**İSTANBUL TECHNICAL UNIVERSITY**
**Faculty of Computer Science and Informatics**

**INTERNSHIP PROGRAM: Software**

# INTERNSHIP PROGRAM REPORT

**Selin DİNÇ**
**150150229**

**Write your training period: 10 June – 5 July / 2019**

**TABLE OF CONTENTS**

## INFORMATION ABOUT THE INSTITUTION

Related Digital is a leading multichannel campaign management solution provider; To provide a wide range of best-in-class digital marketing technologies and interactive services for many of the world's leading brands. Related Digital was originally founded as an email marketing provider in 2002 and evolved to provide an easy-to-use, integrated data-driven marketing automation platform called the relevant marketing cloud (RMC). After allowing the automation of data-driven marketing Related Marketing Cloud (RMC) multi-channel platform has been transformed into Turkey's leading company in the management of digital marketing

RMC is a platform that provides companies with a single, well-coordinated customer profile by gathering and consolidating all customer data and enabling them to reach the right device and the right channel at the right time with personalized messages. RMC offers companies a user-friendly infrastructure to create fully integrated and high-yield digital marketing campaigns. By combining and analyzing data from all sources, both online and offline, it produces technologies that provide a clear view of where to reach the customer. RMC is fully integrated and enables marketers to initiate and automate revenue-generating lifecycle marketing campaigns - increasing conversions and customer lifecycle value. Marketers can fully understand the best way to reach each customer, combining and integrating data from various sources online and offline.

## INTRODUCTION

I did my second internship at Related Digital. In the 20 days, I spent during the internship, IT department of an omnichannel campaign management solution provider company added a lot to me in terms of recognizing a new sector. I had the opportunity to learn the work order and communication between departments within a technology company.

Related Digital contains three main technology products that provides digital marketing technologies and interactive services such as, Euromessage, Visilabs and Semanticum. Euromessage provides multichannel marketing solutions. Visilab is an analytics-driven cross-channel personalization management technology. Semanticum is a social media monitoring, analysis, reporting and social CRM platform.

During my internship, I had the opportunity to observe all the three parts of company; however, I mainly worked at the Euromessage department. Euromessage department also divide into two parts in it; the maintainers of the already builted product and rewriters of the product with modern programming structures since the present product is a bit old and hard to maintain.

Throughout my internship, I had the opportunity to observe the both of these parts of the department and contribute to them. I learned different things not only from my superior but also from all department employees. An internship at the department that implements modern structures and care for the maintenance showed me how important writing maintainable code while improving and adding new features to a product. I have learned to be able to write code proper to the structures of object-oriented programming.I also had the opportunity to understand agile software development environment.

## DESCRIPTION AND ANALYSIS OF THE INTERNSHIP PROJECT

### 1. RESTful Email API Service

My internship project was coding an Email service that will be used throughout the company. Through this service, all applications developed within the company will meet the needs of sending mail. Thanks to the fact that the process of sending email is carried out through a single service, changes in the delivery structure will not affect other applications. In addition, if the company has general decisions to send (for example, no mail will be sent between 20:00 and 07:00), these decisions will be executed from a common point. The diagram of my project is shown in the Figure 1.

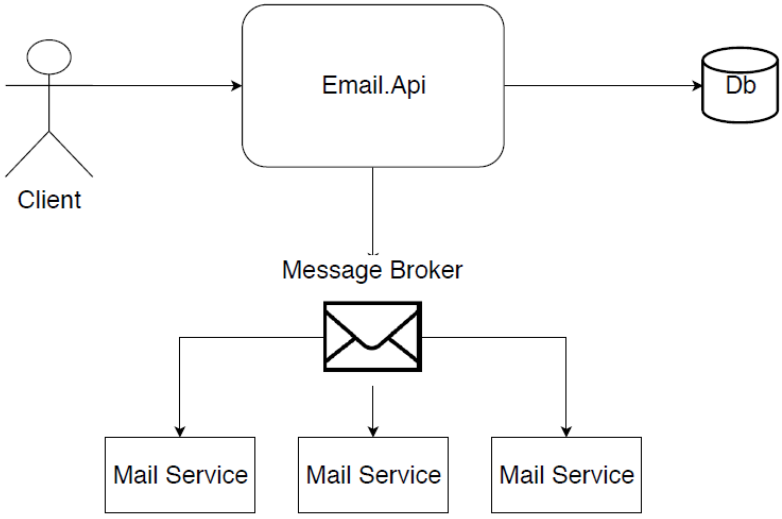



**Figure 1**: Diagram of Email Service Project

### 1.1 Requirments of the Project

**1.1.1Email Sending Feature:** In the first phase, a structure that accepts all mail addresses will be established. E-mails (maximum 1 month) can be scheduled for later. In the first phase, all content will be sent without checking.

**1.1.2 Email Status Inquiry Feature:** An id value must be created when the mail is generated. With this id value, the status query can be sent. Mail status can be pending, ongoing, canceled and sent.

**1.1.3 Email Cancellation Feature:** An id value must be created when the mail is generated. With this id value, mail can be canceled.

**1.1.4 Technical Requirments:** Project has some technical requirments such as, must comply with the principles of S.O.L.I.D, must be developed according to REST rules and it must be scalable. Since I need to provide a code that applies these rules, first I started to research about the topics that I should use on my code.

## 2. S.O.L.I.D Principles of Object Oriented Design

S.O.L.I.D principles are coding standards that developers use when they want to develop a software that has good design. By implementing these design principles programmers develop software that maintainable, understandable and extendable. These principles also provide an opportunity to work easily on an agile environment.

**2.1 Single Responsibility Principle (S):** This principle refers that a class/method should have only have one responsibility. However, this doesnt mean that each class have only one method. This means every method should directly serve to the purpose/responsibility of class. In other words, they all work for the same goal. Applying this principle makes easier to write test cases, leads fewer dependencies and better organization.

**2.2 Open For Extension, Closed For Modification Principle (O):** This principle refers classes/modules/functions should be open for extension and closed for modification. This means a class/module/function should be easily extendable without modifying the class itself.

**2.3. Liskov Substitution Principle (L):** This principle refers that every subclass/derived class should be substitutable for their base/parent class. Basically, it takes care that while coding using interfaces in our code, we not only have a contract of input that the interface receives but also the output returned by different classes implementing that interfac, they should be the same type.

**2.4 Interface Segregation Principle (I):** This principle refers that larger interfaces should be break into smaller ones. By this way we make sure that the classes that we implement only need to be concerned about the methods that are interest to them. A client should never be forced to implement an interface that doesn't use or clients shouldn't be forced to depend on methods they don't use.

**2.5 Dependency Inversion Principe (D):** This principle refers that entities must depend on abstractions not on concretions. It states that the high level module must not depend on the low level module, but they should depend on the abstractions. By using this method modules can be easily changed by other modules just changing dependency module. This principle allows decoupling which is loose coupling. Decoupling minimizes the communication and dependency between classes. This method also allows us to easily test our program.

## 3. Application Programming Interface (API)

API is an interface or a protocol for communication between a client and a server. It's aim is to simplify the process of building the client-side software. Basically, an API is used for communicate the applications with each other. Figure 1 shows that an API is not a database or a server, it is just the code that regulates the access points for the server.

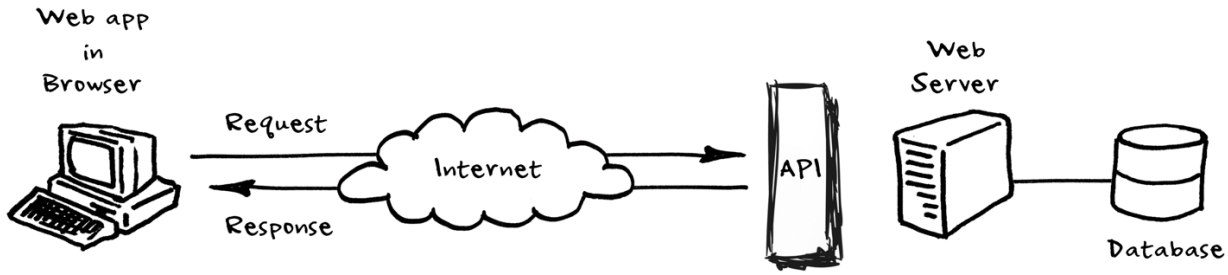



**Figure 2:** Web based APIs

APIs has different elements such as functions, protocols and tools that allow developers to build applications.

We can call an API a contract between client and server which defines specific responses to specific requests from the client. One of the best part of using an API for a developer is, they don't need to understand the underlying implementation because API abstracting that part. Developer only need to care of the objects and actions by using an API. This way really accelerates the process of developing applications since developers don't need to implement the underlying parts. There are different APIs for different type of systems such as operating systems, libraries and Web. In my project I used a Web API. The model of Web Based API is shown in the Figure 2.

### 3.1 Web APIs

A Web API is an interface that provide communication by using Internet and protocols that specific to the Web. They take a request from the client and return data as a response to that. Web APIs make the resources available on the Web appear as local resources. This way they allow client to get data from the outside resources. Generally, Web APIs served through HTTP intefaces. What APIs do is defining a set of endpoints, request messages and response structures.

#### 3.1.1 Why Would We Need an API?

If we create an application or just as a client want to access another outside app's resource/data/functionality, then we could reach the app's company and ask for the data/resource as a spreadsheet. Then we need to find a way for applying this data/resource to our application. We could store this data in our database, however the data would become outdated very quickly. It would be very hard to keep data that we took from company up to date. The solution for this problem is that the company would provide us a way (query) to get their data from their application and we would easily use that data for our purposes. Also, by this way data would stay up to date all the time. It all can be done easily thanks to APIs.

#### 3.1.2 What Is the Difference Between an API and a Regular Database-Backed Project?

An API is doesn't have to contain the frontend part which means ther is no need to show HTML and CSS to the client/user. Request or retrieve data can be done by without frontend. Instead of using frontend you can just send a HTTP request to the server.

### 3.2 REST APIs

REST is stands for Representational State Transfer. REST is an architectural design style for providing standards between computer systems on the web. By using REST architecture we make easier for the systems to communicate with each other. The main work principle of REST API is when you search something you get a list of responses back from the service that you requested from. REST architecture mainly defines what API looks like. While creating a RESTful API, developers follow the set of rules that REST architecture provides. In a RESTful API, each URL is called request and data sent back to the client is response.

#### 3.2.1 RESTful API Architecture

There are some certain architectural characteristics of a RESTful API such as a uniform interface, statelessness and seperateness, cacheable, client-server, layered system and code on demand. The code on demand part is the only optional constraint among the others. If any of these constraints don't satisfied by the system, then we couldn't call it a RESTful architecture. The model of RESTful Architecture is shown in the Figure 3.

**3.2.1.1 Uniform Interface :** This one is one of the key constraints that help us to differentiate between a RESTful API and Non-REST API. This characteristics suggest that there should be only one way to interact with a particular server, regardless of device or application type (website, mobile app).

**3.2.1.2 Statelessness :** This characteristics suggest that the status required to process the request is in the request and the server will not store aanything related to the session. In REST, the client must contain all of the server's information to fulfill the request as part of the query parameters, headers, or URI. Statelessness provides more availability because the server does not need to maintain, update, or forward this session state. There is a disadvantage that the client must send a lot of data to the server, thus reducing the scope of network optimization and requiring more bandwidth.

**3.2.1.3 Seperation of Client-Server :** This characteristics suggest that the implementation of the client and server can be done independently of each other without knowing the other. This means that client-side code can be changed at any time without affecting the operation of the server, and vice versa. It can be modular and separate, as long as both parties know which messages to send to the other. Using the REST interface, different clients reach the same REST endpoints, perform the same operations, and receive the same responses.

**3.2.1.4 Cacheable :** This characteristics suggest that each response should include whether the response can be cached and how long the response can be received by the client. The client returns the data from the cache for a subsequent request, and there is no need to resend the request to the server. Well-managed caching improves availability and performance even more, partially or completely eliminating some client-server interactions. However, sometimes there is a possibility that the user can retrieve old data.

**3.2.1.5 Client-Server :** This suggest that a RESTful application must have a client-server architecture. A client is a person who requests resources and is not interested in the data storage space within each server, and who maintains server resources and is not interested in

the user interface or user status. They can develop independently. The customer does not need to know anything about business logic and does not need to know anything about the server's front-end user interface.

**3.2.1.6 Layered System :** This characteristics suggest that an RESTful application architecture must consist of multiple layers. Each layer knows nothing about a layer other than the instant layer, and there may be too many intermediate servers between the client and the end server. Intermediary servers can increase system availability by providing load balancing and providing shared caches.

**3.2.1.7 Code on Demand :** This is the only characteristics that an optional among the others. According to this characteristics, servers may also provide executable code to the client. On-demand code samples can include compiled components such as Java applications and client-side scripts such as JavaScript.

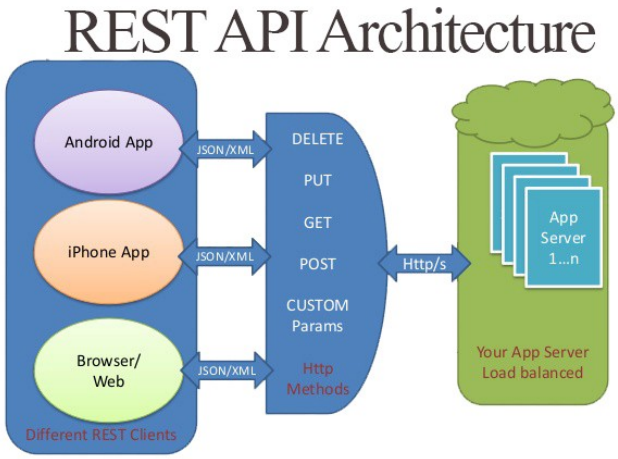



**Figure 3:** REST API Architecture

### 3.3 Creating REST API Endpoints

Request (URL) consist of four things; the endpoint, the method, the headers and the data (body). The endpoint as Figure 4 shows is the URL which is requested by client. Instead of action/verb based approach, REST is based on the resource or noun. This means that an URL of a RESTful API should always end with a noun.

The verbs form the HTTP are used to distinguish the action. Some of the HTTP verbs are GET, PUT, POST, DELETE. These verbs are used to modify the resources. By this way, it is clear that what a developer going to do just by looking the endpoint and the HTTP method used. Using plurals need to be preferred by the developer. By this way, developer can make the URL consistent throughout the application. Also, a developer need to be send proper HTTP codes to indicate a success or error status.

| URI | HTTP VERB | DESCRIPTION |
|---|---|---|
| api/users | GET | Get all users |
| api/users | POST | Add a user |
| api/users/1 | PUT | Update a user with id = 1 |
| api/users/1 | DELETE | Delete a user with id = 1 |
| api/users/1 | GET | Get a user with id = 1 |

**Figure 4:** Endpoints

### 3.3.1 HTTP Verbs

**GET**: It retrieves one or more resources defined by the request URL and it can cache the received information. Example endpoint is shown in the Figure 5b, 5c and 5d.

**POST:** Creates a resource by submitting a request, and in this case the response cannot be cached. This method is not secure when no security is applied to the endpoint because it will allow anyone to create a random resource by sending. Example endpoint is shown in the Figure 5a.

**PUT:** Updates an existing resource on the server specified by the request UR. Example endpoint is shown in the Figure 5e.

**DELETE:** Delete an existing resource on the server specified by the request URI. Always return an appropriate HTTP status for each request.

```
[HttpPost("send-email-task")]
```

**(5a)** Post Endpoint

```
//GET mail
[HttpGet("")]
```

**(5b)** Get Endpoint

```
//GET mail/id
[HttpGet("{id}")]
```

**(5c)** Get/Id Endpoint

```
//GET mail/id/status
[HttpGet("{mailId}/status")]
```

**(5d)** Get/Id/Status Endpoint

```
//PUT mail/id/status
[HttpPut("{mailId}/status")]
```

**(5e)** Put Endpoint

**Figure 5 :** Example endpoints from my RESTful Email API

## 4. Software and Development Environments

Here I am going to explain about the software and development environments that I encountered and used during my internship period.

### 4.1 .Net framework

NET is a software framework designed and developed by Microsoft. .NET framework is a virtual machine used to compile and execute programs written in different languages such as VB.Net and C#. It is used to develop form-based applications, Web-based applications, and Web services. There are several programming languages on the Net platform, VB.Net and C # are the most common ones. Windows, phone, web and so on. Used for applications. It provides many functionality and also supports industry standards.

In my internship I coded my Email API with C# and .NET framework. This makes my application a platform dependent one. Platform dependent means that the application you developed with a specific programming language will run only on particular operating system. Since the .NET framework makes my application platform dependent, my application is only be able to run on Microsoft-based operating system. There is a way to make your application platform independent still using the .NET framework which is called Mono framework. Mono framework makes your application can run on any operating system. However, since it is a third party software and I needed to paid for it, I didn't used this software on my own project.

### 4.2 C#

C # is a general-purpose, modern, object-oriented, high-level programming language. C # is syntactically very similar to Java and is easy for users with C, C ++ or Java knowledge. C # is widely used to develop web applications and desktop applications. It is one of the most popular languages used on the professional desktop.

### 4.3 Visual Studio

Visual Studio is an integrated development environment (IDE) developed by Microsoft. VS is used for developing computer programs, websites, web apps, web services and mobile apps. VS has many built-in languages such as c++. Visual basic, .net, c#, html ,css,etc. In my case I used c# and .net. It also includes a debugger which can be used for source-level debugging and machine-level debugging.

### 4.4 Octopus Deploy

Octopus Deploy is an automated deployment and release management server. It is designed especially for ASP.net applications, windows services and databases. By using this tool company that I worked aims to achieve repeatable and reliable deployments. This tools helps to ensure the releases tested before the deployment, schedule deployment for a specific time and block any part that cause a release broke. This tool make the developer ensure that the code that goes to production is exactly the one you tested. Also this tool helps to make possible that your deployments are consistent between environments.

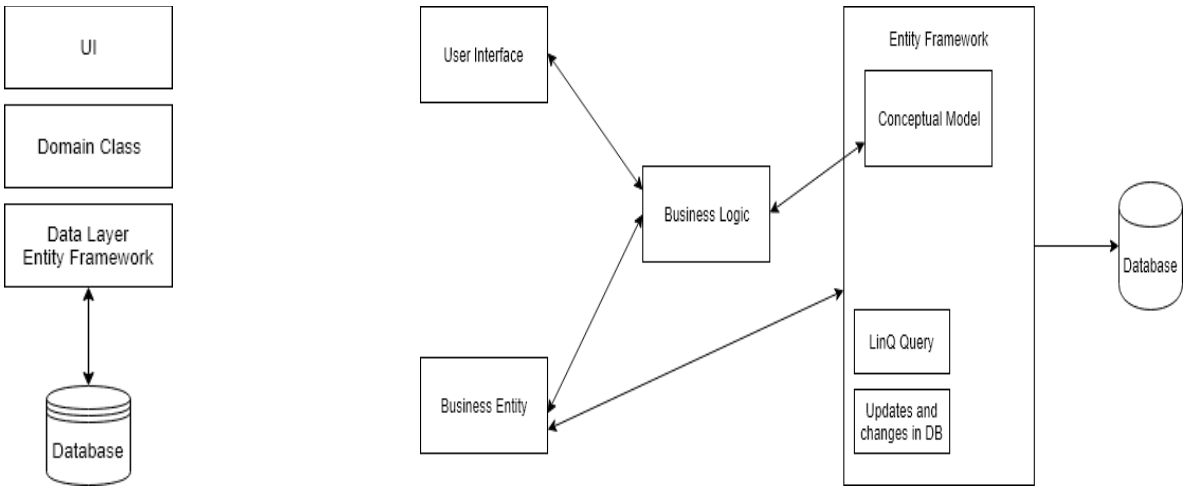
### 4.5 Bitbucket

Bitbucket is a web-based version control repository hosting service. it can handle the projects that use Mercurial or Git revision control systems for the source code. It is similar to GitHub. My company was preffered to use Bitbucket instead of GitHub. Because GitHub is focused around public code, and Bitbucket is focused on private code. Bitbucket has mostly enterprise and business users. By using Bitbucket, company can create and manage repositories through the website or the command line, create merge and pull requests and

open issues or discussions. Basically, it was the place that my company choosed to store their code.

### 4.6 Entity Framework

The Entity Framework is an open-source object-relational mapper framework for .NET applications that are supported by Microsoft. It improves developer productivity because it allows developers to work with data by using objects of field-specific classes without focusing on the underlying database tables and columns where this data is stored as shoen in the Figure 6b. It eliminates the need for most of the data access codes that are often used to interact with the database that developers typically need to type. Using the domain-specific object, it provides developers with an abstract level for working with relational tables and columns. It also reduces the code size of data-specific applications and also increases using the readability of the code. The reasons of I used entity framework in my project was, it was an easy way to access data as shown in the Figure 6a, it generated automated code, it is platform independent and it reduced the development time.




**(6a)** Where the Entity Framework in your app **(6b)** Entity Framework Conceptual Model

**Figure 6:** The Entitiy Framework

### 4.7 NuGet

NuGet is a tool that manage packages for Microsoft development platform. It is a Visual Studio extension. It support programming languages such as .NET Framework packages and native packages written in C++. NuGet provide ability and tools to produce, host and consume packages. NuGet contains compiled code (DLL) that needed in the projects that consume packages. Developers who share code create packages and publish them to a public or private host. Package consumers receive these packages from the appropriate hosts, add them to their projects, and then call the function of a package in project codes. NuGet itself then overcomes all intermediate details as Figure 7 shows.

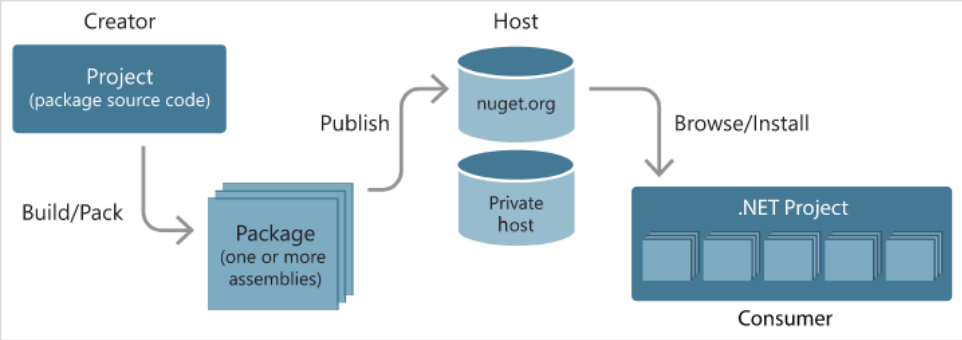



**Figure 7:** NuGet: How it works?

### 4.8 Swagger

Swagger is an open-source software framework that helps developers to design, build, document, and consume RESTful web services. It is a tool that simplifies API development for developers. I used Swagger-User Interface to display my Email API. Swagger UI takes existing JSON and YAML document and creates interactive documentation as Figure 8 shows.

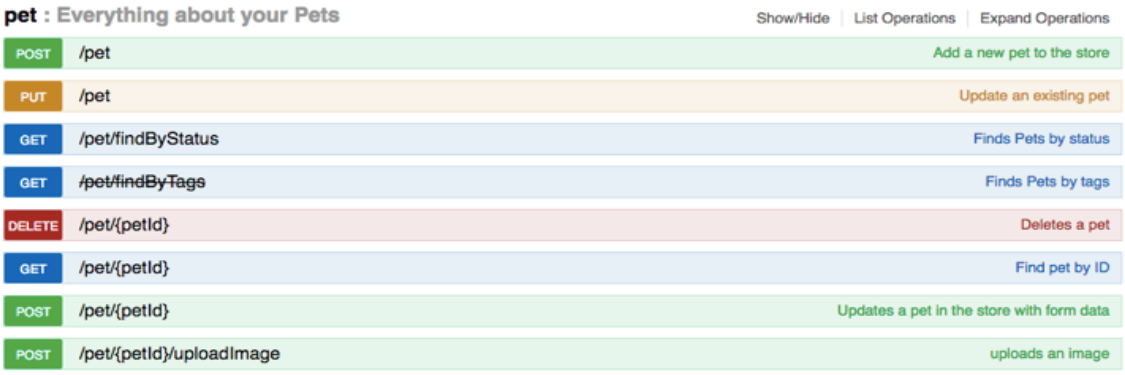



**Figure 8:** Example Swagger-UI

Here as Figure 13 shows we see the Swagger-UI example. Each method is expandable (GET,POST,PUT,DELETE). We can see the full decription of the parameters just by clicking them. One of the major benefit of using Swagger UI is that UI presentation of the API is become really user friendly with this tool. Swagger UI keeps all the logic complexitiy behind the screen. Main reason that I am using this tool is, Swagger enables me to execute and monitor my Email API requests that I sent and the results that I've seen in exchange. Also ,it helped me to test my endpoints too. Swagger UI represents APIs within browser, this is another practical way to use this tool as Figure 9 shows.

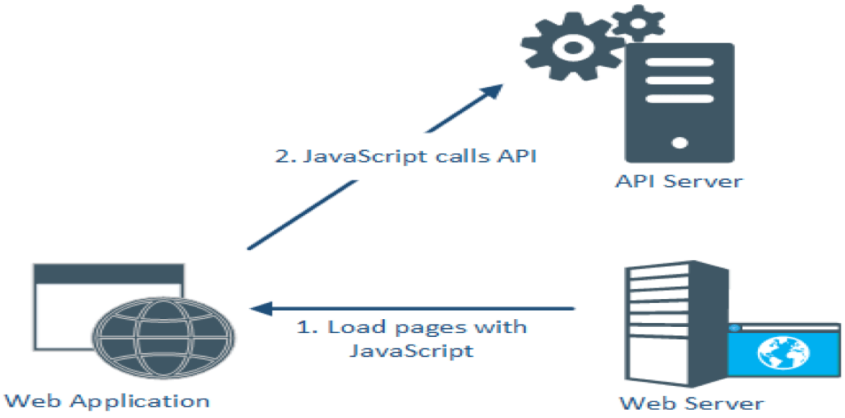



**Figure 9:** How to show your API in a browser?

When you open the web page, the browser loads the web page from the web server and triggers requests to the API server to retrieve data from the database. SwaggerUI is automatically generated from any API defined in the OpenAPI Specification and can be viewed in a browser as Figure 9 shows.

### 4.8.1 How did I apply Swagger-UI to my Email API?

I added the Swagger to the middleware of my code which is Startup.cs which is bootstrap code for my application under the ConfigureServices method. ConfigureServices method gets called by the runtime. This method is used to add services to the container. To enable the Swagger for my API, I added the AddSwaggerGen as Figure 10 shows under the ConfigureServices method. Then it gave me an error which indicates that IServiceCollection doesn't contain the definition for AddSwaggerGen. To solve this problem, I added a NuGet oackage called Swashbuckle.AspNetCore which is used to add Swagger tools for documenting APIs built on ASP.NET Core. After I installed the NuGet package the error was gone.

```
services.AddSwaggerGen(c => c.SwaggerDoc("v1", new Info { Title = "Email Api", Version = "v1" }));
services.ConfigureSwaggerGen(options =>
{
```

**Figure 10:** AddSwagerGen

After that, I added the UseSwagger and UseSwaggerUI under the Configure method as Figure 11 shows which is called by the runtime. This method is used to configure the HTTP request pipeline.

```
app.UseSwagger();
app.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint("/swagger/v1/swagger.json", $"Email Api v1");
});
```

**Figure 11:** UseSwagger and UseSwaggerUI

That way I enabled the Swager in my code. Thus, I was able to see my Email API in the Swagger UI page when I run my program as Figure 12 shows.

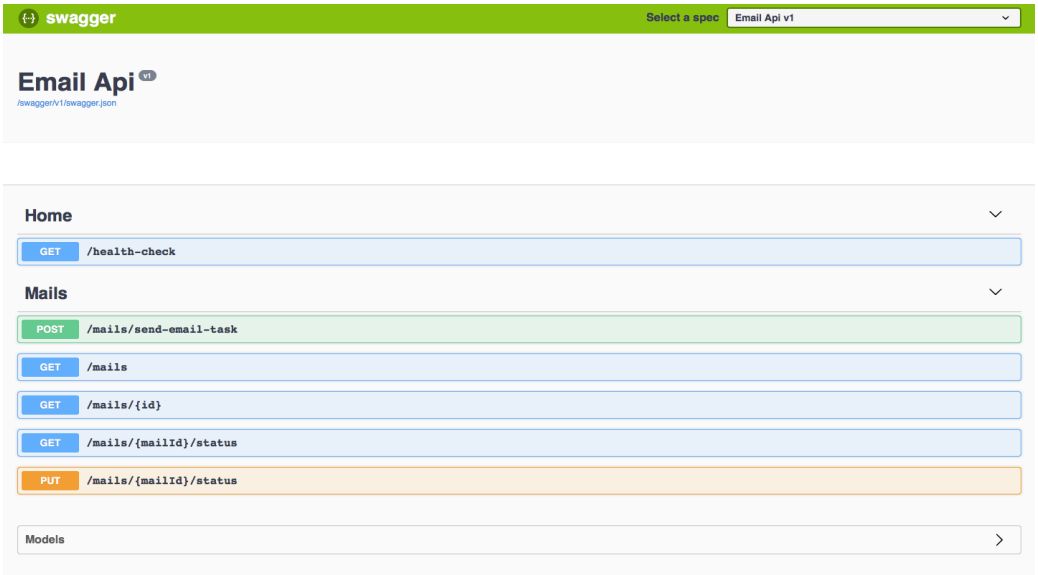



**Figure 12:** Swagger UI of My Email API

### 4.9 MassTransit

MassTransit is an open source distributed application framework for .NET. MassTransit makes it easy to build applications and services that benefit from message-based, loosely–coupled asynchronous communication for higher availability, reliability, and scalability. MassTransit is an easy way to build asynchronous services using message-based speech models. Message-based communication is a reliable and scalable way to implement a service-oriented architecture. To implement MassTransit to my program, I added AddMassTransit under the ConfigurationServices to register the MassTransit as shown in the Figure 13. Then by adding AddConsumers, I registered my consumers in the ASP.NET Core service collection. This part of the code does the followings; hosts service to start and stop the bus following the apps lifecycle, registers the bus instance as a singleton for the required references, adds health checks for the bus instance and receive endpoints and lastly tells the bus to use the ASP.NET Core logger.

```
services.AddMassTransit(serviceCfg =>
{
    serviceCfg.AddConsumers(typeof(Startup).Assembly);
    serviceCfg.AddBus(serviceProvider =>
    {

        IBusControl bus = Bus.Factory.CreateUsingInMemory((cfg) =>
        {
            cfg.ReceiveEndpoint("mail_scheduled_queue", e =>
            {
                e.ConfigureConsumer<SendMailService>(serviceProvider);
            });
        });
        return bus;
    });
});
```

**Figure 13:** AddMassTransit to My Email API

## 5. Detailed Explanation of My Email API

My RESTful Email API is consist of many layers which is a requirement for the REST architecture. I will start to explain my program starting by showing each layer. After that I will explain these layers detailed as possible. I will try to mention about the most essential parts for my project.

### 5.2 General Work Principles of RESTful Email API

1. Event arrives to the API.
2. Check the schedule date.

2a. If the scheduled date is set to a future date then, redeliver.

2b. If the scheduled date is set to a past date then,

2.b.1 Receive the email information.

2.b.2 Set the email status to ongoing.

2.b.2.a If the status is set to ongoing, display the email on the screen and set the email status to the sent.

2.b.2.b. If the email status couldn't set to the ongoing status (e.g. if the email status is cancel or if the status gives an expected error then it couldn't set to ongoing) then, make status error management. If the program encounters an unexpected error then send message to the error-queue.

### 5.3 Structure of RESTful Email API

I have many layers as a requirement of the RESTful architecture. My solution consist of directories such as dependencies, consumers, contracts, controllers, customExceptions, domain, filter, persistence, properties, services, appsettings.json, program and the startup files. I will start with the startup file.

#### 5.3.1. Startup.cs

There are a group of middlewares, small pieces of the application attached to the appliacation pipeline which handles requests and responses, in an ASP.NET Core application. These middlewares are configured in the Startup.cs class. I handle several essential things in this class such as Swagger UI and MassTransit. I enable Swagger UI under the ConfigureService and Configure methods as Figure 14 and 15 shows. This part is mentioned in detail in 4.8.1. I also implement the MassTransit in the Startup.cs. Using MassTransit allows me creating a message consumer under another directory. I will explain that part while explaining consumers directory. I tried to explain how I added the MassTransit in detail in the 4.9.

```
public IConfiguration Configuration { get; }

// This method gets called by the runtime. Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    services.AddLogging(x => x.AddConsole());
    services.AddCors();
    services.AddMvc(x => x.Filters.Add<GeneralExceptionFilter>())
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
    services.AddDbContext<AppDBContext>(x => x.UseInMemoryDatabase("Test"));
    //above I configure the db context.
    //use inmemeory provider to not connect a real db to test app
    services.AddScoped<IMailRepository, MailRepository>();
    services.AddScoped<IMailService, MailService>();

    //used scoped lifetime because these calasses internally have to use
    //the db context class.it specifies the same scope.
    services.AddMassTransit(serviceCfg =>
    {
        serviceCfg.AddConsumers(typeof(Startup).Assembly);
        serviceCfg.AddBus(serviceProvider =>
        {

            IBusControl bus = Bus.Factory.CreateUsingInMemory((cfg) =>
            {
                cfg.ReceiveEndpoint("mail_scheduled_queue", e =>
                {
                    e.ConfigureConsumer<SendMailService>(serviceProvider);
                });
            });
            return bus;
        });
    });


    services.AddSwaggerGen(c => c.SwaggerDoc("v1", new Info { Title = "Email Api", Version = "v1" }));
    services.ConfigureSwaggerGen(options =>
    {
        options.DescribeAllEnumsAsStrings();
    });
```

**Figure 14:** ConfigureServices Method

```
// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app, IHostingEnvironment env, IApplicationLifetime applicationLifetime, IServiceScopeFactory serviceProvider)
{
    applicationLifetime.ApplicationStarted.Register(() => StartBus(serviceProvider));
    applicationLifetime.ApplicationStopping.Register(() => StopBus(serviceProvider));

    app.UseStaticFiles();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.
        app.UseHsts();
    }

    //app.UseHttpsRedirection();
    app.UseMvc();


    app.UseSwagger();
    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json", $"Email Api v1");
    });
```

**Figure 15:** Configure Method

### 5.3.2 Program.cs

This is the class that calls Main when the application starts. Main creates a default web host by using the Startup configuration as Figure 16 shows.

```
namespace EmailApi
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateWebHostBuilder(args).Build().Run();
        }

        public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                .UseStartup<Startup>()
            .UseKestrel()
                    .ConfigureAppConfiguration((hostingContext, config) =>
                    {
                        IHostingEnvironment env = hostingContext.HostingEnvironment;

                        string environmentName = env.EnvironmentName;
                        string basePath = Directory.GetCurrentDirectory();

                        Console.WriteLine("EnvironmentName : " + environmentName);
                        Console.WriteLine("Env Base Path : " + basePath);

                        config.SetBasePath(basePath)
                              .AddJsonFile("appsettings.json");

                        config.AddEnvironmentVariables();
                    });
```

**Figure 16:** Program Class

### 5.3.3 Controllers

**Figure 17:** HTTP POST and HTTP GET

```
[HttpPost("send-email-task")]
public IActionResult CreateSendMailTask([FromBody] CreateEmailSendTask createEmailSendTask)
{
    var createdMail = _mailService.CreateMailTask(createEmailSendTask);

    return StatusCode((int)HttpStatusCode.Created, createdMail);
}

//GET mail
[HttpGet("")]
public async Task<IEnumerable<Mail>> ListAsync()
{
    var mails = await _mailService.ListAsync();

    return mails;
}
```

A controller is the place that program handles the incoming HTTP requests and send response back to the caller. Controller class includes multiple HTTP verbs that match the methods names such as GET, POST as Figure 17 shows, PUT, DELETE. By looking the incoming request URL and HTTP verb, API decides which method and controller will handle the HTTP request.

HTTP GET requests the all mails that exists as Figure 22. In this part endpoint request a specific email by giving its id, request a specific email's status by giving its id and changing a specific email's status by giving its id as Figure 18.

```
//GET mail/id
[HttpGet("{id}")]
public IActionResult Get([FromRoute]int id)
{
    var mail = _mailService.GetMailById(id);
    if (mail == null)
    {
        return NotFound();
    }

    return Ok(mail);
}

//GET mail/id/status
[HttpGet("{mailId}/status")]
public IActionResult Put([FromRoute] int mailId)
{
    var mail = _mailService.GetMailById(mailId);

    return Ok(mail.EmailTaskStatus);
}

//PUT mail/id/status
[HttpPut("{mailId}/status")]
public IActionResult Put([FromRoute] int mailId,[FromBody] EmailTaskStatuses NewStatus)
{

    _mailService.UpdateStatus(mailId, NewStatus);

    return NoContent();
}
```

**Figure 18:** HTTP GET by id, GET by id/status, PUT

### 5.3.4 Consumers

**Figure 19:** Consume

```
public Task Consume(ConsumeContext<MailScheduledEvent> context)
{
    var mailScheduledEvent = context.Message;

    Console.WriteLine($"MailId : {mailScheduledEvent.MailId}{Environment.NewLine}" +
        $"Schedule Date : {mailScheduledEvent.ScheduleDate}");

    var a = (mailScheduledEvent.ScheduleDate - DateTime.UtcNow);


    var value = DateTime.Compare(mailScheduledEvent.ScheduleDate, DateTime.UtcNow);
```

By using MassTransit I created a message consumer which is a class that consumes messages that has one or more types. When a consumer subscribes to a recipient endpoint and a message consumed by the consumer is received by the endpoint, a consumer instance is generated. The message is then transmitted to the Consumer by the Consumption method. The consumption method is asynchronous and returns a Task as Figure 19 shows. The task is expected by MassTransit, during which time the message cannot be used to other

recipient endpoints. If the consumption method is successfully completed, the message is confirmed and removed from the queue. This is how the consumer mechanism works. After I created the consumer then, I checked the status of emails and throw exceptions in case any excepted or unexcepted situation happens while consuming the messages.

### 5.3.5 Domain

Domain layer have my application's model class which that represent my email, repositories and interfaces. Under the domain directory I have another file called Models. The only model that I have is Mail. It is a simple class that have only properties to describe Mail's basic information as Figure 20 shows.

```
namespace EmailApi.Domain.Models
{
    public class Mail
    {
        public int MailId { get; set; }
        public string SenderAddress { get; set; }
        public string Subject { get; set; }
        public string Content { get; set; }
        public DateTime Date { get; set; }
        public string ReceiverAddress { get; set; }
        public int EmailTaskStatus { get; set; }
        public DateTime ScheduledDateTime { get; set; }
    }
```

**Figure 20:** Mail Model

The Mail model has properties such as mail id, sender address, subject, content, date, receiver address, email task status and scheduled date time. Another file under the domain directory is Repositories. Inside that I have only IMailRepository. Repository is the place that used to manage the data from databases. When using the Repository Template, it helps us to define repository classes that encompass all the logic to handle data access. These repositories provide methods for listing, creating, editing, and deleting methods such as collecting, editing, and editing objects in a particular model. Internally, these methods refer to the database to perform CRUD (create, read, update, delete) operations and isolate the access of the database from the rest of the application. So I created a repository that is responsible for handling database communication as Figure 21a and 21b shows.

```
namespace EmailApi.Persistence.Repositories
{
    public class MailRepository : BaseRepository, IMailRepository
    {
        public MailRepository(AppDBContext context) : base(context)
        {

        }
        public async Task<IEnumerable<Mail>> ListAsync()
        {
            return await _context.Mails.ToListAsync();
            //tolistasync for transforming the result of the query into
            //a collection of categories

        }
        public Mail GetById(int id)
        {
            return _context.Mails.FirstOrDefault(x => x.MailId == id);
        }

        public Mail CreateMail(Mail mail)
        {
            _context.Add(mail);
            _context.SaveChanges();

            return mail;
```

**(21a)** Database Communications Repository

```
    }
    public EmailTaskStatuses GetStatus(int mailId)
    {
        var foundMail = GetById(mailId);

        return (EmailTaskStatuses)foundMail.EmailTaskStatus;
    }
    public void UpdateStatus(int mailId, EmailTaskStatuses NewStatus)
    {
        var foundMail = GetById(mailId);
        foundMail.EmailTaskStatus = (int)NewStatus;
        _context.SaveChanges();
    }
    public DateTime GetDate(int mailId)
    {
        var foundMail = GetById(mailId);
        return foundMail.Date;
    }
```

**(21b)** Database Communications Repository

**Figure 21:** Mail Repository

The Domain directory has also the IMailService interface. I will explain the services in details, but this is just the interface of the MailService that has the properties such as create mail task, get mail by id, get mail status and update status.

### 5.3.5 Services

Service is a class or interface that defines methods to handle the business logic part of the application. The main reason of using service is isolating the request and the response handling from the real logic needed to complete the tasks. My MailService has methods for create tasks as shown in Figure 22c, get a mail by id as shown in Figure 22a, get a specific mail's status and update a specific mail's status as shown in Figure 22b. All of these are predefined in the IMailService interface which is under the domain/services directory.

```
public Mail GetMailById(int id)
{

    return _mailRepository.GetById(id);
}

public EmailTaskStatuses GetMailStatus(int mailId)
{
    return _mailRepository.GetStatus(mailId);
}
```

**(22a)** GetMailById Method

```
public bool UpdateStatus(int mailId, EmailTaskStatuses NewStatus)
{

    var mail = GetMailById(mailId);

    _mailRepository.UpdateStatus(mailId, NewStatus);

    switch (NewStatus)
    {
        case EmailTaskStatuses.Pending:
            Console.WriteLine("Mail status is pending");
            break;
        case EmailTaskStatuses.Ongoing:
            Console.WriteLine("Mail status is ongoing");
            break;
        case EmailTaskStatuses.Sent:
            Console.WriteLine("Mail status is sent");
            break;
        case EmailTaskStatuses.Cancel:
            Console.WriteLine("Mail status is cancel");
            break;
    }
    return true;
```

**(22b)** UpdateStatus Method

```
public EmailTaskResponse CreateMailTask(CreateEmailSendTask createEmailSendTask)
{
    if (createEmailSendTask == null)
        throw new CustomExceptions.ValidationException("Argument could not be null");


    if (string.IsNullOrEmpty(createEmailSendTask.ReceiverAddress))
        throw new CustomExceptions.ValidationException("Receiver address should be provided");


    if (new EmailAddressAttribute().IsValid(createEmailSendTask.ReceiverAddress)==false)
    {
        throw new CustomExceptions.ValidationException("");
    }

    Mail mail = new Mail
    {
        Content = createEmailSendTask.Content,
        SenderAddress = "my-company@mail.com",
        ReceiverAddress = createEmailSendTask.ReceiverAddress,
        Subject = createEmailSendTask.Subject,
        Date = DateTime.UtcNow,
        EmailTaskStatus = (int)EmailTaskStatuses.Pending,
        ScheduledDateTime = createEmailSendTask.ScheduledDate
};

    _mailRepository.CreateMail(mail);

    MailScheduledEvent mailScheduledEvent = new MailScheduledEvent
    {
        MailId = mail.MailId,
        ScheduleDate = mail.ScheduledDateTime
    };
    _busControl.Publish(mailScheduledEvent);

    return new EmailTaskResponse()
    {
        EmailId = mail.MailId,
        EmailTaskStatuses = (EmailTaskStatuses)mail.EmailTaskStatus,
        SenderAddress = mail.SenderAddress,
        Subject = mail.Subject,
        ScheduledDateTime = mail.ScheduledDateTime,
        Date=mail.Date ///
    };
```

**(22c)** CreateMailTask Method

**Figure 22 :** MailService

My service talks to the MailRepository, to get the needed information from the database. By creating an interface and then the implanting class, I am using the dependency injection. After creating both the interface and class I bind them.

### 5.3.5 Persistence

I created the repository and the service and now I will create the database. Thus, it will complete the process; access database return requested information then return this information to the client. To do that I need a database. For the database I used Entity Framework Core as my database ORM (object-relational mapping). EF Core is built in the ASP.NET Core as defult. One of the main benefit of EF Core is that it allows programmer to design the application first and after that generate a database by thinking what we defined in my code. This is also called code first. Under the persistence directory I created a contexts file the added the AppDBContextclass as Figure 23 shows. This class needs to inherit the DbContext, which is a class that EF Core uses to map models to database tables.

```
public class AppDBContext : DbContext
{
    public DbSet<Mail> Mails { get; set; }
    public AppDBContext(DbContextOptions options) : base(options) { }

    protected override void OnModelCreating(ModelBuilder builder)
    {
        base.OnModelCreating(builder);


        builder.Entity<Mail>().ToTable("Mails");
        builder.Entity<Mail>().HasKey(p => p.MailId);
        builder.Entity<Mail>().Property(p => p.MailId).IsRequired().ValueGeneratedOnAdd();
        builder.Entity<Mail>().Property(p => p.SenderAddress).IsRequired().HasMaxLength(50);
        //builder.Entity<Mail>().Property(p => p.State).IsRequired();
        builder.Entity<Mail>().Property(p => p.Subject).IsRequired().HasMaxLength(50);
        builder.Entity<Mail>().Property(p => p.Content).IsRequired();
        builder.Entity<Mail>().Property(p => p.Date).IsRequired().ValueGeneratedOnAdd();

    }
```

**Figure 23:** AppDbContext Class

After that, I go back to my Startup class to configure the dependency injection. I configured my database context as Figure 24 shows. I tell the API to use AppDbContext with an in-memory database implementation, which is a nonrelational databse that relies on memory for the storage of data.

```
public IConfiguration Configuration { get; }

// This method gets called by the runtime. Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    services.AddLogging(x => x.AddConsole());
    services.AddCors();
    services.AddMvc(x => x.Filters.Add<GeneralExceptionFilter>())
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
    services.AddDbContext<AppDBContext>(x => x.UseInMemoryDatabase("Test"));
    //above I configure the db context.
    //use inmemeory provider to not connect a real db to test app
    services.AddScoped<IMailRepository, MailRepository>();
    services.AddScoped<IMailService, MailService>();

    //used scoped lifetime because these calasses internally have to use
    //the db context class.it specifies the same scope.
```

**Figure 24:** Dependency Injection

This concluded RESTful Email API coding process.

## CONCLUSIONS

It was my first software internship and I gained so many valuable and helpful information about this part of my future job. The people that I worked with were very experienced and helpful people. My mentor always helped when I was confused or just curious about the things. It was also pretty helpful internship to understand how agile project development works. Throughout my internsip, my team always let me involved the sprint meetings and always valued my thoughts about things that we discussed on meetings.

The company was also really caring about the interns. I was always able to find people ready to ask my questions about how the things works. The project that I completed throughout my internship was a brand new thing for me. My mentor let me make a detailed research about the topics he advised me. During the coding process my mentor and I did code reviews to make sure everything that I do is makes sense to me. Thus, it was the best internship that I worked so far.

**REFERENCES**


Eising, Perry. “What Exactly IS an API?” *Medium*, Medium, 7 Dec. 2017, https://medium.com/@perrysetgo/what-exactly-is-an-api-69f36968a41f.

“What Are Web APIs.” *By*, https://hackernoon.com/what-are-web-apis-c74053fa4072.

Work, Anshul_AggarwalBelieves in Smart, et al. “Introduction to .NET Framework.” *GeeksforGeeks*, 11 Dec. 2018, https://www.geeksforgeeks.org/introduction-to-net-framework/.

“Introduction to C#.” *GeeksforGeeks*, 12 Jan. 2019, https://www.geeksforgeeks.org/introduction-to-c-sharp/.

Octopus Deploy. “Continuous Delivery, Deployment and DevOps Platform.” *Octopus Deploy*, https://octopus.com/.

Programmer, ParthManiyarCompetitive, et al. “What Is Entity Framework in .NET Framework?” *GeeksforGeeks*, 9 Aug. 2019, https://www.geeksforgeeks.org/what-is-entity-framework-in-net-framework/.

Karann-Msft. “What Is NuGet and What Does It Do?” *What Is NuGet and What Does It Do? | Microsoft Docs*, https://docs.microsoft.com/en-us/nuget/what-is-nuget.

“API Development for Everyone.” *Swagger*, https://swagger.io/.

ReadMe. “What Is Swagger and Why It Matters.” *ReadMe Blog*, ReadMe Blog, 12 July 2016, https://blog.readme.io/what-is-swagger-and-why-it-matters/.

“How to Use Swagger UI for API Testing.” *BlazeMeter*, https://www.blazemeter.com/blog/getting-started-with-swagger-ui/.

Neel. “Enable Swagger in Your .Net Core 2.0 Application: Step by Step Guide.” *Neel Bhatt*, 11 Dec. 2017, https://neelbhatt.com/2017/12/11/enable-swagger-in-your-net-core-2-0-application-step-by-step-guide/.

“Configure MassTransit.” *Configure MassTransit · MassTransit*, https://masstransit-project.com/MassTransit/usage/configuration.html.

freeCodeCamp.org. “An Awesome Guide on How to Build RESTful APIs with ASP.NET Core.” *FreeCodeCamp.org*, FreeCodeCamp.org, 6 June 2019, https://www.freecodecamp.org/news/an-awesome-guide-on-how-to-build-restful-apis-with-asp-net-core-87b818123e28/.