Actions on the server-side. However, compared to the server-side approach, the client-side scripting is more responsive, the Ajax feature in the jQuery library allows the change of HTML elements dynamically without reloading the entire web page, thus improves the user experience. After the page load, the events data used in the filter, such as event date, location, and tags, are inserted into the web page as Attributes of HTML elements. When the users apply a filter, the click event is handled by event-listener scripts, where a jQuery selector selects HTML elements that do not conform to the filter criteria and hide them from the display. For example, if a user clicks a location filter called "Tempe" to filter events, then

the jQuery selector will select and hide all events elements that does not have "Tempe" within its Class Attribute. What is more, a reset button is implemented to reset all filters without reloading the web page. Finally, additional scripts are implemented to create filter tags to indicate which filters are currently applied. See Figure 13.

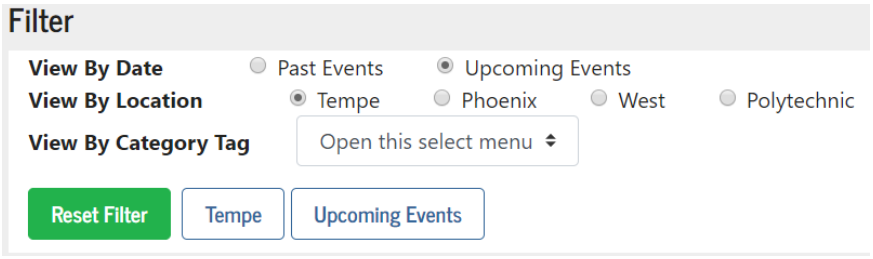


Figure 13. Filters in Alife application.

3.4.6 Parse Events: Crawling Events Data from ASU Website

One of the main goals for the Alife project is to integrate ASU events and Organization events. Therefore, crawling and parsing events information from the official ASU Event website is an essential part of the Alife application. These functions are achieved by the Web Event Parser service. The parser service contains two API Controllers: the Event Parser Controller and the Keyword Extraction Controller.

In the parser Controller, the Html Agility Pack tool is used to parse a given HTML into an HtmlDocument, which is a Document Object Model (DOM) and can be read as a tree structure. There are multiple ways to select a particular HTML element: selecting its XPath, selecting its CSS class, or selecting its JS Path. In this parser service, the XPath is used to select the given HTML element due to the structural similarity of the event elements. In specific, there are thirty event blocks on every page of the official ASU Event website. The relative XPath of each event differs only in one index, thus enable us to find only the relative

XPath for an element of the first event and use that to iterate through the same event elements of all events. Figure 14 shows the sample code of the iterative selection of the event names. Note that HTML contents need to be trimmed appropriately to remove extra whitespaces and line breaks.

```
for (var j = 1; j < 31; j++)
{
    // the place holder {0} inside div[] will be replaced by variable j to iterate through each name path
    string namePath = String.Format("//*[@id=\"block-system-main\"]/div/div/div[1]/div[{0}]/div/div[1]/div[1]/div[2]/h2", j);
    string Ename = document.DocumentNode.SelectNodes(namePath)[0].InnerText.Trim(); // get the inner HTML content
    Ename = Regex.Replace(Ename,@"\s+"," "); // remove extra whitespace and line break
}
```

Figure 14. Sample code of iterative selection of event names.

Unfortunately, the events published on the official ASU Event website are not always consistent, some events may have missing information, such as event location, so that the element XPath obtained by incrementing the index may not contain an element. Thus, proper error handling methods need to be implemented to cope with this situation. Another challenge for the event parsing is that the events published on the official ASU Event website only contain location information as address string. However, the exact latitude and longitude data are needed to present the event location on the Google Map. To get these data, the parser Action calls the Google Geocoding API using the event address it parsed from the ASU Event website. The results are returned in JSON format and are parsed and serialized as a Location object, the coordinate information is extracted from the Location object and store into the Event object. The sample code is shown in the Appendices.

### 3.4.7 Categorize Events: Extracting Keywords and Assigning Tags

A keywords list is generated for each event based on its event description paragraph. When the parser Action extracts each event description from the official ASU Event website,

it passes the event description string to a Cleanup Action, which contains a list of stop words strings and removes the stop words from the event description string along with special characters and unnecessary whitespaces. Then the parser Action calls the keyword extraction Action using the cleaned string as the input parameter.

Various algorithms can be applied to perform keyword extraction; among them, the simplest method is to calculate the total number of occurrences for each word then sort these words based on their frequency. However, this approach has significant limitations: common words, such as "event", "university", and "student" may have higher frequency but provide less meaning to the comprehension of an event description context. On the other hand, words with less frequency, such as places, names, and terminologies, may have significant meanings and can uniquely characterize an event. Therefore, by just calculating the word frequency may not be an optimal solution for keyword extraction.

Alife project uses TF-IDF (Term Frequency-Inverse Document Frequency) algorithm to perform the keyword extraction operation. This algorithm can reflect how important a word is to a document in a collection of documents (Corpus) by assigns a numerical weight, calculated based on TF and IDF, to each word. The TF value increases as a word appears more in a document and the IDF value decreases as that word appears more in the corpus,

$$TF(\mathrm{x}, D) = \#\ of\ occurance\ of\ Word\ x\ in\ Document\ D$$

$$N(\mathrm{x}, C) = \#\ of\ Document\ containing\ Word\ x\ in\ Corpus\ C$$

$$IDF(\mathrm{x}, C) = log\left(\frac{\#\ of\ Document\ in\ Corpus\ C}{1+N(x,C)}\right)$$

$$Weight(\mathrm{x}) = TF(x, D) \times IDF(x, C)$$

Figure 15. Equation for TF-IDF Algorithm

with these two value combined, an average weight value can be calculated for each word. See Figure 15. The corpus used in this project is a JSON file that consists of 462 events descriptions parsed from the official ASU Event website.

Finally, each word-value pair is stored in a *Dictionary<string, double>* data structure, then all the words are sorted in descending order based on their weight value, and the top 20 words form the keyword list of the event. The sample code is shown in the Appendices.

After the event parser Action calls the keyword extraction Action and converts the return Dictionary data into a list of keywords, it calls the Parse Tag Action with the list of keywords as the input parameter; the latter can categorize the list of keywords into predefined category tags. All predefined tags are stored in a *Dictionary<string, string>* data structure in a JSON file. The key of each Dictionary element is the name of the category tag, such as “Art”, “Academic”, and “Literature”. The value of each Dictionary element is a string containing words that are related to a given category separated by commas. For example, a category tag with a key of “Art” may have its elements as “artist, film, gallery, exhibition, sculpture, photography…”. Each word in the keyword list is taken to match the category tags in the tag Dictionary, and the match results form a tag list.

The accuracy of categorizing the keywords into tags depends on the scope of the tag Dictionary, which should contain as many words related to the tag category as possible; it is not ideal to manually type those words. Therefore, an existing web service, called Datamuse API, is used to get the list of words related to a given category name. Datamuse is a word-querying engine designed to find words that match given constraints; if proper constraints are

applied, it can be used to find a list of words related to a specific word. This API is consumed as a RESTful service by the keyword parser; the results are converted into strings and stored in the tag Dictionary. The sample code is shown in the Appendices

### 3.5 Tools and Technologies Used

- GitHub Desktop

GitHub Desktop is used throughout the whole development process of the Alife project as the primary version control solution. A new commit is created every time a new feature is developed or tested so that every change can be tracked and reverted. Compared to the traditional command-line interface of Git, the GitHub Desktop provides a more user-friendly interface and a flat learning curve. The syntax highlighting feature is particularly useful when comparing code changes.

- Bootstrap 4

Bootstrap is an open-source front-end framework containing CSS based templates for designing the graphic user interface of web applications. It provides nicely designed CSS styles of different themes for every HTML element. These designs can be applied by simply adding CSS classes in the elements' Class Attribute. Bootstrap classes are used in every View Page of the Alife project.

- jQuery DataTable

jQuery DataTable is a plug-in for the jQuery JavaScript library. It can be applied to the HTTP Table element to stylize the data tables on the HTML webpage. It also offers many practical features, such as table pagination, sorting, and basic searching. jQuery DataTable is

used in the Event Manage page, and Tag Manage page of the Alife project to provide a table-based management feature.

- Talend API Tester

Talend API Tester is a light-weight API testing tool on the Google Chrome platform. It can be used to consume and RESTful APIs by making API calls using different HTML Methods, such as GET, PUT, DELETE. It can also present request headers and response bodies of API calls in JSON format, which are useful for testing and debugging the web APIs. This tool is commonly used in the Alife project during each unit testing process. Each Action in the API Controllers can be configured as individual web APIs using attribute routing and can be tested independently from the entire application using Talend API Tester. See Figure 16. In this way, we can analyze the API behavior by examining the request headers and response bodies to ensure the correctness of each functional component before integrating it into the entire application.

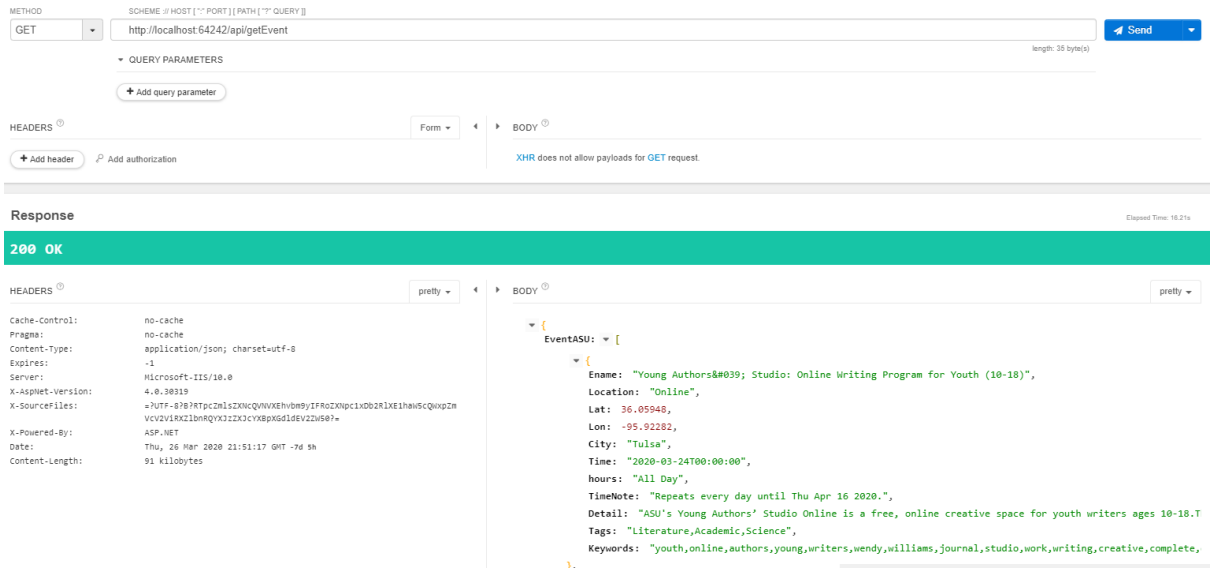


Figure 15. Unit testing of GetEvent Action using Talend API Tester

- Microsoft Azure

Azure is a cloud computing platform created by Microsoft for application development, testing, deploying, and managing services. It offers various essential services required in the Alife project, such as SQL Database Server for hosting the online database and App Services for deploying Alife application and Web Event Parser service. Azure is also well integrated inside the Visual Studio, which simplifies the deployment process for the developer. In essence, an application can be deployed along with the database, and code-first migrations are automatically executed on the deployed server. These features provided by Azure free account are utilized by the Alife project during the deployment and testing process. Alife application and Web Event Parser are deployed as separate App Services, and the SQL database used by Alife is also hosted on the Azure server.

## 4. Future Work

Even though the project requirement is met, moderate improvements are still needed for the Alife project to be considered production-ready. Additional features should be implemented to advance the functionalities and user experience provided by Alife application. For instance, an email-based real-time notification system can be implemented to notify users about events updates. Popular social media platforms, such as Facebook and Twitter, can be integrated for users to share event information. What is more, a mobile version of the Alife application can be developed using existing API Controllers.

## 5. Conclusion

Alife is an event publishing and searching website designed to enhance the ASU students' experience of viewing university-sponsored events and student organization events. It offers an excellent opportunity to get hands-on practice of various concepts of Distributed Service-Oriented software development, such as web application development, MVC architecture, client-side development, database integration, web service development and consuming. Moreover, this project incorporates some aspects of software engineering and project management, such as requirement analysis, system design, and software testing. Finally, this project demonstrates the Agile software development principle with the evolving requirements, sustainable development, and the continuous delivery of functional components. The finish of this Honors Thesis Project should not signify the end of the Alife but a new starting point for future refinement.

## 6. Acknowledgments

I would like to show my gratitude to people who have offered assistance in this Honors Thesis Project.

**Yinong Chen - Thesis Director**

Dr. Chen is a professor in the School of Computing, Informatics, and Decision Systems Engineering. I first met him when I took his *CSE 240: Introduction to Programming Language* in Spring 2018. Then I took *CSE 445: Distributed Software Development* and *CSE 446: Software Integration and Engineering* with him in my senior year. Alife project is inspired by *CSE 445 and CSE 446* classes, and I was able to apply what I learned from these classes to the Alife project. I would like to thank Dr. Chen for his supervision and help with my Honors Thesis Project.

**Xuerong Feng – Second Committee Member**

Dr. Feng is a professor in the School of Computing, Informatics, and Decision Systems Engineering. I took *CSE 205: Object-Oriented Programming and Data Structures* with her in Spring 2017. Her class allows me to build a solid foundation on my programming skills as well as the understanding of data structures and algorithms. I would like to thank Dr. Feng for offering advice on the front-end development of the Alife project.

**Cassin Dyson – Honors Academic Advisor**

Mr. Dyson is an Honors Academic Advisor of Barrett, The Honors College. He was my advisor since my junior year. I would like to thank Mr. Dyson for providing me useful information about the Honors Thesis Project and answering my questions through email and

appointments.

## 7. Reference

Thornton, C. G., Balasooriya, J., & Nakamura, M. (2013). *Event Searching Web Site.* Honors Thesis Project, Arizona State University, Barrett, The Honors College.

Chen, Y. (2017). Service-Oriented Computing and System Integration: Software, IoT, Big Data, and AI as Services (6th ed.). United States of America: Kendall Hunt Publishing.

Microsoft. (2016, October 27). Entity Framework Core. Retrieved from Microsoft Docs: https://docs.microsoft.com/en-us/ef/core/

# 8. Appendices

## Code Sample

- Search Action

```
public SearchViewModel getSearchResult(string keywords) {
    HashSet<Event> eventSet = new HashSet<Event>();
    HashSet<Organizations> orgSet = new HashSet<Organizations>();
    HashSet<Event_ASU> eventASUSet = new HashSet<Event_ASU>();
    var keywordList = keywords.Split(' '); // split keywords string to string list
    foreach (string keyword in keywordList) // iterate over the list
    {
        // query Events table and add the result into Hashset of Event Object
        eventSet.UnionWith(_context.Events.Where(e => e.Ename.Contains(keyword) ||
        e.Detail.Contains(keyword) ||
        e.Location.Contains(keyword) ||
        e.Tags.Contains(keyword)).ToHashSet());

        // query Organizations table and add the result into Hashset of Organization Object
        orgSet.UnionWith(_context.Organizations.Where(o => o.Oname.Contains(keyword) ||
        o.Description.Contains(keyword)).ToHashSet());

        // query ASU Event table and add the result into Hashset of ASU_Event Object
        eventASUSet.UnionWith(_context.Event_ASU.Where(e => e.Ename.Contains(keyword) ||
        e.Location.Contains(keyword) ||
        e.Detail.Contains(keyword) ||
        e.Tags.Contains(keyword)).ToHashSet());
    }

    // Convert Hashsets to Lists
    IEnumerable<Event> eventList = eventSet.ToList();
    IEnumerable<Organizations> orgList = orgSet.ToList();
    IEnumerable<Event_ASU> eventAsuList = eventASUSet.ToList();

    // create View Model to return the search result
    SearchViewModel searchResult = new SearchViewModel() {
        EventResults = eventList,
        EventASUResult = eventAsuList,
        OrgResults = orgList,
        keywords = keywords
    };
    return searchResult;
}
```

- Location Parser

The private API keys are covered for security purpose.

```
public static string[] GetLatLon(string address) {
    string[] coordinate = {"","","" };
    using (var client = new HttpClient())
    {
        // assign the base uri of Google Geocoding API
        client.BaseAddress = new Uri("https://maps.googleapis.com/maps/api/geocode/json?address=");

        // create the full uri with address variable and private api key.
        var fullurl = client.BaseAddress + address + "&key=A                                                    HxY";
        var responseTask = client.GetAsync(fullurl);
        responseTask.Wait();
        var result = responseTask.Result;
        if (result.IsSuccessStatusCode)
        {
            // parse and serialize the JSON result as Location object
            var parse = result.Content.ReadAsAsync<LocationModel.Rootobject>();
            parse.Wait();
            var parseResult = parse.Result;
            if (parseResult.status == "OK")
            {
                // extract the coordinate data and city data from the Location object
                coordinate[0] = parseResult.results[0].geometry.location.lat.ToString();
                coordinate[1] = parseResult.results[0].geometry.location.lng.ToString();
                foreach (var component in parseResult.results[0].address_components) {
                    if (component.types[0] == "locality") {

                        coordinate[2] = component.short_name;
                    }
                }

            }
            else { // apply defluat value if no data is found
                coordinate[0] = "N/A";
                coordinate[1] = "N/A";
                coordinate[2] = "N/A";
            }
        }
        else {
             // apply defluat value if API call fails
            coordinate[0] = "N/A";
            coordinate[1] = "N/A";
            coordinate[2] = "N/A";
        }
    }

    return coordinate;
}
```

- TF-IDF Function

```
[Route("api/GetTFIDF")]
[HttpGet]
public  Dictionary<string, double> GetTFIDF(string Document) {

    // load the Corpus JSON file
    string jsonPath = Path.Combine(HttpRuntime.AppDomainAppPath, @"App_Data\EventCorpus.json");

    // Read and deserialize the JSON file into Corpus object
    var Corpus = JsonConvert.DeserializeObject<CorpusModel>(System.IO.File.ReadAllText(jsonPath));
    List<string> CorpusList = Corpus.Corpus.ToList(); // create the corpus list.
    var Doc = Document.Split(' ').ToArray(); // split input document into string array.
    Dictionary<string , double > TFValues = new Dictionary<string, double>() {}; // create dictionary to store TF value
    foreach (string word in Doc) { // iterate through the input document
        var TFinDoc = 0;
        string pattern = @"\b" + word + @"\b";
        var TF = Regex.Matches(Document,pattern).Count; // use Regex pattern to match exact word.
        CorpusList.ForEach(str => // calculate the number of Document containing word in the Corpus
        {
            if (Regex.Matches(str, pattern).Count > 0 ) {
                TFinDoc += 1;
            }
        });
        double IDF = Math.Log((double)Corpus.Count / (double)(1+TFinDoc)); // calculate IDF
        double TF_IDF = TF * IDF; // calculate TF_IDF weight

        TFValues[word] = TF_IDF; // assign the weight value to each word
    }

    // use linq to order the dictionary in descending order and select top 20 words
    var items = (from pair in TFValues orderby pair.Value descending select pair).Take(20);

    // convert ordered pair to a new dictionary
    Dictionary<string,double> RankedTF = items.ToDictionary(x => x.Key, y => y.Value);
    return RankedTF;
}
```

- Tag Parser

```
[Route("api/GetTag/{keywords}")]
[HttpGet]
public string ParseTag(string keywords)
{
    var keywordList = keywords.Split(','); // Split keywords into list
    HashSet<string> Tagset = new HashSet<string>(); // create hashset to avoid duplication
    string tags = "";
    Dictionary<string, string> TagDictionary = new Dictionary<string, string>(); // create tag dictioinary
    string jsonPath = Path.Combine(HttpRuntime.AppDomainAppPath, @"App_Data\Category.json");
    TagDictionary = JsonConvert.DeserializeObject<Dictionary<string,string>>(System.IO.File.ReadAllText(jsonPath)); // read the JSON file as dictionary
    foreach (var keyword in keywordList) { // iterate through the list
        foreach (var dic in TagDictionary) { // iterate through the dictionary
            if (!Tagset.Contains(dic.Key))
            {
                string pattern = @"\b" + keyword + @"\b"; // use regular expression to match whole word
                if (Regex.IsMatch(dic.Value, pattern))
                {
                    Tagset.Add(dic.Key); // use set to avoid duplicate
                }

            }
        }
    }

    foreach (var tag in Tagset) { // combine the tag in the tag set into a string, each tag is separated by a comma
        tags += tag + ",";
    }
    tags = tags.Trim(','); // trim the extra comma
    return tags;
}
```

- Consuming the Datamuse API

```
[Route("api/GetCategory")]
[HttpGet]
public Dictionary<string, string> GetCategoryDict()
{
    Dictionary<string, string> Dict = new Dictionary<string, string>(); // create dictionary to store tag and tag list
    List<WordClass> WordsList = new List<WordClass>();
    List<string> Category = new List<string>()
    {
        "Art","Academic","Sports","Learning", "Club","Music", "Career", "Literature","Business","Science" // predefined tag category
    };
    // initialize the dict
    foreach (var cat in Category) { // initialize the tag dictionary
        Dict[cat] = "";
    }
    foreach (var cat in Category) // iterate through the category list
    {
        using (var client = new HttpClient())
        {
            client.BaseAddress = new Uri("https://api.datamuse.com/words?rel_trg=" + cat + "&max=100");
            var responseTask = client.GetAsync(client.BaseAddress); // call Datamuse API
            responseTask.Wait();
            var result = responseTask.Result;
            if (result.IsSuccessStatusCode)
            {
                var parse = result.Content.ReadAsAsync<List<WordClass>>(); // parse the JSON result as a list of WordClass object
                parse.Wait();
                WordsList = parse.Result;
                foreach (var word in WordsList) {
                    Dict[cat] += word.Word + " "; // add the related words as dictionary value
                }
                Dict[cat] = Dict[cat].Trim(); // trim extra whitespaces
            }
        }
    }
    string jsonPath = Path.Combine(HttpRuntime.AppDomainAppPath, @"App_Data\Category.json"); // save the dictionary into JSON file
    System.IO.File.WriteAllText(jsonPath, JsonConvert.SerializeObject(Dict));
    return Dict;
}

public class WordClass
{
    public string Word { get; set; }
    public int Score { get; set; }
}
```