Summer Practice Report
Department of Computer Engineering
METU

Fatih YILDIZ – 2306793

Summer 2020

1)Table of Contents

2) CRS Soft: Cross Rational Solutions

CRS Soft is a company located in Yildiz Technical University Teknokent in Istanbul. It was founded in 2010 and it specializes in software development and consultation. Company mostly uses Microsoft technologies (.NET) to develop software. Services such as e-invoice are also offered by the company.

3) Introduction

I have practiced my summer internship between 04.08.2020 and 11.09.2020. I was working under the title “Web Development Intern”. Thus, I was expected to create a web app that responds to the needs of the society.

3.1 Getting Started with Web Development

Since I did not have previous experience with web development or .NET technologies, I started going through basic web development courses and documentations. I faced a lot of new concepts and keywords such as HTTP, JSON, AJAX, etc. After following extensive courses and doing exercise projects, I was ready to start developing my own project.

3.2 Coming Up with an Idea

We were free to come up with our own app idea provided that we utilize .NET technologies. While I was going through introductory web development courses, I was thinking of an app idea to build during my internship. In the end, I decided to build a website that lets travelers share their experiences with public, explore places, and create their traveler profile which includes their contact details and experiences. I decided to name my app “BeenThere”.

3.3 Utilized Tools

Programming Language

The app was to be created using .NET technologies, so I used ASP.NET Core to create my website. C#, F# and Visual Basic can be used to create websites with ASP.NET Core. I decided to use C# since it has the most popularity and documentation.

Database

The app includes a database to store travelers' data and allow CRUD operations (create, read, update, delete). I was recommended to use Microsoft SQL Server to hold my data. Since I am using a computer with MacOS, I used a Docker container to access SQL Server.

Database Access

I utilized Entity Framework Core to establish database access. LINQ queries provided by Entity Framework are especially useful to manipulate data.

4) Analysis and Feasibility Study

Every app that is to be created serves a purpose. Most of the time, the purpose is to create capital gain. Whatever the purpose is, the target group of the app and the needs of this group should be analyzed carefully in order to come up with a sustainable app that will thrive in the market.

Most of the time, this is possible by conducting surveys in big scale and analyzing big data collected by extensive methods. However, since it was not possible for me to do such big researches, I tried to analyze the data and the ideas that I was in reach of.

4.1 Target Audience

BeenThere appeals to;

- Travelers looking for a blog to record their experiences and build an audience.
- Travelers looking for first-hand information.
- People who plan to take part in voluntary working positions or exchange programs and want to gather information about the host.
- People who want to follow their favorite traveler's adventures.

4.2 Costs and Profits

Since the Google Maps API is used, there is a charge each time a user queries an information on my website. Since this is a pay-as-you-go subscription, it is highly scalable. The fact that it is a collaborative blog makes BeenThere inexpensive to maintain. No content-creator needed to be paid. However, active moderators may be needed to verify the data provided by the users. Moderators can be chosen from the active users.

The app is created to provide reliable and first-hand information to travelers. From this perspective, paid partnership between businesses and BeenThere to promote the business violates the ground rule. So, no profit is projected in short-term.

4.3 Differences to Competitors

The idea I came up with had major global competitors such as Expedia and TripAdvisor. In order to be successful in the market, It had to have some differences to these competitors.

Advantages to Competitors

- Social media, for sure, gives people the ability to share their discoveries or experiences about a place. But the audience is only the followers of the person. With BeenThere, any public experience will be easily reachable to any adventure-seeker.
- In social media, posts don't contain detailed, adequate information about the details of the place. Most people do not provide exact location, phone number or prices in the posts. With BeenThere's detailed post templates, it's easier to keep track of places.

Disadvantages to Competitors

- Location's critical data (phone number, address, etc.) provided by users may not always be up-to-date or correct. The data might have to be manually checked and corrected by admins or moderators.
- It will take a lot of time to gather enough users to have a wide net of reviewed places. During this time, search results may not return useful data.

5) Implementation

In order to have a loosely coupled and modular codebase, a 3 layered architecture is used. These are:

- User Interface Layer
- Data Layer
- Entity Layer

5.1 User Interface Layer

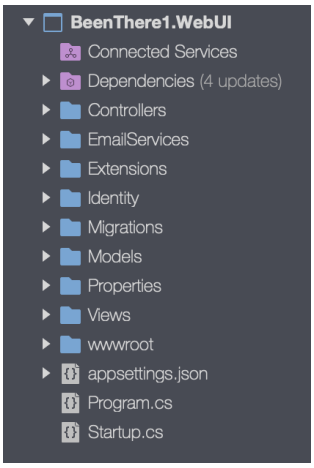

Figure 1

This layer contains the files in Figure 1.

Following the MVC (Model-View-Controller) pattern, this layer includes the folders for Controllers, Models, and Views. Also, our static files and email service folders reside at this layer.

5.1.1 Controllers

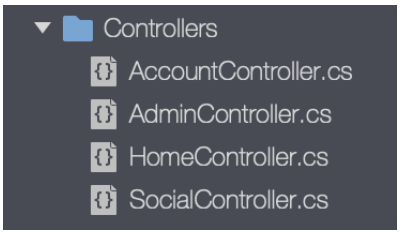

Figure 2

Controllers are the components that handle the traffic between views using models. Certain validations can be done in controllers.

Different controllers are used to represent different classes of actions. As can be seen in Figure 2, these are:

- HomeController: Deals with the landing page of the website, which includes a map powered by Google Maps JS API.
- AccountController: Deals with the actions related to user register, sign in, password reset etc.
- AdminController: Deals with the admin actions such as creating and assigning Roles, creating and deleting users etc.
- SocialController: Deals with the blog post actions such as post/comment create, edit, delete etc.

HomeController

Constructor Method

In order for us to be able to access methods defined in Data Layer Repositories, these dependencies are injected as parameters to the constructor function of HomeController. Injected services can be seen in Figure 3.

```
public HomeController(ILogger<HomeController> logger,
                      IExperienceService experienceService,
                      ILocationService locationService,
                      ICategoryService categoryService,
                      UserManager<BeenThereUser> userManager
){
    _logger = logger;
    _ExperienceService = experienceService;
    _LocationService = locationService;
    _CategoryService = categoryService;
    _UserManager = userManager;
}
```

Figure 3

Index Action

Index Action is the action that controls the landing page of the website. When the website is visited, “Home/Index” method (Figure 4) is called and the map is loaded. In order to display the markers on the map, they must be acquired from the database.

```
[HttpGet]
public IActionResult Index()
{
    var markerDisplayObject = _LocationService.GetMarkerDisplayList(); 1

    ViewBag.Markers = BuildDisplayString(markerDisplayObject); 2

    var countryList = _LocationService.GetAll().Select(s => s.Country).Distinct().ToList();

    var mapModel = new MapModel()
    {
        Countries = countryList.OrderBy(q => q).ToList(),
        AllCategories = _CategoryService.GetAll(),
        ChosenCategoryArray = new int[0]
    };

    return View(mapModel);
}
```

Figure 4

“GetMarkerDisplayList” method (number 1 in Figure 4) gets all the markers from the database through “_LocationService”, which is an interface that includes the function signatures related to the operations on locations. In order for the JS script in the View to be able to read the marker data, the data must be converted to a universal data format, JSON. “BuildDisplayString” method (number 2 in Figure 4) converts the data into JSON format and then to a string since carrying complex data types via ViewBag is not possible. ViewBag is a transfer method between the controllers and the views.

Along with the marker data, also country and category data are transferred to the view, this time using the Page Model. This way, we can display the country and category list to enable filters (Figure 5).

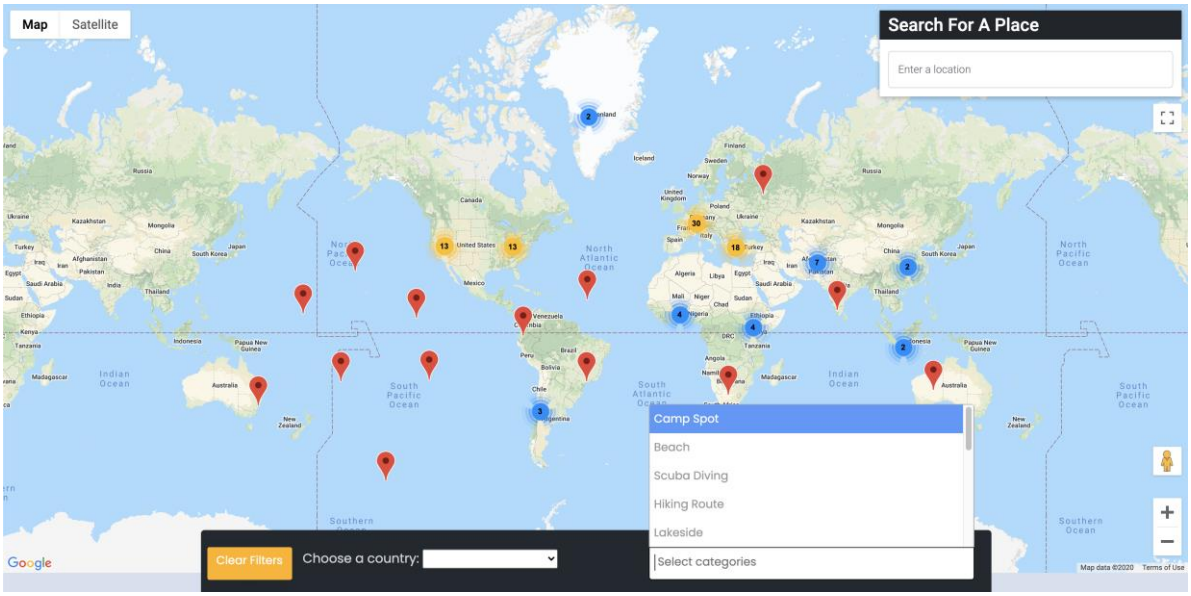

Figure 5

FilterMarkers Action

In order to enable filtering without refreshing the page, I used AJAX.
Whenever a change is made on the form which includes the country and category select fields, “filterMarkers()” function is called.

```
function filterMarkers(e){

    var valdata = $("#filterMarkerForm").serialize();

    $.ajax({
        url: "/home/filtermarkers",
        type: "POST",
        dataType: 'json',
        contentType: 'application/x-www-form-urlencoded; charset=UTF-8',
        data: valdata,
        success: function (data) {

            DeleteAllMarkers();
            markersToDisplay = JSON.parse(data['markerString']);
            initMap();
        }
    })
};
```

1 2 3

Figure 6

This function serializes the form data and triggers an AJAX function.
“url” property (number 1 in Figure 6) specifies the action to call and “type” property (number 2 in Figure 6) specifies the request type which is “POST” since we are sending form data.

```
[HttpPost]
public IActionResult FilterMarkers(FilterModel fm)
{
    string newMarkerString;

    if (fm.filterCategoryIds != null)
    {
        newMarkerString = System.Text.Json.JsonSerializer.Serialize(_LocationService.LocationListByFilters(fm.f
    }
    else
    {
        newMarkerString = System.Text.Json.JsonSerializer.Serialize(_LocationService.LocationListByFilters(null
    }

    var data = new FilterModel()
    {
        markerString = newMarkerString
    };
    return Json(data);
}
```

1

Figure 7

This is the action that receives the AJAX request. It acquires the filtered markers from the database (number 1 in Figure 7) and loads them into JSON data, which is returned to the AJAX call. Upon successful return, AJAX function deletes existing markers and initializes the map with the updated markers (number 3 in Figure 6).

AccountController

Constructor Method

In order for us to be able to access methods defined in Data Layer Repositories, these dependencies are injected as parameters to the constructor function of AccountController. Injected services can be seen in Figure 8.

```
public AccountController(UserManager<BeenThereUser> userManager,
                         SignInManager<BeenThereUser> signInManager,
                         IEmailSender emailSender)
{
    _UserManager = userManager;
    _SignInManager = signInManager;
    _EmailSender = emailSender;
}
```

Figure 8

UserManager is used to manage actions such as create, delete user, reset password, etc.
SignInManager is a service to arrange cookies.
IEmailSender is a service to send mail to the user to confirm account or reset password.

Login Action

```
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Login(LoginModel model)
{
    if (ModelState.IsValid)
    {
        var user = await _UserManager.FindByNameAsync(model.UserName);

        if (user == null)   First Validation
        {
            ModelState.AddModelError("", "User not found.");
            return View(model);
        }

        if(! await _UserManager.IsEmailConfirmedAsync(user))   Second Validation
        {
            ModelState.AddModelError("", "Please confirm your account from the link in your inbox.");
            return View(model);
        }

        var result = await _SignInManager.PasswordSignInAsync(user, model.Password, false, false);

        if (result.Succeeded)   Third Validation
        {
            //check if the return url is empty. if so, redirect to homepage.
            return Redirect(model.ReturnUrl ?? "~/");
        }

        ModelState.AddModelError("", "Wrong password.");
        return View(model);

    }
    return View(model);
}
```

Figure 9

The Login action has some validations so that the user is informed about the invalid fields and not let through. This way, we can prevent collisions or invalid data formats in our database. First, it is checked whether the username entered by the user exists in our database. If not, “User

not found.” Is displayed through the Model Error. Second validation is if the user confirmed his/her account through the link sent to their e-mail address.

Third validation checks if the password entered matches the username. This comparison is made by the method “PasswordSignInAsync” which is included in “SignInManager”. If the login is successful, user is redirected to the URL specified in the model. If it is null, user is redirected to the main page.

Logout Action

```
public async Task<IActionResult> Logout()
{
    await _SignInManager.SignOutAsync();

    return Redirect("~/");
}
```

This is a simple function that logs the user out through SignInManager. The user is redirected to the main page afterwards. Implementation can be seen in Figure 10.

Figure 10

Register Action

Register action gets the register form model and does the necessary validation steps. After creating the user in the database, a unique token is generated (number 1 in Figure 11) and sent to user through the EmailSender service (number 2 in Figure 11). Users can not login unless they confirm their account by clicking the link sent to their inbox.

```
var result = await _UserManager.CreateAsync(user, model.Password);

if (result.Succeeded)
{
    // generate token
1   var code = await _UserManager.GenerateEmailConfirmationTokenAsync(user);
    var url = Url.Action("ConfirmEmail", "Account", new
    {
        userId = user.Id,
        token = code
    });

    // email
2   await _EmailSender.SendEmailAsync(model.Email, "Confirm your BeenThere account.",
        $"Please confirm your account by clicking <a href='https://localhost:5001{url}'>this link</a>.");
    return RedirectToAction("Login", "Account", new { popup = "IsNewAccount" });
}
```

Figure 11

ConfirmEmail Action

```
public async Task<IActionResult> ConfirmEmail(string userId, string token)
{
    if(userId==null || token == null)
    {
        return View();
    }
    var user = await _UserManager.FindByIdAsync(userId);
    if(user != null)
    {
        var result = await _UserManager.ConfirmEmailAsync(user, token);

        if (result.Succeeded)
        {
            return Redirect("/account/login");
        }
    }
    return View();
}
```

1

2

ConfirmEmail action compares the code generated for the user and the code included in the URL user visited through their inbox (number 1 in Figure 12). If they match, account is confirmed successfully and user is directed to the login page (number 2).

Figure 12

AdminController

In order for us to be able to access methods defined in Data Layer Repositories, these dependencies are injected as parameters to the constructor function of AdminController. Injected services can be seen in Figure 13.

```
public AdminController(IExperienceService experienceService,
                       ICategoryService categoryService,
                       ILocationService locationService ,
                       RoleManager<IdentityRole> roleManager,
                       UserManager<BeenThereUser> userManager)
{
    _ExperienceService = experienceService;
    _CategoryService = categoryService;
    _LocationService = locationService;
    _RoleManager = roleManager;
    _UserManager = userManager;
}
```

Figure 13

Admin controller manages the actions related to the admin. Some of these are creating and editing roles, categories, and locations. In order to gather all admin actions in one place, I constructed an admin panel that has User, Role, Location, and Category edit capabilities.

Roles

Roles are necessary because we want to restrict some access for the basic users while giving more extensive access to moderators and the admin. This way, we are able to let moderators and admins maintain and edit the website while allowing users only the core abilities (create post, comment, etc.) on the website.

To enable access restriction, all the actions in the AdminController are protected by the following (Figure 14) Data Annotation. This way, users other than Admin get an “Access Denied” error while trying to reach a restricted area. If we want to grant access to moderators too, we can add “Moderator” keyword in the annotation.

```
[Authorize(Roles = "Admin")]
```

Figure 14

Also, if we want to hide some of the HTML components from some users (like hiding the admin panel link from non-admin users), we can implement the condition in Figure 15 in our page views. Note that hiding the component doesn't necessarily mean we are restricting access completely to the undesired users. An authorization statement like in Figure 14 shall be used with the access method to leave no way out.

```
@if (User.IsInRole("Admin"))...
```

Figure 15

SocialController

In order for us to be able to access methods defined in Data Layer Repositories, these dependencies are injected as parameters to the constructor function of AdminController. Injected services can be seen in Figure 16.

```
public SocialController(UserManager<BeenThereUser> userManager,
                        IExperienceService experienceService,
                        ILocationService locationService,
                        ICategoryService categoryService,
                        ICommentService commentService)
{
    _UserManager = userManager;
    _ExperienceService = experienceService;
    _LocationService = locationService;
    _CategoryService = categoryService;
    _CommentService = commentService;
}
```

Figure 16

SocialController regulates the actions made by users. Some of these are creating/editing experiences, comments, locations and viewing them through the respective View page and creating a personal profile.

ExperienceCreate

```
[Authorize]  1
[HttpPost]
public async Task<IActionResult> ExperienceCreate(ExperienceCreateUpdateModel model,
                                                  int[] selectedCategoryIds,
                                                  IFormFile file)
{
    if (ModelState.IsValid)
    {
```

Figure 17

Some of these actions are regulated by the "[Authorize]" annotation (number 1 in Figure 17). This way, whenever a user that hasn't logged in to BeenThere attempts to use one of these methods, he/she is redirected to the Login page with the necessary return URL. User authorization (making sure the user is signed in) is required because all experiences must a have an owner and our database is structured that way.

Upon creating an experience, users are allowed to choose a location (either by clicking on the map or searching for it) and categories (hiking, fishing, sunset, etc.) describing their visit. Using the "AttachCategoriesToLocation" method (Figure 18), these categories are added to the pool of categories of the chosen location and are displayed on the location summary to guide other travellers planning to visit this location.

```
var locationToBeEdited = _LocationService.GetById(model.LocationOfExperience.LocationId);

_LocationService.AttachCategoriesToLocation(locationToBeEdited, selectedCategoryIds);
```

Figure 18

ExperienceList

In addition to viewing locations on the map on the landing page, I also constructed a page to view listed experiences. Through this page, users can filter experiences by keywords, countries, and keywords to find relevant information (Figure 19).

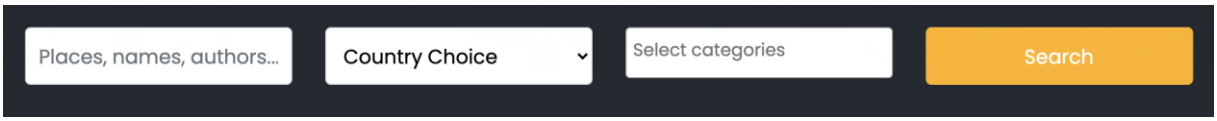

Figure 19

The fields of this form are posted to the ExperienceList function (Figure 20) and necessary filtering is done.

In order to prevent loading all experiences at once and yielding long loading times, I decided to implement paging. The field "pageSize" (number 1 in Figure 20) represents the number of experiences to be displayed in one page. "ExperienceListByFilters" method (number 2) is used to get exactly 6 (6 = pageSize) filtered experiences upon giving the form data as parameters. Other than the actual experiences, we need to know the number of experiences that match our criteria to know how many pages to create. This is found by "GetFilteredExperienceCount" method (number 3). All these information are sent to the View by the PageInfo model. View gets the information and calculates the number of pages to create by dividing the total number of filtered results by the page size.

```
[HttpPost]
public IActionResult ExperienceList(int[] filterCategoryIds, string searchKeyWord, string countryCode, int page = 1)
{
    const int pageSize = 6; 1
    var experiencesToDisplay = _ExperienceService
                .ExperienceListByFilters(filterCategoryIds, searchKeyWord, countryCode, page, pageSize); 2
    var totalNumberOfItemsToDisplay = _ExperienceService
                .GetFilteredExperienceCount(filterCategoryIds, searchKeyWord, countryCode); 3

    var experienceListModel = new ExperienceListModel()
    {
        Experiences = experiencesToDisplay,
        AllCategories = _CategoryService.GetAll(),
        FilterCategoryIds = filterCategoryIds,
        ChosenCountryCode = countryCode,
        PageInfo = new PageInfo()
        {
            CurrentPage = page,
            ItemsPerPage = pageSize,
            TotalItems = totalNumberOfItemsToDisplay,
            filterCategoryIds = filterCategoryIds
        }
    };

    return View(experienceListModel);
}
```

PageInfo to be used to calculate page count and indicate current page

Figure 20

LocationCreate

Locations are created by users to display on the map while creating experiences. If the user is writing an experience about a place that already exists on the map, an additional location is not created. Instead, experience is added to the existing location's experience pool.

Every user profile has a cover map with the visited countries highlighted. When creating an experience, if the place is located in a country that is not yet in the visited countries list of the user, it is added to the list automatically and the country is highlighted on the map.

LocationSummary

LocationSummary displays summary information about the location on a separate page. Categories, experiences and the last visitors of the location is displayed. Steps are shown in Figure 21.

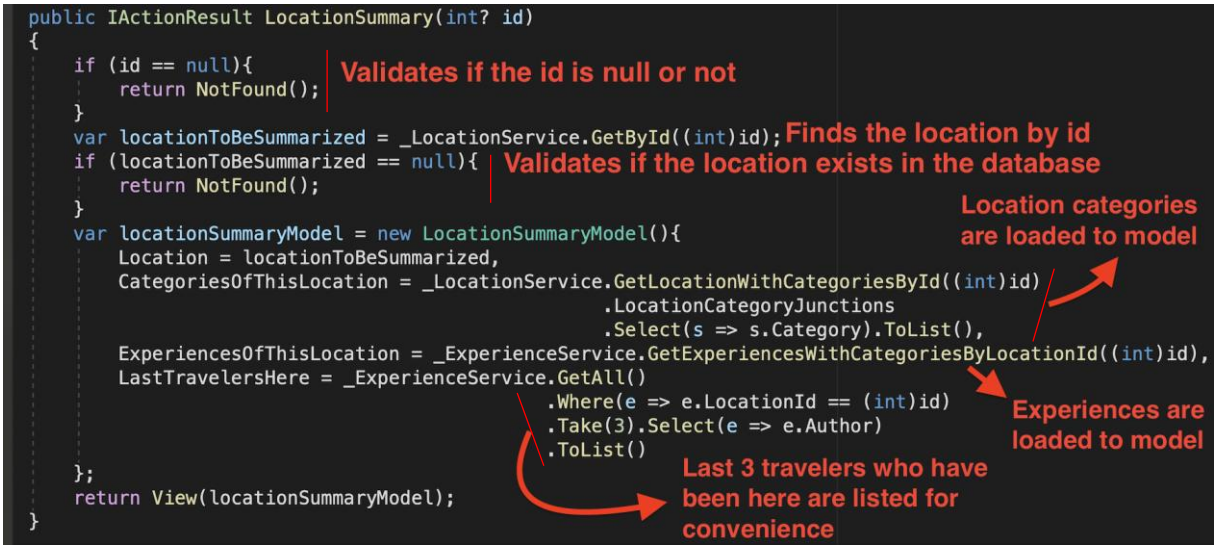

Figure 21

5.1.2 Views

Views handle the app's data presentation and user interaction interface. The backbone of the pages are written in HTML. CSS is used custom styling the page and JS is used to provide interactive functions. ASP.NET Core allows us to embed dynamic code in HTML code so that we can display objects dynamically.

HomeController Views

Index View

This is the view of the landing page of the website. It includes an interactive map and an autocomplete search powered by Google Maps JS API. The location of the map on the page is determined by the placement of the following div element:

```
<div id="map"></div>
```

The map is shown in Figure 22.

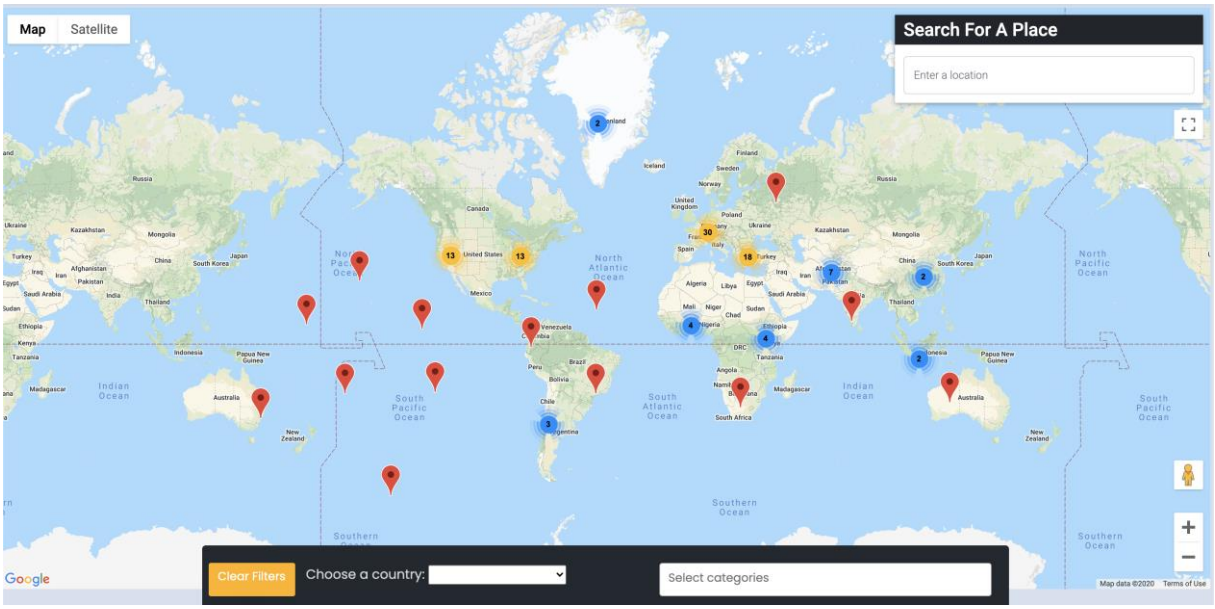

Figure 22

Several features of the Google Maps JS API are utilized. Some of these are marker clustering, autocomplete, reverse geolocation, and place details.

Marker Clustering

As can be seen on Figure 22, in order to prevent displaying an overwhelming number of red markers on the map, adjacent markers are clustered into dots. Upon clicking on the dots, clusters are dispersed, and the red markers inside are displayed. Marker cluster colors range from blue to red in accordance with the marker density.

Autocomplete Feature

Autocomplete feature is utilized via a search box on the top-right corner of the map. It searches for cities, establishments, etc. Whenever a query is made through the search box, a request is made to the Google Maps JS API and results are gathered. The map focuses on the result and views basic details about the place on an info window (Figure 23).

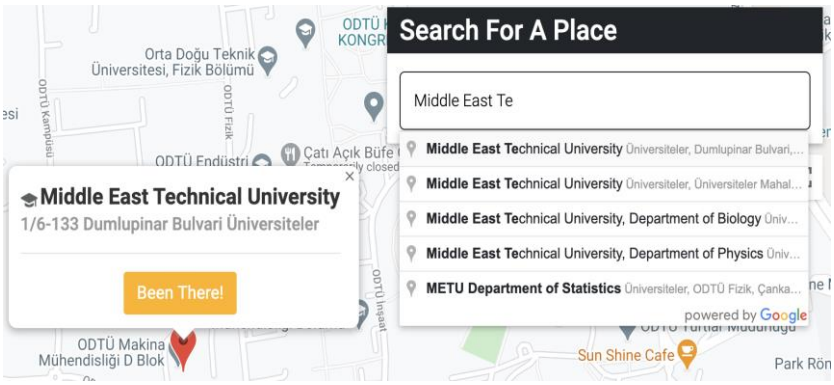

Figure 23

Reverse Geolocation

One of many use cases of BeenThere is to share discoveries with other people. Some discoveries like hidden lakes, secret camping spots or small waterfalls may not have a name hence may not be displayed on Google Maps. I utilized reverse geolocation to be able to pin anyplace on earth. Upon clicking an arbitrary place on the map, clicked place's coordinates are queried in Google Maps database and available information (not necessarily present) is displayed on the marker (Figure 24).

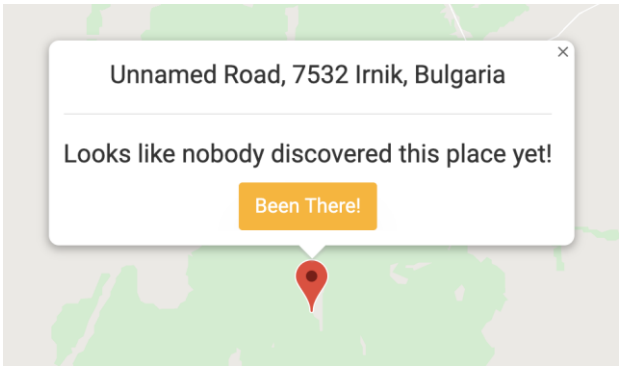

Figure 24

Place Details

When a search is made through the search box (autocomplete widget), additional place details such as phone number, website are queried through Google Maps API Place Details service. Details of the place (address, phone number, website, etc.) are stored in hidden input fields on the View. Whenever the yellow "Been There!" button is clicked (the button in Figure 24), a place is created and saved to the database through LocationCreate action in SocialController. If the place is discovered before (i.e. user is discovering places through the existing markers on the map), a link to the previous experiences at this place is provided along with the union of the tagged categories for this place (Figure 25).

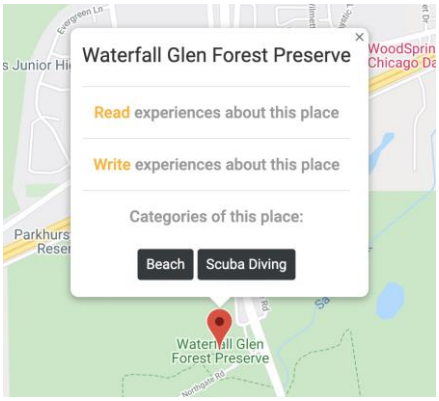

Figure 25

SocialController Views

Profile View

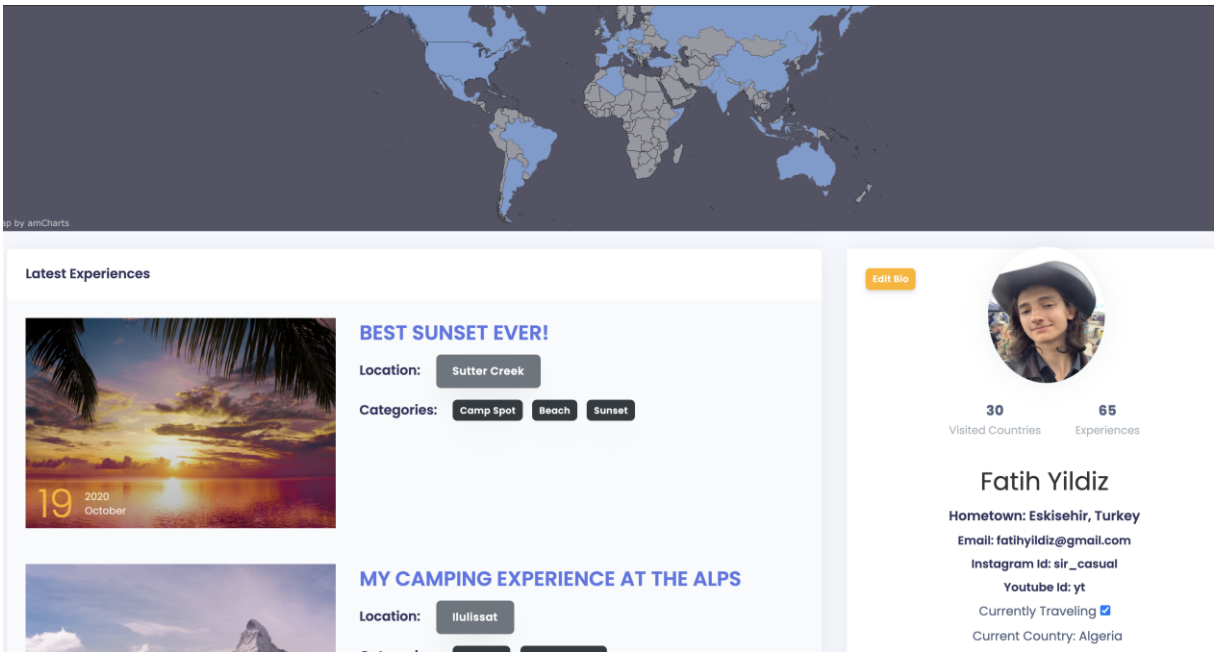

Figure 26

Each traveler has a unique profile that includes the experiences published by the user, visualized display of visited countries, and personal information such as hometown, social media links, and a short biography (Figure 26).

I used “amCharts” to display a responsive map and highlight the visited countries.

The experiences in Figure 26 have the same structure (same html elements and sizes). When displaying different cards, only specific information such as the location name, categories, and the thumbnail are changed. This is why I used a partial view named “_ExperienceCard” and displayed all the experiences trough a for loop (Figure 27).

```
<div class="container">
    <div class="row">
        @foreach (var experience in Model.Experiences)
        {
            @await Html.PartialAsync("_ExperienceCard", experience);
        }
    </div>
</div>
```

Figure 27

Partial views are template codes used to display similar structured elements. An example of it can be seen in Figure 28. Partial views also have models that they display the information according to. Instead of hardcoding the displayed values, Model’s attributes are referenced. This way, we can display multiple experiences using the same model.

```
@model Experience
<div class="card mb-4 shadow-sm">
    <img class="card-img-top" src="~/img/@Model.ImageUrl" alt="" />
    <div class="card-body">
        <h4 class="card-text">@Model.Name</h4>
        <p class="card-text">@Model.Author</p>
        <p class="card-text">@Model.Body</p>
    </div>
</div>
```

Figure 28

When constructing links (anchor tags), I took advantage of the tag helpers provided by ASP.Net Core. One example of it can be seen in Figure 29. “asp-controller” defines the controller that is going to recieve the request, and “asp-action” defines the action to visit inside the controller.

```
<li class="nav-item">
    <a class="nav-link" asp-controller="admin" asp-action="adminpanel">Admin Panel</a>
</li>
```

Figure 29

5.1.3 Models

In order to transfer complex data between controllers and views, Models are used. Models are arbitrary data types that we can define in the Models folder in User Interface Layer. Data annotations such as “[Required] or [Datatype]” (Figure 30) are used to restrict the input user gives. Required flag prevents us from creating empty fields in our database that might be of key importance. Datatype flag gives us the abilities such as hiding password fields with “*”, rejecting any invalid e-mail addresses (such as ones without “@” symbol) and such. Below is the register model we use while transferring data between the Register View and the Register Action in AccountController.

```
public class RegisterModel
{
    [Required]
    public string FirstName { get; set; }

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

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

    [Required]
    [DataType(DataType.Password)]
    public string Password { get; set; }

    [Required]
    [DataType(DataType.Password)]
    public string RePassword { get; set; }

    [Required]
    [DataType(DataType.EmailAddress)]
    public string Email { get; set; }
}
```

Figure 30

Each View can have only one model and the used model is indicated on top of the page (Figure 31).

```
1    @model RegisterModel
2
```

Figure 31

Another tag helper provided by ASP.NET Core is “asp-for” (Figure 32). By adding this to the input field, we bind the input field with the property in the page model, e.g. “RegisterModel”. This lets us exclude properties such as “type” and “name” while constructing input elements because these fields are inherited from the model property with the tag helper.

```
<input class="input100" asp-for="UserName" placeholder="Username">
```

Figure 32

When we send Models with existing fields from the controller and want to display them on the input field, “asp-for” tag catches those inputs and displays them on the input element. This is especially useful when we are editing an entity. This saves us a lot of JavaScript code and time.

5.1.4 wwwroot

This is the root file for the static elements of the website, such as CSS files, JS files, images, and fonts (Figure 33). It is important to avoid using inline CSS or script tags on top of the view and use separate files for these implementations. This way, we have a more organized and easy to manage structure.

Figure 33

5.2 Entity Layer

This layer includes the entities being referred to throughout the construction of the website. Some of these entities are:

- Experience: The blog post that is created by the user. Can be associated with multiple categories. Includes properties such as Title, Author, Body, ImageUrl, LocationId, List<Comment> in order to create one-to-many relations with Comments, and the junction table in order to create many-to-many relations with Categories.

- Category: The categories that blog posts and locations are related to. Has properties such as Name, Url, and the junction tables in order to create many-to-many relations with Experiences and Locations.
- Location: Location that is pinned by the user. Can be found using the autocomplete search or can be an arbitrary location clicked on the interactive map. Has properties such as Latitude, Longitude, PlaceId, List<Experience> in order to create one-to-many relations with Experiences, and the junction table in order to create many-to-many relations with Categories.

One-To-Many Relation

In order to establish one-to-many relations between two entities, a list of the respective entities is added to the main one and the main entity's Id is added as a foreign key to the respective entity. For example, when List<B> is added as a property to A's properties and A's Id is added to the properties of B, this means that A is associated with multiple B's but B's can be associated with only one A.

Many-To-Many Relation

In order to establish many-to many relations between two entities, a junction table which includes the id's of the two entities is constructed. In addition to this, in the application context (in this case BTContext), the two entities are bound to each other using Fluent API (Figure 34). Reference to this junction table is added to each of the entity properties.

```
modelBuilder.Entity<ExperienceCategoryJunction>()
    .HasKey(t => new { t.ExperienceId, t.CategoryId });

modelBuilder.Entity<ExperienceCategoryJunction>()
    .HasOne(j => j.Experience)
    .WithMany(e => e.ExperienceCategoryJunctions)
    .HasForeignKey(j => j.ExperienceId);

modelBuilder.Entity<ExperienceCategoryJunction>()
    .HasOne(j => j.Category)
    .WithMany(c => c.ExperienceCategoryJunctions)
    .HasForeignKey(j => j.CategoryId);
```

Figure 34

5.3 Data Layer

This layer includes three folders:

- Abstract: Contains the Interfaces of the repositories in the Concrete folder.
- Concrete: Contains the implementations of the functions that are referred in the abstract folder.
- Migrations: Contains the migration records to the database.

A repository pattern is utilized to yield separation of concerns and to be able to build loosely couple applications with dependency injection concept. Functions are implemented in the concrete classes and introduced in the abstract classes with interfaces. When we want to use a function, we refer to the interfaces so that we don't have to change our code whenever the implementation changes.

5.3.1 Concrete Classes

Contents of the Concrete folder can be seen in Figure 35.

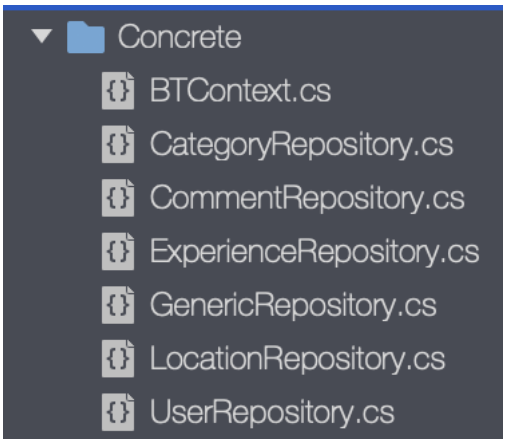

Figure 35

BTContext

BTContext includes the references to the entities the website is using. In order to configure a new entity, after creating a class for it in Entity layer, a reference must be given here.

BTContext also includes the connection string to establish connection to the database (number 1 in Figure 36). Whenever database access is needed, a "using BTContext" statement used. This statement is self-closing so we don't have to break the database access every time we make a connection.

```
public DbSet<Experience> Experiences { get; set; }

public DbSet<Category> Categories { get; set; }

public DbSet<Location> Locations { get; set; }

public DbSet<Comment> Comments { get; set; }

public DbSet<CountryList> CountryLists { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    optionsBuilder.UseSqlServer(
1       @"Server=localhost;Database=BeenThereDB1;User Id=sa;Password=myPassw0rd;MultipleActiveResultSets=true");

    optionsBuilder.EnableSensitiveDataLogging();
}
```

Figure 36

GenericRepository

```
public class GenericRepository<TEntity, TContext> : IRepository<TEntity>
    where TEntity: class
    where TContext: DbContext, new()
{


    public void Create(TEntity entity)
    {
        using (var context = new TContext())
        {
            context.Set<TEntity>().Add(entity);
            context.SaveChanges();
        }
    }

    public void Delete(TEntity entity)
    {
        using (var context = new TContext())
        {
            context.Set<TEntity>().Remove(entity);
            context.SaveChanges();
        }
    }
```

GenericRepository (Figure 37) is a generic repository that contains the basic CRUD methods expected from a repository. This way, whenever a new entity is created, we don't have to implement basic operations again and again. Instead we inherit from GenericRepository and define further functions in the respective Repository (for example, ExperienceRepository).

Figure 37

ExperienceRepository

This is one of many repositories used to implement entity specific functions. For example, the function in Figure 38 gets an Experience Id as parameter and establishes a connection to the database using BTContext. Since there is a many-to-many relationship between entities "Experience" and "Category", the junction table is included to the entity found with ".Where" and the respective Experience is returned. Since "using" statement is self-closing, we don't have to close the connection once we are done.

```
public Experience GetExperienceWithCategoriesById(int id)
{
    using (var context = new BTContext())
    {
        return context.Experiences.Where(i => i.ExperienceId == id)
                                  .Include(i => i.ExperienceCategoryJunctions)
                                  .ThenInclude(i => i.Category)
                                  .FirstOrDefault();


    }
}
```

Figure 38

5.3.2 Abstract Classes

This is the folder that contains the interfaces. Interfaces are the structures we use to build loosely coupled applications and improve testability. Interfaces don't include function implementations. Instead, they only contain the function signatures and indicate that the duties of the class have been fulfilled by a method, doesn't matter which. When we want to use the

class functions, we refer to the interface signatures and not the real function implementations. We saw earlier (for example in Figure 3) that we inject the interfaces as parameters to Controller constructors so that we can user the methods provided by the services. By this we achieve loosely coupled applications.
Function signature includes the return type, function name, and the parameters. Contents of the Abstracts folder can be seen in Figure 39.

Figure 39

IRepository

IRepository is the generic interface that includes the function signatures implemented in GenericRepository. As can be seen in Figure 40, there are no function implementations.

```
public interface IRepository<T>
{
    T GetById(int id);

    List<T> GetAll();

    void Create(T entity);

    void Update(T entity);

    void Delete(T entity);
}
```
Figure 40

IExperienceRepository

This is the interface that includes the signatures of functions which are implemented in ExperienceRepository. These are (Figure 41) the functions that extend the GenericRepository class and provide extra capabilities other than basic Get, Delete, Update etc. functions.

```
public interface IExperienceRepository: IRepository<Experience>
{

    Experience GetExperienceWithCategoriesById(int id);

    void AddCategoriesToExperience(Experience experienceToBeAddedCategories, int[] selectedCategoryIds);

    List<Experience> GetExperiencesByFilter(string categoryUrl, int page, int countPerPage);

    List<Experience> GetExperiencesWithCategoriesByLocationId(int id);
```

Figure 41

5.3.3 Migrations Folder

When we are creating the database for the first time, or when we need to make changes to the existing database (such as adding/deleting columns from an entity or specifying foreign keys etc.) we exercise migrations. Migrations enables us to update the database while preserving the current data in the database.

Since I exercised code-first approach, I created the entities in the Entity Layer and referred to them in BTContext. When I added my first migration, all tables that are necessary to represent my entities are generated in the database.

We start the migration with the following command (Figure 42) in the NuGet Package Manager Console.

```
add-migration TitleColumnAdded -BTContext
```

Figure 42

After the “add-migration” keyword, the migration name is specified. It is important to name the migration meaningfully because we may need to migrate back to old migrations, and we need to be able to figure out which is which. Lastly, the context name, i.e. BTContext, is specified.

When a migration is added, EF Core compares the updated model with the existing model and determines the differences. The migration source files are generated at this stage and saved into the Migrations Folder.

When the following command (Figure 43) is executed in the NuGet Package Manager Console, the differences specified in the generated migration source file are applied to the database.

```
update-database -BTContext
```

Figure 43

If we need to go back to a previous migration, the following command (Figure 44) is executed.

```
update-database PreviousMigrationName -BTContext
```

Figure 44

In Figure 45 we can see an example of a migration source file. This file is generated when the "add-migration" command is executed. In this particular migration, "PlaceId" row is being added to the Locations table in the database (number 1 in Figure 45). When the "update-database" command is executed, the "Up" function is executed. When we need to update to an earlier migration, respective "Down" functions are executed to remove the effects of the migrations from the database (in this particular example, this is done by dropping the column "PlaceId" (number 2 in Figure 45)).

```
public partial class PlaceIdAdded : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.AddColumn<string>(
            name: "PlaceId",
            table: "Locations",
            nullable: true);
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropColumn(
            name: "PlaceId",
            table: "Locations");
    }
}
```

1

2

Figure 45

6) Conclusion

I have experienced a very rewarding and informative summer internship at CRS Soft. I gained a valuable skill, web development, that I can advance my skills and pursue a carreer with. Using C#, I got my hands dirty with OOP (Object Oriented Programming) which eases my job when I decide to learn languages such as Java or C++. I have experienced both front-end and back-end development and decided that I am more interested in the back-end part. I now have a broader perspective in terms of Computer Science and can navigate myself to the areas that I like.