Trabalho Prático 2

Henrique Ademar Ferreira dos Santos Rodrigues Sarmento

Jorge Miguel Gomes Rocha

Nº 16959– Regime Pós-laboral

Nº 17026 – Regime Pós-laboral

Docente

Professor Óscar Ribeiro

Ano letivo 2020/2021

Licenciatura em Engenharia de Sistemas Informáticos

Escola Superior de Tecnologia

Instituto Politécnico do Cávado e do Ave

Conteúdo

1. Introdução

Com este trabalho pretende-se a exploração e desenvolvimento de processos de interoperabilidade entre sistemas, assentes em serviços web.

O trabalho foi desenvolvido em duas partes:

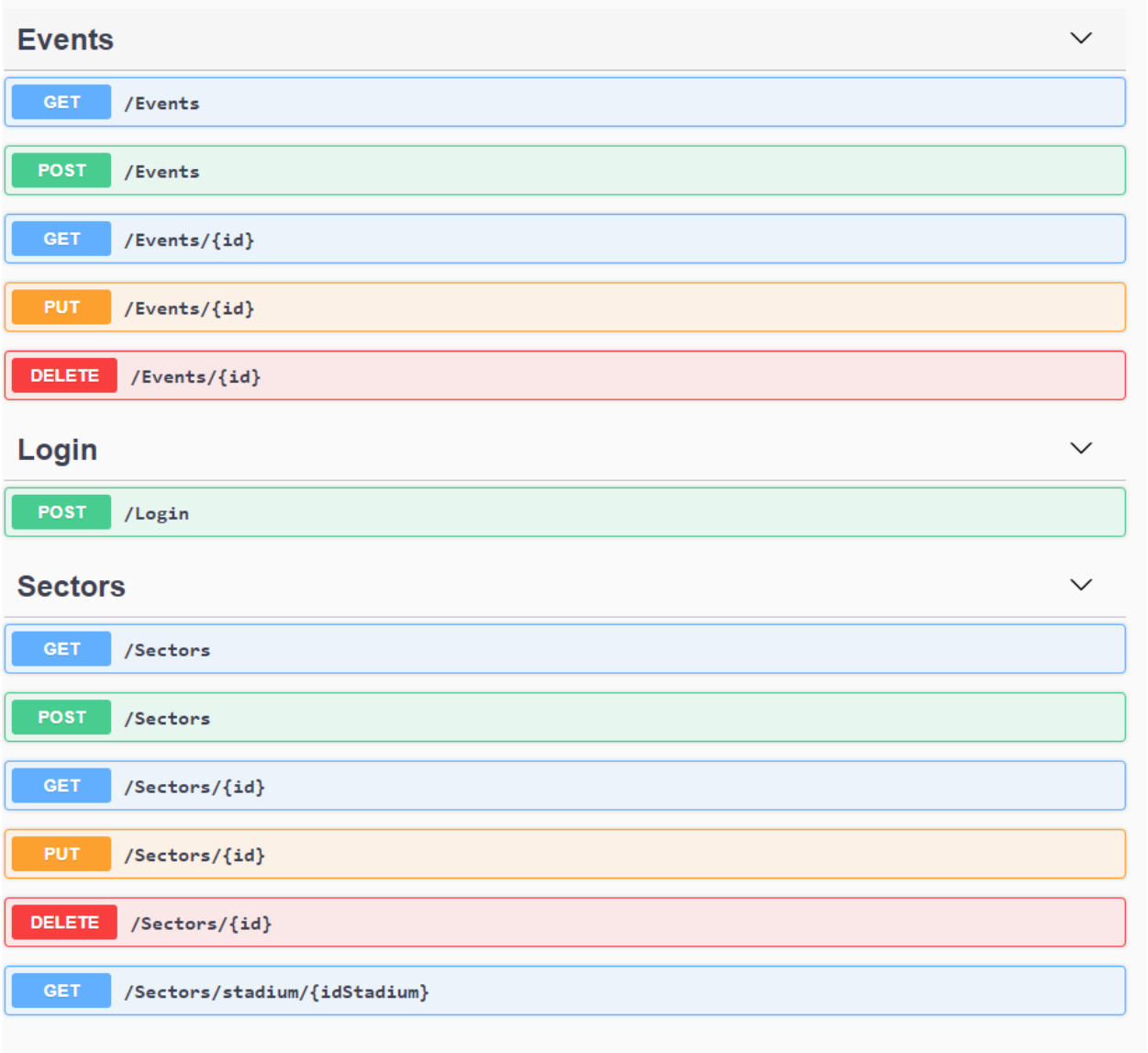
- Servidor;
- Cliente.

Na parte do Servidor foi utilizado ASP NET core versão 5.0 focado para API, enquanto que para a parte do cliente foi feito também em ASP NET core, mas para web application MVC.

O trabalho consiste numa aplicação capaz de gerir estádios, como introduzir um estádio, alterar os seus dados, acrescentar setores, remover setores, criar eventos com número reduzido à percentagem permitida por quem introduzir, etc.

Para um bom manuseamento do projeto utilizamos um repositório do Github.

Foi utilizado SWAGGER para testes durante a realização do servidor.

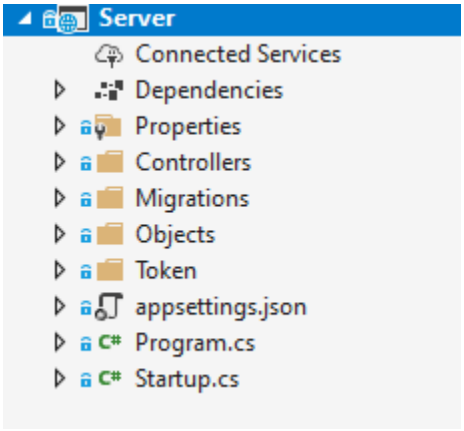


Em cima temos parte do swagger.

2. Servidor

Na parte do servidor temos então ele dividido por partes:

- Controllers: onde são criados os métodos que irão fazer de endpoints – fazem a ligação do exterior com a API.
- Objects: objetos utilizados nesta aplicação.
- Token: JWT utilizado para gerir as autenticações.
- Migration/Entityframework: Para contacto do servidor com a base de dados, foi utilizado em estilo codefirst.



O servidor permite o manuseamento de dados que estão inseridos numa base de dados.

```
services.AddDbContext<DBManagement>(p => p.UseSqlServer(@"Server=.\SQLEXPRESS;Database=ISITable;Trusted_Connection=True;"));
```

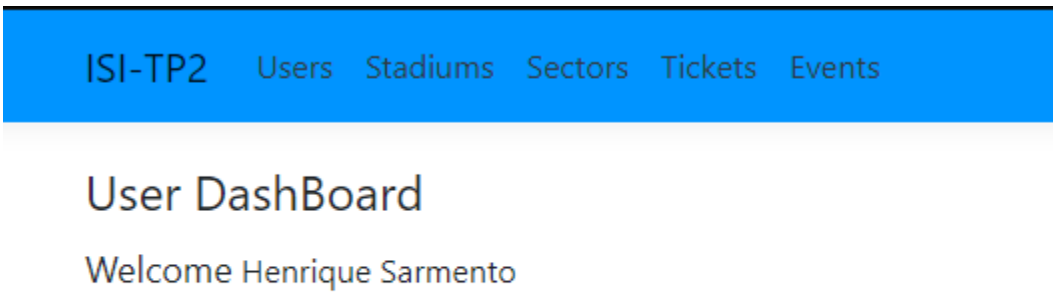
```
0 references
class ContextFactory : IDesignTimeDbContextFactory<DBManagement>
{
    0 references
    public DBManagement CreateDbContext(string[] args)
    {
        string connectionString = @"Server=.\SQLEXPRESS;Database=ISITable;Trusted_Connection=True;";
        var builder = new DbContextOptionsBuilder<DBManagement>();

        builder.UseSqlServer(connectionString);
        return new DBManagement(builder.Options);
    }
}
```

Em cima temos como é feita a ligação do servidor com a base de dados utilizando entityframework.

3. Cliente

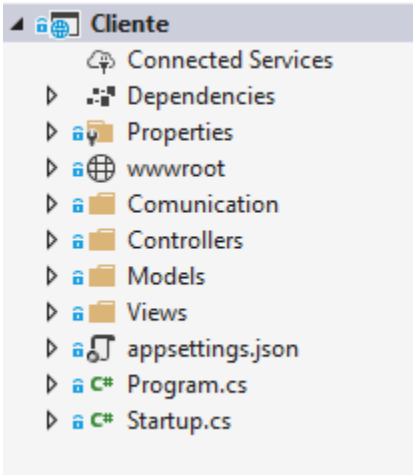
No cliente temos um ambiente WEB, que é utilizado para fazer a ligação entre o utilizador e a api, conseguindo chamar todos os métodos criados na API através de serviços RESTful (POST,GET, etc).



Em cima temos exemplo do cabeçalho depois de iniciada sessão do admin.

Parte do cliente tem como estrutra:

- Communication: Funções que fazem chamadas à API.
- Controllers: Onde são executadas as funções faladas acima e onde é trabalhada a conexão com as Views;
- Views: Onde estão todos os ficheiros cshtml, que vai ser o que o utilizador “vê” quando executa a aplicação;
- Models: Os objetos que são utilizados na aplicação, que são os que vêm da Base de dados criada através do servidor.



4. Ligação Cliente/Servidor

Na parte do servidor, tal como dito em cima, são utilizadas funções que vão servir de endpoint para o contacto com o cliente.

```
[HttpPost]
0 references
public async Task<ActionResult<Event>> PostEvent(Event @event)
{
    int count = 0;
    try
    {
        var exist = await _context.events.AnyAsync(c => c.StadiumID.Equals(@event.StadiumID) && @event.StartDate > c.StartDate && @event.StartDate < c.EndDate && @event.EndDate > c.StartDate && @event.EndDate < c.EndDate);
        if (exist == true)
            return BadRequest();

        var stadiumExist = await _context.stadiums.FirstOrDefaultAsync(i => i.ID.Equals(@event.StadiumID));
        if (stadiumExist == null)
            return BadRequest();
        var sectors = _context.sectors.Where(x => x.StadiumID.Equals(@event.StadiumID));

        foreach(var item in sectors)
        {
            count += item.Capacidade;
        }
        stadiumExist.Capacidade = count;
        @event.Stadium = stadiumExist;
        int capacityEvent = @event.Percentage * @event.Stadium.Capacidade / 100;

        var sector = _context.sectors.Where(x => x.StadiumID.Equals(@event.StadiumID) && x.Capacidade / 2 < capacityEvent).OrderByDescending(x => x.Capacidade).First();
        if (sector == null)
            return BadRequest("Não é possivel realizar evento");
        @event.AvailableSeats = sector.Capacidade;
        @event.Sector = sector;
        _context.events.Add(@event);
        await _context.SaveChangesAsync();

        return CreatedAtAction("GetEvent", new { id = @event.ID }, @event);
    }catch (Exception ex)
    {
        return BadRequest(ex.Message);
    }
```

Na figura anteriormente vista temos então um exemplo de endpoint, que representa uma função RESTful (POST) onde é criado um evento. Em seguida vamos ver como é feita a ligação pela parte do cliente, ou seja, como é chamado o método POST por ele:

```
public static async Task<Event> PostEvent(Event @event)
{
    Event aux = null;
    try
    {
        using (var httpClientHandler = new HttpClientHandler())
        {
            httpClientHandler.ServerCertificateCustomValidationCallback = (message, cert, chain, errors) => { return true; };
            using (var client = new HttpClient(httpClientHandler))
            {
                client.BaseAddress = new Uri("https://localhost:44367/");
                client.DefaultRequestHeaders.Accept.Clear();
                client.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue("application/json"));
                var stringContent = new StringContent(JsonConvert.SerializeObject(@event), Encoding.UTF8, "application/json");
                HttpResponseMessage response = await client.PostAsync(postEvents, stringContent);

                if (response.IsSuccessStatusCode)
                {
                    aux = JsonConvert.DeserializeObject<Event>(await response.Content.ReadAsStringAsync());
                }
                else return null;
            }
        }
    }
    catch (Exception)
    {
        throw;
    }
    return aux;
}
```

Como podemos ver, após na parte cliente ele preencher os campos e enviar para esta função como parâmetro, ele utiliza o objeto, serializa-o num JSON para poder enviar no método POST para a API, sendo depois processado pela API e enviado um objeto para o cliente, ele "desserializa" este objeto para poder recebe-lo no cliente, como prova que foi criado com sucesso.

5. Conclusão

Embora a pouca disponibilidade de ambos os membros por motivos pessoais e outros trabalhos, ao executar este trabalho podemos aplicar tudo o que aprendemos nesta cadeira, assim como explorar novas coisas como a utilização do web application MVC utilizando views, como é utilizada uma API, RESTful, SOAP e até mesmo da conexão com AZURE, que por falta de tempo não foi aplicada neste projeto, mas que a pesquisa e a informação sobre como fazer foi feita e será implementada na mesma após entrega deste trabalho.