ASP520

Module 6

Building a Web API for Webassembly Application

1 Server to Webassembly

1.1 Project Setup

Create a new project and solution using the following information. You will notice that the solution contains two projects; the Server project **Newrise** and the Client project **Newrise.Client**. Since the client project is referenced by the server project, anything that you want to share between the server and the client can be placed into the client project. Alternatively, you can always create a separate **Newrise.Shared** project that will be referenced by both projects. Make sure nullable feature is set to **disable** on all the projects.

Project information

```
Project name                    : Newrise
Project type                    : Blazor Web App
Location                        : <repository_folder>\src
Solution                        : Module6
Target Framework                : .NET 8.0
Authentication Type             : none
Interactive render mode         : Webassembly
Interactive location            : Global
Configure for HTTPS             : check
Do not use top-level statements : check
```

Run Visual Studio 2022 again to open Module 5. Drag **wwwroot** and **Content** folders from Module 5 to Module 6. Then confirm that you allow files with the same name to be overwritten. Module 6 will now have all the content files from Module 5.

1.2 Shared Models

Copy the **Newrise.Shared** project from Module5 to Module6 folder and then add the project file to Module6 solution file. Reference this project from both **Newrise** project and **Newrise.Client** project. It contains all the models we can use across the entire solution.

Models in Newrise Shared

```
EventType.cs
Event.cs
NewEvent.cs
Participant.cs
NewParticipant.cs
LoginInfo.cs
UserSession.cs
Office.cs
```

1.3 Server Services

All our Blazor server application services can continue to run on the server. Drag the **Services** folder from Module 5 to Module 6. Ensure that Module 6 have the following services. Some of our services require Entity Framework Core so install the following 3 NuGet packages. Check each service source file and remove any invalid or unused namespaces that you may have accidentally added.

Services in Newrise

```
OfficeListProvider.cs
NewriseDbContext.cs
ServerAuthenticationStateProvider.cs
EventDataService.cs
ParticipantDataService.cs
```

Entity Framework Core packages

```
Microsoft.EntityFrameworkCore
Microsoft.EntityFrameworkCore.SqlServer
Microsoft.EntityFrameworkCore.Tools
```

Since the UI in Webassembly project runs on the client and not the server, we need to implement a Web API to expose the services over the network. This is what we will do in the next chapter. First change the port numbers in launch settings on server project to 7000 for **https** and 5000 for **http**. Then make sure **Program** class have at least the basic startup code below. This is to register server-side services and middleware components.

Basic startup code for Newrise server project: Program.cs

```
public class Program {
    public static void Main(string[] args) {
        var builder = WebApplication.CreateBuilder(args);
        var services = builder.Services;
        services
            .AddRazorComponents()
            .AddInteractiveWebAssemblyComponents();
        var app = builder.Build();

        if (app.Environment.IsDevelopment()) {
            app.UseWebAssemblyDebugging();
        }
        else {
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseAntiforgery();
        app
            .MapRazorComponents<App>()
            .AddInteractiveWebAssemblyRenderMode()
            .AddAdditionalAssemblies(typeof(Client._Imports).Assembly);
        app.Run();
    }
}
```

1.4 Component & Pages

Our Razor components and pages should be moved over to **Newrise.Client** project because the UI runs on the client-side in a Webassembly project. However we cannot do it now because they are currently using services that only runs on the server. Once we have implemented client-side version of those services, we can move components and pages over and make some small changes. Client-side services will communicate with the server-side services through the Web API. Once all these changes are done, our application should work as usual. The following shows all the major additions that have to be implemented that was not required for Blazor server application.

Major operations on server project

```
Add a Web API to expose server-side services
Generates and uses JWT tokens for server-side authorization
```

Major operations on client project

```
Implementing client-sides that communicates with Web API
Accepts and uses JWT tokens for Web API access & client-side authorization
```

1.5 Logging

Since Web APIs are not called directly from the UI, they are difficult to debug because there are many layers from UI to Web Client, Web Client to API, API to data services, data services to database. Controllers and services are probably the components that are most like to do logging. You can easily get access to ASP.NET logging service thru constructor injection of **ILogger<T>**.

Following are the available logging levels. The logger has log methods for every level. You can set the minimum logging level in **appsettings.json** based on environment. For debugging, you are most likely to log at Debug level. Information will usually be the default logging level. You can configure different levels for the many frameworks that are utilized in a web application.

Logging Levels

```
Trace   = 0         Debug = 1       Information = 2
Warning = 3         Error = 4       Critical    = 5      None = 6
```

Setting logging levels: appSettings.json

```
"Logging": {
    "LogLevel": {
        "Default": "Information",
        "Microsoft.AspNetCore": "Warning",
        "Microsoft.EntityFrameworkCore.Database.Command": "None"
    }
}
```

2 Web API

2.1 API Controllers

ASP.NET Web API uses controllers as hubs to receive and process HTTP requests. This is like a standard MVC controller but is attached both an **ApiController** attribute and a **Route** attribute to assign a unique route to each controller. Use *[controller]* in the route to automatically map the controller name to the class. The default location of all controllers should be in a **Controllers** folder. To support Web API controllers, you need to call the **AddControllers** method to add controller services.

An example API controller class

```
[ApiController]
[Route("[controller]")]
public class MyController : ControllerBase {
        :
}
```

Add a **Controllers** folder and then add the following controllers into the folder. Routes will be hardcoded for each controller. You can use the empty API controller template to create the following classes.

Web API controller for office services: Controllers/OfficeController.cs

```
using Microsoft.AspNetCore.Mvc;
namespace Newrise.Controllers {
    [ApiController, Route("api/offices")]
    public class OfficeController : ControllerBase {
    }
}
```

Web API controller for event services: Controllers/EventController.cs

```
using Microsoft.AspNetCore.Mvc;
namespace Newrise.Controllers {
    [ApiController, Route("api/events")]
    public class EventController : ControllerBase {
    }
}
```

Web API controller for participant services: Controllers/ParticipantController.cs

```
using Microsoft.AspNetCore.Mvc;
namespace Newrise.Controllers {
    [ApiController, Route("api/participants")]
    public class ParticipantController : ControllerBase {
    }
}
```

Add support for controllers

```
services.AddControllers();
```

Mapping routes to ApiControllers

```
app.MapControllers();
```

Let us now use **OfficeController** to expose **OfficeListProvider** service. We will first register the required services; **IMemoryCache** and **OfficeListProvider**. Constructor injection can be used by controllers to get access to services.

Register required services: Program.cs

```
services.AddMemoryCache();
services.AddSingleton<OfficeListProvider>();
```

Get access to services through constructor injection: OfficeController.cs

```
readonly OfficeListProvider _officeListProvider;

public OfficeController(OfficeListProvider officeListProvider) {
    _officeListProvider = officeListProvider;
}
```

All public methods in a controller are API action methods by default and accessible by its route with a HTTP GET/PUT/POST/DELETE request. If the action method does not have its own route, the controller route will be used to trigger the action. Run the application and enter the following URL to test the action. You should get a JSON response containing the office list.

Controller action method

```
[HttpGet]
public List<Office> GetList() { return _officeListProvider.GetList(); }
```

URL route that maps to controller action

```
https://localhost:7000/api/offices
```

2.2 Handling Exceptions

When a successful operation guarantee a **List<Office>** response, then what would a failed operation return? Can we return **null**? Yes but returning a null response could simply trigger the next middleware to process the request or a 404 Not Found error. Plus the client and the user will not know why the operation failed. Responses should be standardized so clients know exactly whether an operation is successful or not and be able to access the results on success and process the error on failure.

You can implement a data model specifically to be returned as the response for every API request. This model can have a boolean **Success** property that a client can check to see if the operation is successful or not. If it is successful, the result is accessible from the **Value** property. If not you can then access the error message from a **Error** property.

A standard generic API response model class: ApiResult.cs

```
public class ApiResult<T> {
    public bool Success { get; set; }
    public string Error { get; set; }
    public T Value { get; set; }
    public ApiResult() { }  // used by JSON deserialization
    public ApiResult(string error) { Error = error; }
    public ApiResult(T value) { Value = value; Success = true; }
}
```

A better response model from API action

```
public ApiResult<List<Office>> GetList() {
    try {
        var offices = _officeListProvider.GetList();
        return new ApiResult<List<Office>>(offices);
    }
    catch (Exception ex) {
        return new ApiResult<List<Office>>(
            $"Unable to return office list. {ex.Message}");
    }
}
```

2.3 API Testing

Build a client for a Web API can consume a lot of time and sometimes a developer is only responsible to construct the Web API but not for building clients. In many cases, clients are suppose to build by third-party developers or companies to interface to our Web API. However we still need to test our Web API and a web browser by default can only issue HTTP GET requests and unable to test actions that requires authentication. In this case, we suggest using a third-party API test application like **Postman**. Start **Postman** and login with your registered account. Add a new **Newrise** collection to store requests for Newrise Web API. Add a folder named **Offices** to group requests for offices API. Then add the following request to the folder.

Test request to add

```
Get List    GET https://localhost:7000/api/offices
```

Note that you might be using repeated content across multiple requests. You should create a variable for the content so that you can easily change the content later. Add in a **url** variable with the value is **https://localhost:7000/api** for the **Newrise** collection.Then replace the base part of the URL in all the requests with the **{{url}}** variable. It will make it easier if the URL changes in the future. For example we may wish to perform the same tests on production server and not only the development server.

2.4 API Documentation

It will be impossible for anyone to build a client for a Web API without having proper documentation. Instead of manually producing the documentation, you can make use of **Swagger** to dynamically generate online documentation directly from a Web API. It follows the Open API standard so the generated documentation can be easily use to build clients regardless of programming language and platform. Swagger generates a documentation UI that can also be used as a client to the Web API so you can test the API without using Postman. To use **Swagger** download the following package.

Package for ASP.NET Core Swagger support

```
Swashbuckle.AspNetCore
```

Call **AddSwaggerGen** for documentation generation services in **ConfigureServices** method. Call **UseSwagger** and **UseSwaggerUI** to register the required middleware to produce the documentation JSON file on request and generate the online UI for you to view and test the Web API. If you are not developing a public Web API, make sure to call these methods only in the Development environment. It is possible to use tools to automatically generate an API client from the Open API documentation. Search for **Swagger CodeGen** to learn more about this.

Configure documentation generation services

```
services.AddSwaggerGen(options => {
    options.SwaggerDoc("v1", new OpenApiInfo {
        Title = "Newrise API",
        Version = "v1"
    });
});
```

Add Swagger middleware

```
app.UseSwagger();
```

Configure endpoint to expose Swagger UI

```
app.UseSwaggerUI(options =>
    options.SwaggerEndpoint(
        "/swagger/v1/swagger.json",
        "Newrise API"));
```

Examine and test your Web API through Swagger UI

```
https://localhost:7000/swagger/index.html
```

To get the Open API documentation

```
https://localhost:7000/swagger/v1/swagger.json
```

2.5 Event Web API

EventDataService was initially implemented for a Blazor server application. Below is a list of data access methods available from this service. No changes are required for accessing **EventDataService** methods from a Web API.

EventDataService methods

```
public async Task<List<Event>> GetEventsAsync();
public async Task<Event> GetEventAsync(string id);
public async Task AddEventAsync(Event item);
public async Task RemoveEventAsync(string id);
public async Task UpdateEventAsync(Event item);
public async Task<bool> HasParticipantAsync(string eventId, string participantId);
public async Task AddParticipantAsync(string eventId, string participantId);
public async Task RemoveParticipantAsync(string eventId, string participantId);
```

You will have to work out the route required to activate each method and from where you can access the input parameters. Even when the route is the same, you can use a different HTTP method to activate a different action. The HTTP method can determine how a web client can pass data to the web API.

API HTTP methods and routes

Get events	GET	events	
Get event	GET	events/{id}	
Add event	PUT	events	*(event object in HTTP body)*
Update event	POST	events	*(event object in HTTP body)*
Remove event	DELETE	events/{id}	
Check event has participant	GET	events/{eventId}/participants/{participantId}	
Add participant to event	PUT	events/{eventId}/participants/{participantId}	
Remove participant from event	DELETE	events/{eventId}/participants/{participantId}	

Simple short strings or numerical data can be embedded into a route. You can use **{}** to identity which parts of a route represent data. The **id**, **eventId** and **participantId** can becomes input parameters to action methods. Parameters can also be passed as part of a query string (**?**) that is attached to the end of the route rather than within a route.

Using route to pass parameters

```
GET https://localhost:7000/CSC100
GET https://localhost:7000/events/CSC100/participants/xnamp
```

Using query string to pass parameters

```
GET https://localhost:7000/events?id=CSC100
GET https://localhost:7000/events/participants?eventId=CSC100&participantId=xnamp
```

Since the length of a URL is limited you should not pass a large volume of data using the route or query string. A HTTP request has head and body sections. An object or a large volume of data can be passed in the HTTP body section. However only **PUT** and **POST** HTTP methods can have a body section. You can use **FromQuery** attribute to identify an input parameter from a query string and **FromBody** attribute to identity data stored in the HTTP body section.

Make sure to register **EventDataService** so it can be accessed by the controller. You can register it as a singleton as the service does not require to maintain any state per request. Make sure application settings contain the database connection string. Add a database context factory for **NewriseDbContext** and register all data services.

Setup for data services: Program.cs

```
var connectionString = builder.Configuration.GetConnectionString("NewriseDb");
    services.AddDbContextFactory<NewriseDbContext>(options =>
    options.UseSqlServer(connectionString));
services.AddSingleton<EventDataService>();
```

Controller for events: Controllers\EventController.cs

```
[ApiController]
[Route("api/events")]
public class EventController : ControllerBase {
    private readonly EventDataService _eventDataService;
    private readonly ILogger<EventController> _logger;

    public EventController(
        EventDataService eventDataService,
        ILogger<EventController> logger) {
        _eventDataService = eventDataService;
        _logger = logger;
    }
}
```

API controller actions can be synchronous or asynchronous but we will implement all our action methods to be the later. Action methods that do not have a specific return value will return back the input data. For example, **AddEvent** returns back the **Event** added, and **AddParticipant** returns back the **Participant** added. If there is nothing to return, just return a **bool** value of true. Following is a list of all the action methods exposed from our Web API.

Returning a list of events

```
[HttpGet]
public async Task<ApiResult<List<Event>>> GetEvents() {
    try {
        var value = await _eventDataService.GetEventsAsync();
        return new ApiResult<List<Event>>(value);
    }   catch (Exception ex) { return new ApiResult<List<Event>>(ex.Message); }
}
```

Returning a single event

```
[HttpGet, Route("{id}")]
public async Task<ApiResult<Event>> GetEvent(string id) {
    try {
        var value = await _eventDataService.GetEventAsync(id);
        return new ApiResult<Event>(value);
    }
    catch (Exception ex) {
        return new ApiResult<Event>(ex.Message);
    }
}
```

Adding a new event

```
[HttpPut]
public async Task<ApiResult<Event>> AddEvent([FromBody] Event item) {
    try {
        await _eventDataService.AddEventAsync(item);
        var value = await _eventDataService.GetEventAsync(item.Id);
        _logger.LogInformation($"Event {item.Id} added.");
        return new ApiResult<Event>(value);
    }
    catch (Exception ex) {
        return new ApiResult<Event>(ex.Message);
    }
}
```

Updating an existing event

```
[HttpPost]
public async Task<ApiResult<Event>> UpdateEvent([FromBody] Event item) {
    try {
        await _eventDataService.UpdateEventAsync(item);
        var value = await _eventDataService.GetEventAsync(item.Id);
        _logger.LogInformation($"Event {item.Id} updated.");
        return new ApiResult<Event>(value);
    }
    catch (Exception ex) {
        return new ApiResult<Event>(ex.Message);
    }
}
```

Removing an existing event

```
[HttpDelete, Route("{id}")]
public async Task<ApiResult<Event>> RemoveEvent(string id) {
    try {
        var value = await _eventDataService.GetEventAsync(id);
        await _eventDataService.RemoveEventAsync(id);
        _logger.LogInformation($"Event {id} removed.");
        return new ApiResult<Event>(value);
    }
    catch (Exception ex) {
        return new ApiResult<Event>(ex.Message);
    }
}
```

Checking if an event has a specific participant

```
[HttpGet, Route("{eventId}/participants/{participantId}")]
public async Task<ApiResult<bool>> CheckEventHasParticipant(
    string eventId, string participantId) {
    try {
        var value = await _eventDataService.HasParticipantAsync(eventId, participantId);
        return new ApiResult<bool>(value);
    }
    catch (Exception ex) {
        return new ApiResult<bool>(ex.Message);
    }
}
```

Adding a participant to an event

```
[HttpPut, Route("{eventId}/participants/{participantId}")]
public async Task<ApiResult<bool>> AddParticipantToEvent(
    string eventId, string participantId) {
    try {
        await _eventDataService.AddParticipantAsync(eventId, participantId);
        _logger.LogInformation($"Participant {participantId} added to event {eventId}.");
        return new ApiResult<bool>(true);
    }
    catch (Exception ex) {
        return new ApiResult<bool>(ex.Message);
    }
}
```

Removing a participant from an event

```
[HttpDelete, Route("{eventId}/participants/{participantId}")]
public async Task<ApiResult<bool>> RemoveParticipantFromEvent(
    string eventId, string participantId) {
    try {
        await _eventDataService.RemoveParticipantAsync(eventId, participantId);
        _logger.LogInformation($"Participant {participantId} removed from event {eventId}.");
        return new ApiResult<bool>(true);
    }
    catch (Exception ex) {
        return new ApiResult<bool>(ex.Message);
    }
}
```

Methods that uses the same route still can be differentiated when not using the same HTTP request method. A **GET** request is normally used to fetch data, a **PUT** request to add new data, **POST** request to update data, and **DELETE** request to remove data.

2.6 Participant Web API

While **EventDataService** was not modified in any way for a Web API this is not true for **ParticipantDataService**. **AuthenticationServiceProvider** service is only used for Blazor. However if your Blazor application is no longer running on the server then there is no need for it. Web API uses a different system altogether for authentication and authorization. We need to change or remove methods that uses this service.

ParticipantDataService methods

```
public async Task InitializeAsync();
public async Task SignInAsync(string userId, string password);         // change
public async Task AddParticipantAsync(NewParticipant participant);
public async Task UpdatePhotoAsync(string id, byte[] photo);
public async Task<Participant> GetParticipantAsync(string id);
public async Task<Participant> GetParticipantWithEventsAsync(string id);
public async Task<Participant> GetCurrentParticipantAsync();           // remove
public async Task<Participant> GetCurrentParticipantWithEventsAsync(); // remove
public async Task SignOutAsync();                                      // remove
```

The **SignInAsync** method no longer uses **AuthenticationStateProvider**. It simply returns a valid **UserSession** object after a successful login. There are still more work that has to be done to support Web API authorization but this is not something a data service has to deal with.

Remove usage of AuthenticationStateProvider

```
public async Task<UserSession> SignInAsync(string userId, string password) {
    var hashedPassword = HashPassword(password);
    using (var dc = await _dcFactory.CreateDbContextAsync()) {
        var user = dc.Participants.FirstOrDefault(p =>
            (p.Id == userId || p.Email == userId) && p.PasswordHash == hashedPassword);
        if (user == null) throw new Exception("Invalid user ID or password");
        var userSession = new UserSession {
            UserId = user.Id,
            IsAdmin = user.IsAdmin
        };
        return userSession;
    }
}
```

A Web API does not need to maintain the login state and thus has no need to provide a logout option. So **SignOutAsync** can be removed. Only the Blazor application that runs on the web browser would need to maintain the login state. Web API can use the **HttpContext.User** property for identity and authorization whenever required and not the data service so remove both the **GetCurrentParticipantAsync** method and also the **GetCurrentParticipantWithEventsAsync** method. Finally remove the injection and field for **AuthenticationStateProvider**. Following is the new constructor and the services we will be injecting.

Constructor injection of services for ParticipantDataService

```
public class ParticipantDataService {
    readonly IDbContextFactory<NewriseDbContext> _dcFactory;
    readonly ILogger<ParticipantDataService> _logger;

    public ParticipantDataService(
        IDbContextFactory<NewriseDbContext> dcFactory,
        ILogger<ParticipantDataService> logger) {
        _dcFactory = dcFactory;
        _logger = logger;
    }
                :
}
```

Since **ServerAuthenticationStateProvider** is no longer used on the server, you can now safely remove it from **Newrise** project. Register the data service as a singleton as it no longer need to maintain any state per user or request.

Setup for data services: Program.cs

```
var connectionString = builder.Configuration.GetConnectionString("NewriseDb");
    services.AddDbContextFactory<NewriseDbContext>(options =>
    options.UseSqlServer(connectionString));
services.AddSingleton<EventDataService>();
services.AddSingleton<ParticipantDataService>();
```

Controller for participants: Controllers\ParticipantController.cs

```
[ApiController]
[Route("api/participants")]
public class ParticipantController : ControllerBase {
    private readonly ParticipantDataService _participantDataService;
    private readonly ILogger<ParticipantController> _logger;

    public ParticipantController(
        ParticipantDataService participantDataService,
        ILogger<ParticipantController> logger) {
        _participantDataService = participantDataService;
        _logger = logger;
    }
}
```

API HTTP methods and routes for participant

Add participant	PUT	participants
Participant sign-in	POST	participants/login
Get participant	GET	participants/{id}
Get events for participant	GET	participants/{id}/events
Update participant photo	POST	participants/{id}/photo

Adding a new participant

```
[HttpPut]
public async Task<ApiResult<Participant>> AddParticipant(
    [FromBody] NewParticipant participant) {
    try {
        await _participantDataService.AddParticipantAsync(participant);
        _logger.LogInformation($"Participant {participant.Id} has been added.");
        var value = await _participantDataService.GetParticipantAsync(participant.Id);
        value.PasswordHash = null; return new ApiResult<Participant>(value);
    }
    catch (Exception ex) {
        return new ApiResult<Participant>(ex.Message);
    }
}
```

Note that the above method clears **PasswordHash** before returning the data back to the client but understand that it is impossible for them to decipher the password from the hash if you are using a strong password and the password salt is a secret. Even so it is always good not to be lazy and complacent about security concerns.

Participant login

```
[HttpPost, Route("login")]
public async Task<ApiResult<UserSession>> Login([FromBody] LoginInfo login) {
    try {
        var value = await _participantDataService.SignInAsync(login.UserID, login.Password);
        _logger.LogInformation($"Participant {value.UserId} has logged-in.");
        return new ApiResult<UserSession>(value);
    }
    catch (Exception ex) {
        return new ApiResult<UserSession>(ex.Message);
    }
}
```

Retrieving a participant

```
[HttpGet, Route("{id}")]
public async Task<ApiResult<Participant>> GetParticipant(string id) {
    try {
        var value = await _participantDataService.GetParticipantAsync(id);
        value.PasswordHash = null; return new ApiResult<Participant>(value);
    }
    catch (Exception ex) {
        return new ApiResult<Participant>(ex.Message);
    }
}
```

Even though the **GetParticipantWithEventsAsync** service method returns both the participant and it's related events, the API method returns the related events but not the participant since we expect that the client already retrieved a participant before it tried to access the events.

Retrieving events for a participant

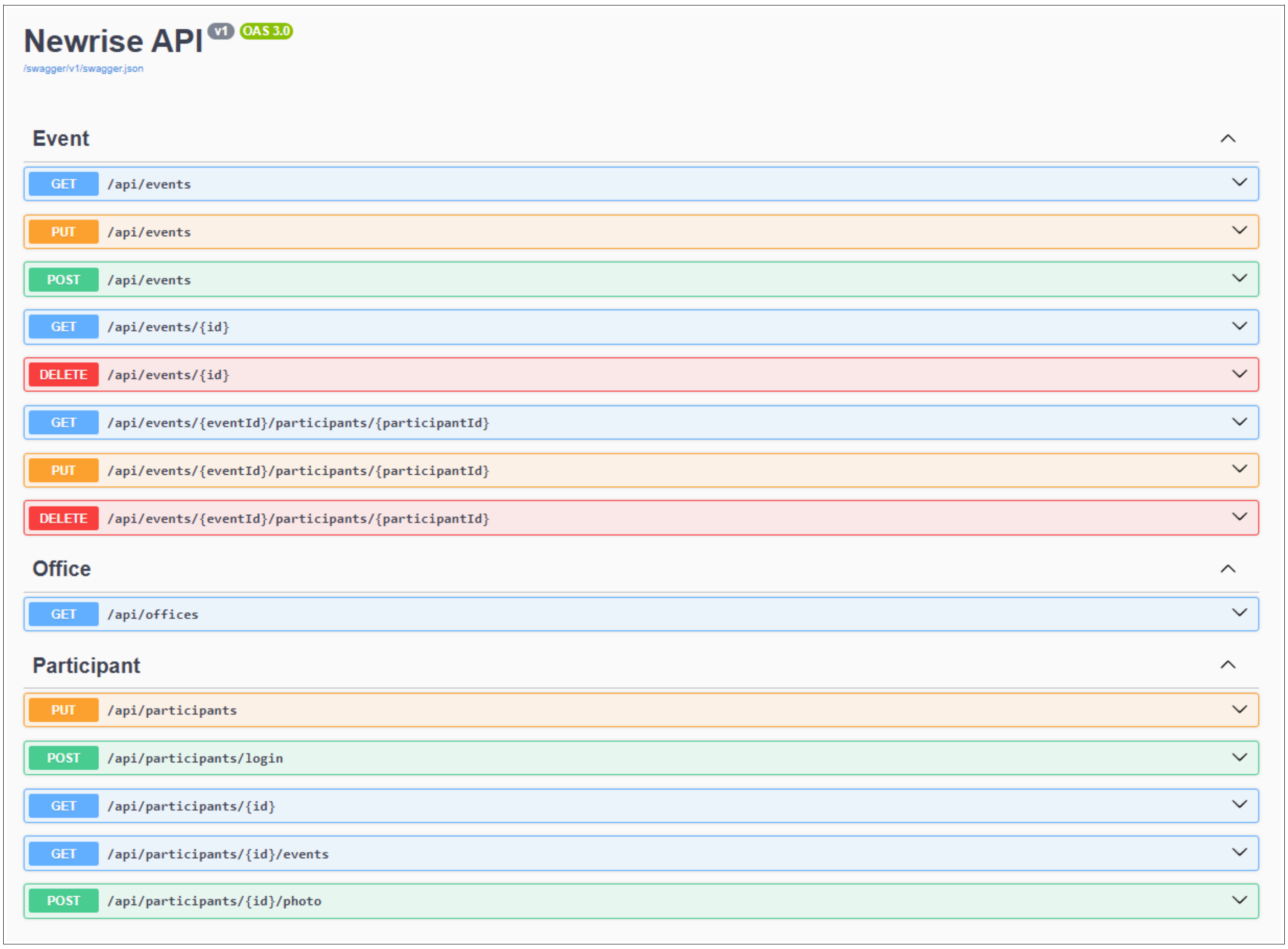
```
[HttpGet, Route("{id}/events")]
public async Task<ApiResult<List<Event>>> GetEventsForParticipant(string id) {
    try {
        var participant = await _participantDataService.GetParticipantWithEventsAsync(id);
        var value = participant.Events.ToList(); // return events only, not the participant
        return new ApiResult<List<Event>>(value);
    }
    catch (Exception ex) {
        return new ApiResult<List<Event>>(ex.Message);
    }
}
```

Updating participant photo

```
[HttpPost, Route("{id}/photo")]
public async Task<ApiResult<bool>> UpdateParticipantPhoto(string id, [FromBody] byte[] photo) {
    try {
        await _participantDataService.UpdatePhotoAsync(id, photo);
        _logger.LogInformation($"Participant {id} updated their photo.");
        return new ApiResult<bool>(true);
    }
    catch (Exception ex) {
        return new ApiResult<bool>(ex.Message);
    }
}
```

Currently this Web API is not secured. Anyone can access it without having to log-in. No authentication or authorization has been established. Still all the methods can be tested at this time by using Postman or Swagger during development but not safe for production. In the following chapter, we will secure our Web API by implementing JWT Authentication.

The Newrise Web API



Newrise API v1 OAS 3.0

/swagger/v1/swagger.json

Event

GET /api/events

PUT /api/events

POST /api/events

GET /api/events/{id}

DELETE /api/events/{id}

GET /api/events/{eventId}/participants/{participantId}

PUT /api/events/{eventId}/participants/{participantId}

DELETE /api/events/{eventId}/participants/{participantId}

Office

GET /api/offices

Participant

PUT /api/participants

POST /api/participants/login

GET /api/participants/{id}

GET /api/participants/{id}/events

POST /api/participants/{id}/photo

3 JWT Authentication

3.1 Jason Web Token

Authentication is implemented differently for Web API. A client will use the API to sign in and a JWT (Jason Web Token) will be returned back to the client together with user information neccessary for authorization on client-side. The token will be send back to the server for each API request for authorization on the server-side. We need to install following packages to generate the JWT token and to perform authentication and authorization from the the API request token. We need to update **UserSession** model to include token and token expiry details.

Packages to install to support JWT authentication

```
System.IdentityModel.JasonWebTokens
Microsoft.AspNetCore.Authentication.JwtBearer
```

Update UserSession to support token

```
namespace Newrise.Shared.Models {
    public class UserSession {
        public string UserId { get; set; }
        public bool IsAdmin { get; set; }
        public string Token { get; set; }
        public TimeSpan ExpiresIn { get; set; }
        public DateTime ExpiresAt { get; set; }
                 :
    }
}
```

We need to write code to generate the JWT token. It is not necessary to implement a separate service just to generate a token so we will add a static **SecurityExtensions** class in **Services** to expose information and operations related to security. We declare a **JWT_SECURITY_KEY** field to store a 32-character secret key used for encrypting the token. We also add a **JWT_TOKEN_EXPIRES** field to assign a default expiry time on the token. You can judge how long your users will stay on your site after the initial login. If the token is expired, the user have to login again to obtain a new token. You can also decide to automatically refresh the token if it has not expired for a long time so user will not be forced to login again.

Token-related fields: Services\SecurityExtensions.cs

```
public static class SecurityExtensions {
    public const string JWT_SECURITY_KEY = "mXu4yaf7oDGF5nRCXnrB1KQGhFTDlSax";
    public static readonly TimeSpan JWT_TOKEN_EXPIRES = TimeSpan.FromHours(8);
}
```

Site to generate random passwords and keys

```
https://randomkeygen.com/
```

We will now add an extension method for **UserSession** to generate a token and then store it together with expiry details in the object. If you do not pass in an expiry time, we will use the default expiry time in **JWT_TOKEN_EXPIRES**. Following shows the operations we will perform to get the token.

Extension method to generate JWT token

```
public static void GenerateToken(this UserSession session, TimeSpan? expiresIn = null) {
            :
}
```

Generate claims identity

```
var principal = session.GetPrincipal();
```

We then use the secret security key and the security algorithm to generate an object called as **SigningCredentials** to sign the token. The security key is symmetric which means it can be used for both encryption and decryption. Anyone having the key and knows what is the security algorithm will be able to decrypt the token.

Generate signing credentials

```
var tokenKey = Encoding.ASCII.GetBytes(JWT_SECURITY_KEY);
var signingCredentials = new SigningCredentials(
    new SymmetricSecurityKey(tokenKey),
    SecurityAlgorithms.HmacSha256Signature);
```

Use **SecurityTokenDescriptor** to contain the subject, absolute expiry date and time, and signing credentials. Then create a **SecurityTokenHandler** that is responsible for serializing, validating, and deserializing security tokens. Call **CreateToken** to create the security token and **WriteToken** to convert the token into a string. The descriptor can contain additional information such as **Issuer** and **Audience**.

Construct the security descriptor

```
if (expiresIn == null) expiresIn = JWT_TOKEN_EXPIRES;
var expiresAt = DateTime.UtcNow.Add(expiresIn.Value);
var securityDescriptor = new SecurityTokenDescriptor {
    Subject = claimsIdentity,
    Expires = expiresAt,
    SigningCredentials = signingCredentials
};
```

Creating the token and converting the token into a string

```
var securityTokenHandler = new JwtSecurityTokenHandler();
var securityToken = securityTokenHandler.CreateToken(securityDescriptor);
```

Store all token details into UserSession

```
session.Token = securityTokenHandler.WriteToken(securityToken);
session.ExpiresIn = expiresIn.Value;
session.ExpiresAt = expiresAt;
```

You can now update your sign-in method to automatically generate the token stored into the **UserSession** object. The token will be sent along with the object back to the client.

Generate the JWT token on login: ParticipantController.cs

```
[HttpPost, Route("login")]
public async Task<ApiResult<UserSession>> Login([FromBody] LoginInfo login) {
    try {
        var value = await _participantDataService.SignInAsync(login.UserID, login.Password);
        _logger.LogInformation($"Participant {value.UserId} has logged-in.");
        value.GenerateToken(); return new ApiResult<UserSession>(value);
    }
    catch (Exception ex) {
        return new ApiResult<UserSession>(ex.Message);
    }
}
```

To verify that the token is valid, go to any of the following sites and enter the JWT token, select the signing algorithm and your non-Base64 signing key. You should be able to see the claims and **iat** (issued at), **nbf** (not usable before), and **exp** (expires at) timestamps after the token is decrypted and checking whether the signature is correct.

Sites to encode/decode JWT token

```
https://jwt.io
https://token.dev
```

Online conversion between timestamp and human time

```
https://www.epochconverter.com/
```

3.2 ASP.NET Authentication

We will now configure JWT authentication and authorization on the server-side. First call **AddAuthentication** method to use bearer authentication and then add the JWT Bearer authentication, configure the authentication and token validation parameters.

Setup JWT Bearer authentication and configure token validation: Program.cs

```
services
    .AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(JwtBearerDefaults.AuthenticationScheme, options => {
        options.RequireHttpsMetadata = false;
        options.SaveToken = true;
        options.TokenValidationParameters = new() {
        ValidateIssuerSigningKey = true,
        IssuerSigningKey = new SymmetricSecurityKey(
            Encoding.ASCII.GetBytes(SecurityExtensions.JWT_SECURITY_KEY)),
            ValidateAudience = false,
            ValidateIssuer = false
        };
    });
```

If security descriptor contains **Issuer and Audience**, you can validate them against **ValidIssuer** or **ValidIssuers**, and **ValidAudience** or **ValidAudiences**. Following is an example of doing this. If **RequireHttpsMetadata** is **true**, then addresses must be **HTTPS** and not **HTTP**. Make sure to enable ASP.NET authentication and authorization middleware with **UseAuthentication** and **UseAuthorization**.

Issuer and Audience in security descriptor

```
securityDescriptor.Issuer = "https://newrise.com.my"
securityDescriptor.Audience = "https://www.newrise.com.my"
```

Issuer and Audience in validation parameters

```
validationParameters.ValidateIssuer = true;
validationParameters.ValidateAudience = true;
validationParameters.ValidIssuer = "https://newrise.com.my";
validationParameters.ValidAudience = "https://www.newrise.com.my";
```

Make sure to enable middleware for authorization and authentication

```
app.UseAuthentication();
app.UseAuthorization();
```

3.3 ASP.NET Authorization

You can now mark controllers or individual action methods with **Authorise** attribute. If most of the methods in the controller require authorization, mark the class instead. For certain methods that do not require authorization, attach the **AllowAnonymous** attribute.

Event API Controller requires authorization: Controllers\EventController.cs

```
[ApiController]
[Authorize, Route("api/events")]
public class EventController : ControllerBase {
        :
}
```

GetEvents does not require authorization

```
[HttpGet]
[AllowAnonymous]
public async Task<ApiResult<List<Event>>> GetEvents() {
        :
}
```

Adding an event require admin role

```
[HttpPut, Authorize(Roles = "admin")]
public async Task<ApiResult<Event>> AddEvent([FromBody] Event item) {
        :
}
```

Updating and removing event should also require admin role

```
[HttpPost, Authorize(Roles = "admin")]
public async Task<ApiResult<Event>> UpdateEvent([FromBody] Event item) {
            :
}

[HttpDelete, Route("{id}"), Authorize(Roles = "admin")]
public async Task<ApiResult<Event>> RemoveEvent(string id) {
            :
}
```

Participant API Controller requires authorization: Controllers\ParticipantController.cs

```
[ApiController]
[Authorize, Route("api/participants")]
public class ParticipantController : ControllerBase {
        :
}
```

Participant registration does not require authorization

```
[HttpPut, AllowAnonymous]
public async Task<ApiResult<Participant>> AddParticipant(
    [FromBody] NewParticipant participant) {
        :
}
```

Login does not require authorization

```
[HttpPost, Route("login"), AllowAnonymous]
public async Task<ApiResult<UserSession>> Login([FromBody] LoginInfo login) {
        :
}
```

3.4 Programmatic Authorization

Sometimes we need to write code for complex authorization requirements. Controllers have access to **HttpContext.User** object using the **User** property. This is actually the **ClaimsPrincipal** created by the authentication process. You can access **Identity** or check roles by calling the **IsInRole** method. There are a few operations that can only be done by either the administrator or only allowed if the operation affects their own account.

Method that uses ClaimsPrincipal

```
private void EnsureAdminOrSelf(string userId) {
    if (!User.IsInRole("admin") && User.Identity.Name != userId)
        throw new Exception("You are not allowed to perform this operation.");
}
```

For example, if you are not admin you cannot get other Participant's details, events or update their photo. If you are not admin, you should only be able to add yourself to an event or remove yourself from the event. Add the above method to the controllers and then decide which method needs to call it.

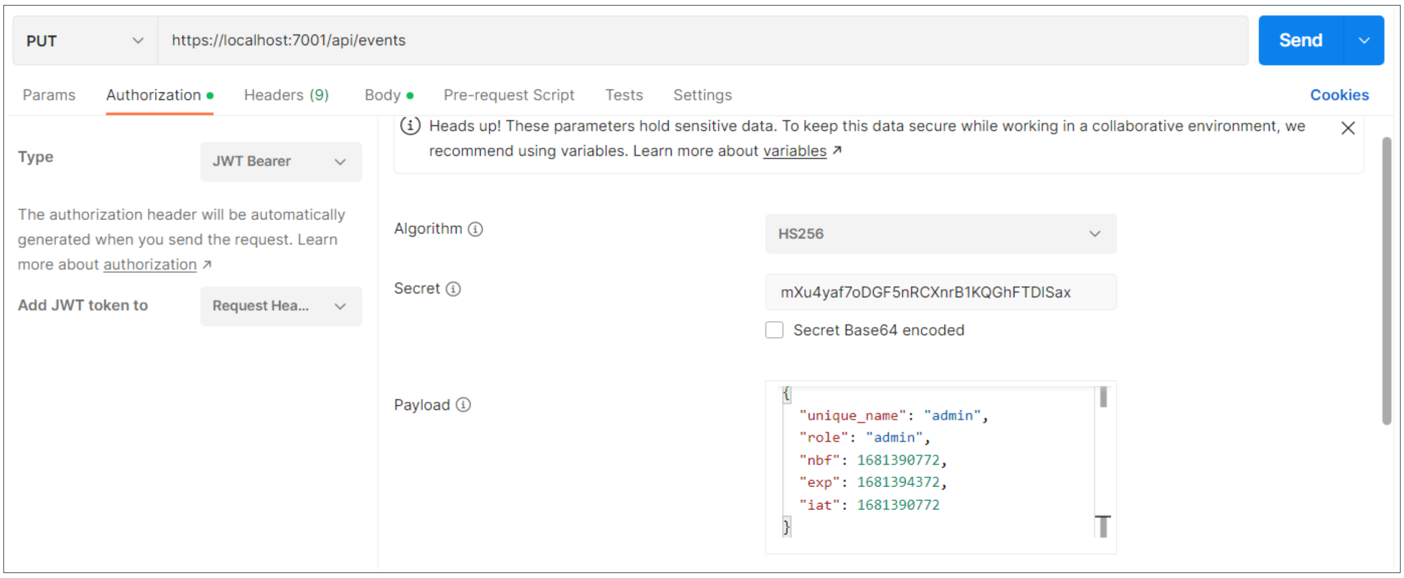
3.5 Testing with Postman

If you want to test such an authorized method, login as admin to get the token. Then goto https://token.dev to decrypt the token. Copy the payload. In Postman, create a PUT request and select Authorization and provide the following information. Change the expiry timestamp in the payload so the JWT token generated by Postman would not expire for some time. You can set this up per request or for an entire collection.

JWT Authorization settings

```
Authentication Type : JWT Bearer
Add JWT Token to    : Request Header
Algorithm           : HS256
Secret Key          : e.g. mXu4yaf7oDGF5nRCXnrB1KQGhFTDlSax
Payload             : e.g. {
                         "unique_name": "admin",
                         "role": "admin",
                         "nbf": 1681390772,
                         "exp": 1714521599,
                         "iat": 1681390772
                      }
```

Postman authorization settings


PUT https://localhost:7001/api/events

Params | Authorization | Headers (9) | Body | Pre-request Script | Tests | Settings | Cookies

Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. Learn more about variables ↗

Type: JWT Bearer

The authorization header will be automatically generated when you send the request. Learn more about authorization ↗

Add JWT token to: Request Hea...

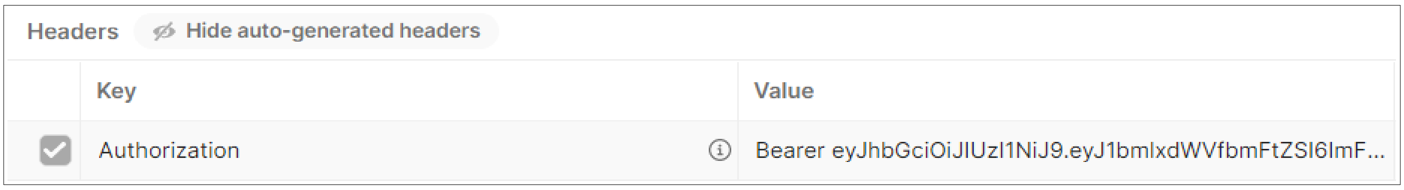
Algorithm: HS256

Secret: mXu4yaf7oDGF5nRCXnrB1KQGhFTDlSax

Secret Base64 encoded

Payload:
```
{
  "unique_name": "admin",
  "role": "admin",
  "nbf": 1681390772,
  "exp": 1681394372,
  "iat": 1681390772
}
```

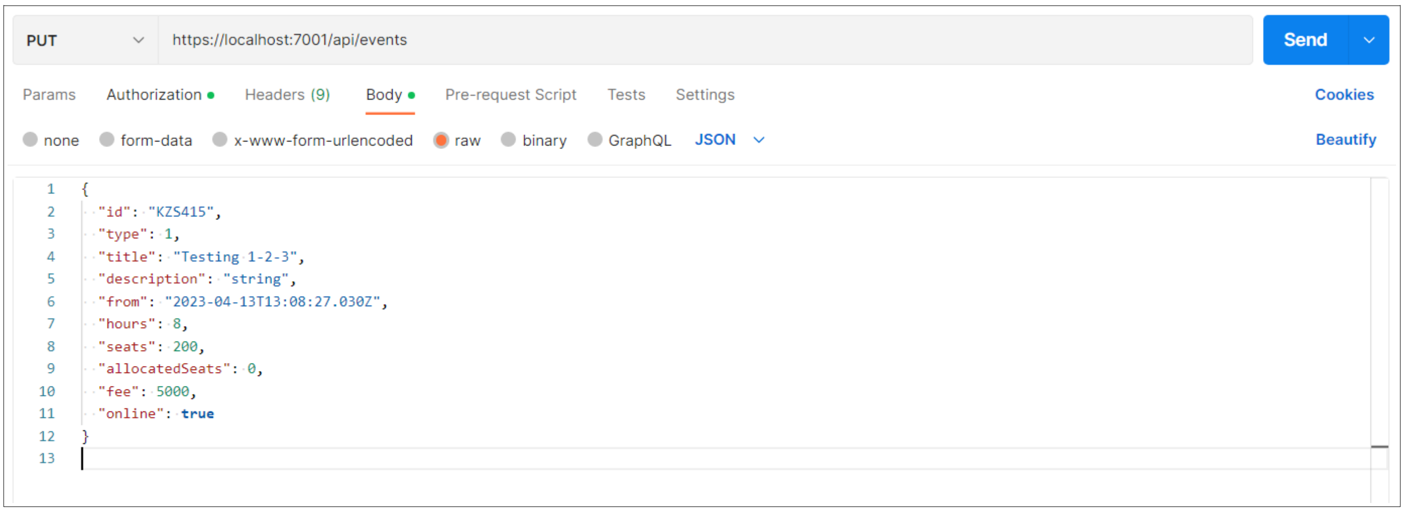
Authorization header with generated JWT Token added by Postman


Headers Hide auto-generated headers

	Key	Value
✓	Authorization	Bearer eyJhbGciOiJIUzI1NiJ9.eyJ1bmlxdWVfbmFtZSI6ImF...

To demonstrate adding an event, you also need a sample of a new **Event** to send. You can use Swagger to generate the sample for you. In Postman, select Body, set to raw and select JSON instead of Text and paste in the sample. Modify sample data and remove the fields that are not required. You can now try to submit the request.

Sample JSON-formatted Event in request body



https://localhost:7001/api/events

```json
{
  "id": "KZS415",
  "type": 1,
  "title": "Testing 1-2-3",
  "description": "string",
  "from": "2023-04-13T13:08:27.030Z",
  "hours": 8,
  "seats": 200,
  "allocatedSeats": 0,
  "fee": 5000,
  "online": true
}
```

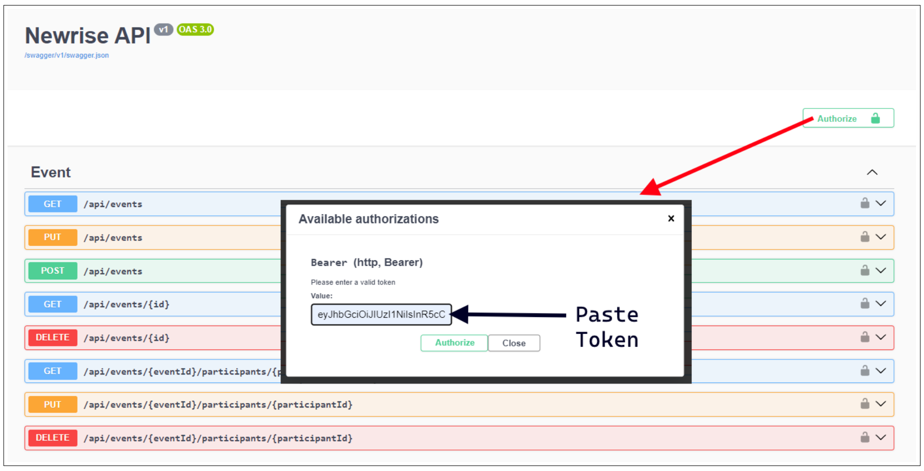
3.6 Testing with Swagger

To use JWT with Swagger UI for testing add additional option configuration when you call **AddSwaggerGen** method. There will be an **Authorize** button in the Swagger UI where you can click on the button and paste in your JWT token. You can test methods that require authorization to work.

Enabled JWT support in Swagger: Program.cs

```
services.AddSwaggerGen(options => {
            :
    options.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme {
        In = ParameterLocation.Header, Description = "Please enter a valid token",
        Name = "Authorization", Type = SecuritySchemeType.Http,
        BearerFormat = "JWT", Scheme = "Bearer"});
    options.AddSecurityRequirement(new OpenApiSecurityRequirement {{
        new OpenApiSecurityScheme {
            Reference = new OpenApiReference {
                Type = ReferenceType.SecurityScheme, Id="Bearer" }},
        new string[]{}}});
});
```

Swagger authorize button and dialog



GET /api/events
PUT /api/events
POST /api/events
GET /api/events/{id}
DELETE /api/events/{id}
GET /api/events/{eventId}/participants/{p
PUT /api/events/{eventId}/participants/{participantId}
DELETE /api/events/{eventId}/participants/{participantId}