2ª entrega

Entrega Final

Integração de Sistemas de Informação

Aluno/os:
21140 - Pedro Vieira Simões
21145 – Gonçalo Moreira da Cunha
21152 – João Carlos da Costa Apresentação

Professor/es: Óscar Ribeiro
Licenciatura em Engenharia de Sistemas Informáticos

Barcelos, janeiro de 2023

IPCA GYM

Resumo

Este projeto prático visou melhorar a performance de trabalho em equipa e explorar as necessidades de um smart campus no IPCA através da criação de um sistema para um ginásio e uma aplicação para os seus utilizadores.

Como parte da unidade curricular de **Integração de sistemas de informação**, o projeto incluiu a construção e interação com a API que sustentou o nosso sistema. Além de demonstrar técnicas e conceitos abordados tanto inter quanto extracurricular, o projeto teve como objetivo ajudar os ginásios a gerenciar o seu negócio de forma mais eficiente e proporcionar uma experiência excecional aos seus clientes.

Conteúdo

Índice de figuras

1. Introdução

1.1. Contextualização

A unidade curricular insere-se na construção da arquitetura do sistema proposto no projeto inicial, implementando a API que irá executar os serviços web. A implementação da API é uma parte importante do projeto, pois ela irá fornecer a base para a comunicação entre o sistema e os utilizadores.

1.2. Motivação e Objetivos

1.2.1. Motivações:

Criar um sistema que possa ajudar a gerir o ginásio e fornecer uma experiência excecional aos seus clientes, á medida que o IPCA cresce e adquire mais instalações.

1.2.2. Objetivos pessoais:

Cimentar os conhecimentos adquiridos durante o percurso académico.

1.2.3. Objetivos do projeto:

Construir a arquitetura do sistema de gestão do ginásio e da aplicação para os clientes.

Desenvolver uma API que suporte serviços web, incluindo:

- Swagger para documentação da API.
- Uma base de dados para armazenar informações relevantes.
- Mecanismo de autenticação para garantir a segurança dos dados.

1.3. Estrutura do Documento

Este documento foi estruturado de forma a ser de fácil leitura, com a inclusão de referências ao material fornecido pelo professor Óscar Ribeiro e/ou excertos de Web grafia. Além disso, o documento foi dividido em tópicos e subseções, de forma a facilitar a navegação e a associação do conteúdo com o material fornecido pelo docente. Acreditamos que essa estrutura tornará mais fácil para o leitor encontrar e compreender as informações relevantes.

2. Produto

2.1. Visão do Produto

O IPCA GYM é uma aplicação mobile e conjunto de hardware de gestão de acessos que se enquadra no subtópico Smart Campus da Saúde, criado para acompanhar a vida saudável e atlética dos estudantes do IPCA e incentivar um estilo de vida saudável.

A aplicação permite aos estudantes acompanhar a sua atividade física e os exercícios que podem fazer durante o treino, enquanto os gestores do ginásio podem monitorizar as pessoas inscritas no ginásio através do sistema de gestão de acesso com chip/cartão eletrónico.

Com o IPCA GYM, os estudantes podem ter um acompanhamento móvel da sua atividade física e os gestores do ginásio podem maximizar a eficiência e oferecer um valor excecional aos seus clientes.

3. Diagramas

3.1. Diagrama Entidade-Relação

Segue-se abaixo o diagrama de entidade-relação da base de dados:

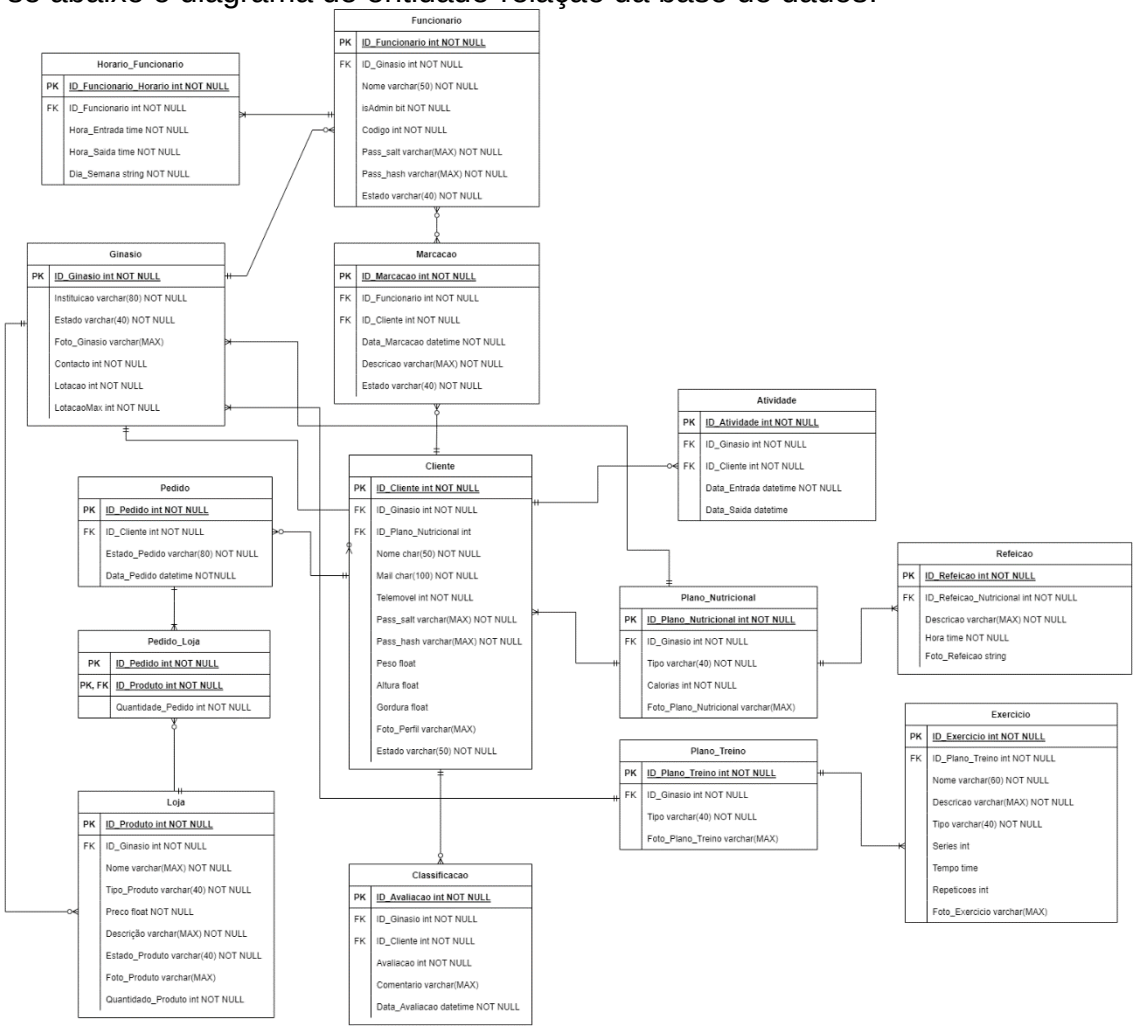


Figura 1 - Diagrama de Entidade-Relação

Este diagrama de entidade-relação possui as seguintes entidades principais:

- **Cliente**: armazena os dados de um cliente que está utilizando a aplicação.
- **Funcionário**: armazena os dados de um funcionário do ginásio, incluindo o atributo "isAdmin" para determinar se ele tem a função de gerente ou não.
- **Ginásio**: armazena os dados do ginásio em questão, de forma que o projeto possa suportar várias instituições acadêmicas no futuro.
- **Loja**: armazena os dados de todos os produtos disponíveis e indisponíveis na loja de cada ginásio.
- **Atividade**: armazena informações sobre as entradas e saídas de cada cliente no ginásio (recebe informação do Arduíno).

Para suportar outras funcionalidades, foram adicionadas as seguintes entidades:

- **Plano_Nutricional** e **Refeição** – armazenam dados sobre diferentes refeições e seus horários, cada ginásio define os seus planos nutricionais.
- **Plano_Treino** e **Exercício** – armazenam dados sobre diferentes exercícios e suas descrições, cada ginásio define os seus planos de treino;
- **Pedido** e **Pedido_Loja**: armazenam informações sobre cada encomenda feita pelo usuário na loja do ginásio em que ele está inscrito.
- **Horario_Funcionario**: regista o horário diário de cada funcionário, para verificar sua disponibilidade para as diferentes marcações.
- **Marcação**: armazena informações sobre as marcações feitas pelo cliente com o funcionário, associadas a cada ginásio.
- **Classificação**: armazena todas as avaliações feitas pelos clientes a cada ginásio.

4. Código

4.1. Programação por Camadas

Neste projeto, optamos por utilizar a arquitetura de programação em camadas para organizar o código de forma mais eficiente e segura. A utilização desta arquitetura permite uma melhor performance, facilita a deteção e correção de anomalias e permite substituir partes ou mesmo toda a camada sem comprometer o sistema inteiro. Esta abordagem de programação torna o código mais organizado e fácil de manter.

Níveis:

1ª Camada - Back-End API
2ª Camada - BLL
3ª Camada - BOL
4ª Camada - DAL

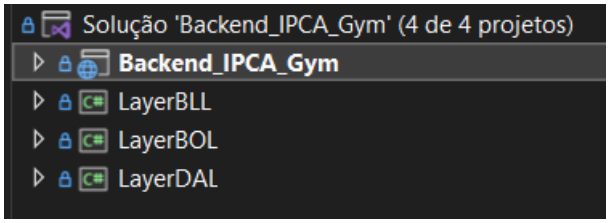


Figura 2 - Camadas

4.1.1. Camada Backend_IPCA_Gym

A camada "Backend_IPCA_Gym" é responsável por gerenciar os Controllers de todas as entidades do nosso sistema. Nesta camada, são executadas as chamadas à API e utilizadas funções lógicas provenientes da camada de Acesso a Dados (DAL).

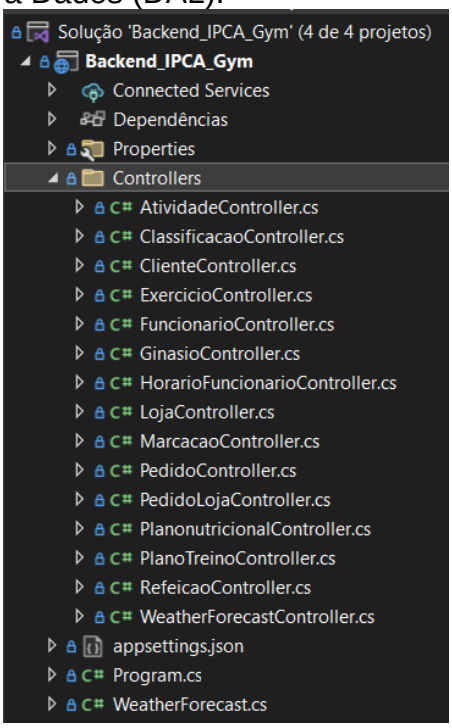


Figura 3 - Lista dos controladores

O Main do sistema também se encontra nesta camada e é invocado quando o sistema é inicializado.

```
//-----------------------------------------------------

builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = false,
            ValidateIssuerSigningKey = true,

            ValidIssuer = builder.Configuration["Jwt:Issuer"],
            ValidAudience = builder.Configuration["Jwt:Audience"],
            IssuerSigningKey = new SymmetricSecurityKey
            (Encoding.UTF8.GetBytes(builder.Configuration["Jwt:Key"]))
        };
    });

// -----------------------------------------------------
var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

//app CORS
//app.UseCors("corsapp");

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Figura 4 - Main

Cada controlador no projeto corresponde a uma parte específica do sistema e é responsável por fazer a conexão com a base de dados através de funções específicas. Antes de executar qualquer chamada, é necessário especificar o tipo de request desejado (como HTTP GET, POST, DELETE, etc.) dentro dos controladores. Isso permite que o sistema saiba como processar e lidar com as solicitações feitas pelos usuários.

- Listagem de todos os clientes

```
public class ClienteController : Controller
{
    private readonly IConfiguration _configuration;
    0 referências
    public ClienteController(IConfiguration configuration)
    {
        _configuration = configuration;
    }

    [HttpGet]
    0 referências
    public async Task<IActionResult> GetAll()
    {
        string sqlDataSource = _configuration.GetConnectionString("DatabaseLink");
        Response response = await ClienteLogic.GetAllLogic(sqlDataSource);

        if (response.StatusCode != LayerBLL.Utils.StatusCodes.SUCCESS) return StatusCode((int)response.StatusCode);

        return new JsonResult(response);
    }
```

Figura 5 - Get dos clientes todos

- Listagem de um cliente através do seu ID

```
[HttpGet("{targetID}")]
0 referências
public async Task<IActionResult> GetByID(int targetID)
{
    string sqlDataSource = _configuration.GetConnectionString("DatabaseLink");
    Response response = await ClienteLogic.GetByIDLogic(sqlDataSource, targetID);

    if (response.StatusCode != LayerBLL.Utils.StatusCodes.SUCCESS) return StatusCode((int)response.StatusCode);

    return new JsonResult(response);
}
```

Figura 6 - Get de um cliente pelo ID

- Criação de um novo cliente

```
[HttpPost]
0 referências
public async Task<IActionResult> Post([FromBody] Cliente newCliente)
{
    string sqlDataSource = _configuration.GetConnectionString("DatabaseLink");
    Response response = await ClienteLogic.PostLogic(sqlDataSource, newCliente);

    if (response.StatusCode != LayerBLL.Utils.StatusCodes.SUCCESS) return StatusCode((int)response.StatusCode);

    return new JsonResult(response);
}
```

Figura 7 - Adicionar Cliente

- Remoção de um cliente

```
[HttpDelete("{targetID}")]
0 referências
public async Task<IActionResult> Delete(int targetID)
{
    string sqlDataSource = _configuration.GetConnectionString("DatabaseLink");
    Response response = await ClienteLogic.DeleteLogic(sqlDataSource, targetID);

    if (response.StatusCode != LayerBLL.Utils.StatusCodes.SUCCESS) return StatusCode((int)response.StatusCode);

    return new JsonResult(response);
}
```

Figura 8 - Remover Cliente

- Alteração dos dados relativos a um cliente

```
[HttpPatch("{targetID}")]
0 referências
public async Task<IActionResult> Patch([FromBody] Cliente cliente, int targetID)
{
    string sqlDataSource = _configuration.GetConnectionString("DatabaseLink");
    Response response = await ClienteLogic.PatchLogic(sqlDataSource, cliente, targetID);

    if (response.StatusCode != LayerBLL.Utils.StatusCodes.SUCCESS) return StatusCode((int)response.StatusCode);


    return new JsonResult(response);
}
```

Figura 9 - Alteração dados de um cliente

4.1.2. Camada BLL

"BLL" significa "Business Logic Layer" e é um termo comum no desenvolvimento de software para se referir a uma camada na arquitetura de uma aplicação. A camada BLL é responsável por implementar a lógica de negócios da aplicação, incluindo tarefas como validar a entrada do usuário, realizar cálculos e interagir com a camada de Acesso a Dados (DAL) para recuperar e armazenar dados. No C#, a camada BLL é implementada como um conjunto de classes que contêm os métodos que executam a lógica de negócios da aplicação. Esses métodos são chamados pela camada de Apresentação (como uma interface do usuário) ou por outros componentes da aplicação para realizar tarefas específicas.

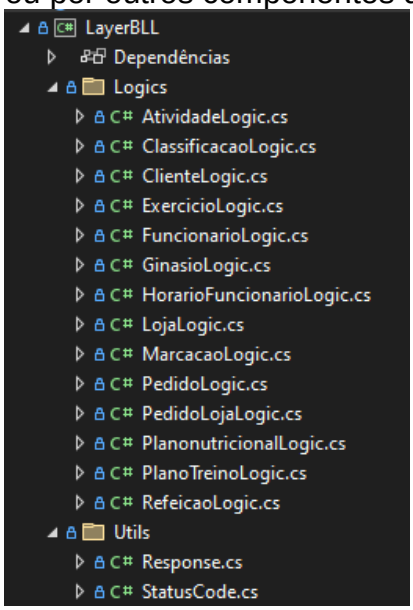


Figura 10 - Camada BLL

- No package *Logics* temos as funções lógicas que são chamadas na camada Back-End para realizar tarefas específicas.
- No package *Utils* temos o código que nos fornece informações sobre o status da solicitação, permitindo verificar se a mesma está funcionando corretamente.

```
public class Response
{
    99+ referências
    public StatusCodes StatusCode { get; set; }
    72 referências
    public string Message { get; set; }
    72 referências
    public object Data { get; set; }

    /// <summary>
    /// Construtor com dados inicializados
    /// </summary>
    /// <param name="statusCode">Código do estado do request</param>
    /// <param name="message">Mensagem do request</param>
    /// <param name="data">Dados que o request envia</param>
    0 referências
    public Response(StatusCodes statusCode, string message, object data)
    {
        StatusCode = statusCode;
        Message = message;
        Data = data;
    }
```

Figura 11 - Exemplo Utils

Seguem-se exemplos, para um cliente do nosso sistema, do que é necessário

para que seja permitido à camada da apresentação fornecer métodos para a camada de negócios.

```
5 referências
public class ClienteLogic
{
    1 referência
    public static async Task<Response> GetAllLogic(string sqlDataSource)
    {
        Response response = new Response();
        List<Cliente> clienteList = await ClienteService.GetAllService(sqlDataSource);

        if (clienteList.Count != 0)
        {
            response.StatusCode = StatusCodes.SUCCESS;
            response.Message = "Lista de clientes obtidos com sucesso";
            response.Data = new JsonResult(clienteList);
        }

        return response;
    }
```

Figura 13 - Logics lista de Clientes

```
public static async Task<Response> GetByIDLogic(string sqlDataSource, int targetID)
{
    Response response = new Response();
    Cliente cliente = await ClienteService.GetByIDService(sqlDataSource, targetID);

    if (cliente != null)
    {
        response.StatusCode = StatusCodes.SUCCESS;
        response.Message = "Cliente obtido com sucesso";
        response.Data = new JsonResult(cliente);
    }

    return response;
}
```

Figura 12 - Logics Amostrar cliente por ID

```
1 referência
public static async Task<Response> PatchLogic(string sqlDataSource, Cliente cliente, int targetID)
{
    Response response = new Response();
    bool updateResult = await ClienteService.PatchService(sqlDataSource, cliente, targetID);

    if (updateResult)
    {
        response.StatusCode = StatusCodes.SUCCESS;
        response.Message = "Success!";
        response.Data = new JsonResult("Cliente alterado com sucesso");
    }

    return response;
}
```

Figura 16 - Logics Alterar dados do Cliente

```
public static async Task<Response> PostLogic(string sqlDataSource, Cliente newCliente)
{
    Response response = new Response();
    bool creationResult = await ClienteService.PostService(sqlDataSource, newCliente);

    if (creationResult)
    {
        response.StatusCode = StatusCodes.SUCCESS;
        response.Message = "Success!";
        response.Data = new JsonResult("Cliente adicionado com sucesso");
    }

    return response;
}
```

Figura 14 - Logics Adicionar Cliente

```
1 referência
public static async Task<Response> DeleteLogic(string sqlDataSource, int targetID)
{
    Response response = new Response();
    bool deleteResult = await ClienteService.DeleteService(sqlDataSource, targetID);

    if (deleteResult)
    {
        response.StatusCode = StatusCodes.SUCCESS;
        response.Message = "Success!";
        response.Data = new JsonResult("Cliente removido com sucesso");
    }

    return response;
}
```

Figura 15 - Logics Remover Cliente

Nestas imagens podemos verificar que em todos os requests (neste caso para o cliente) necessitamos de comunicar com a camada BOL (ClienteService).

4.1.3. Camada BOL

A camada "BOL" (Business Object Layer) é um termo comummente usado no desenvolvimento de software para se referir a uma camada que é responsável por representar entidades de negócios e os seus relacionamentos. A camada BOL normalmente fica entre a camada de apresentação (como uma interface de usuário) e a camada de acesso a dados (que é responsável por comunicar com a base de dados ou outro armazenamento de dados). No C#, a camada de objeto de negócios foi implementada como um conjunto de classes, cada uma correspondendo a uma entidade de negócio e seus relacionamentos.

Essas classes também incluem propriedades que representam os atributos das entidades de negócios e métodos que executam a lógica de negócios. Abaixo, podemos ver um exemplo da classe Ginásio, com seus atributos e métodos devidamente comentados. Ao lado da figura, podemos observar na base de dados do projeto que os campos são os mesmos da tabela Ginásio.

```
public class Ginasio
{
    /// <summary>
    /// Id do ginásio
    /// </summary>
    /// <example>1</example>
    5 referências
    public int id_ginasio { get; set; }
    /// <summary>
    /// Nome da instituição/estabelecimento
    /// </summary>
    /// <example>UMinho</example>
    6 referências
    public string instituicao { get; set; }
    /// <summary>
    /// Estado do ginásio (Ativo ou Inativo)
    /// </summary>
    /// <example>Ativo</example>
    6 referências
    public string estado { get; set; }
    /// <summary>
    /// Foto do ginásio que poderá ser nula
    /// </summary>
    /// <example>C:\OneDrive\ginasio.png</example>
    9 referências
    public string? foto_ginasio { get; set; }
    /// <summary>
    /// Contacto do ginásio
    /// </summary>
    /// <example>911922933</example>
    6 referências
    public int contacto { get; set; }
}
```

Figura 18 - Propriedades Ginásio

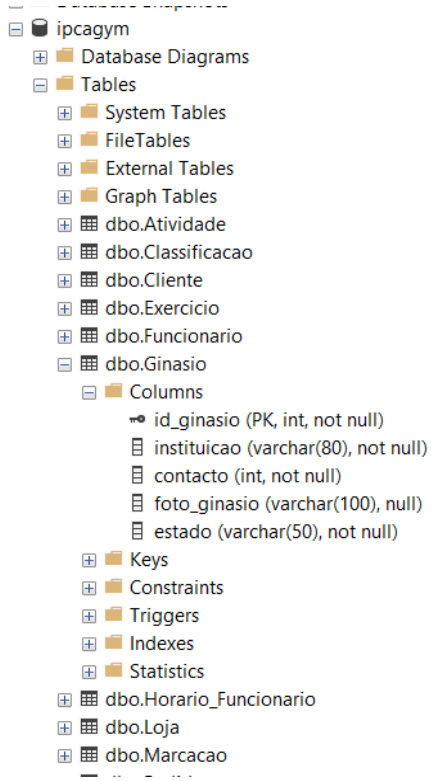


Figura 17 - Propriedades Ginásio DB

4.1.4. Camada DAL

"DAL" significa "Data Access Layer" e é um termo comum usado no desenvolvimento de software para referir-se a uma camada na arquitetura de uma aplicação que é responsável pela comunicação com uma base de dados ou outro armazenamento de dados. A DAL normalmente é responsável por tarefas como executar consultas SQL e interagir com a base de dados para recuperar e armazenar dados.

No C#, a camada de acesso a dados foi, como as restantes, implementada como um conjunto de classes que contém os métodos que executam as interações da base de dados. Esses métodos são chamados pela Business Logic Layer (BLL) e outros componentes na aplicação para recuperar e armazenar dados na base de dados.

```
5 referências
public class GinasioService
{
    /// <summary>
    /// Leitura dos dados de todos os ginásios da base de dados
    /// </summary>
    /// <param name="sqlDataSource">String de conexão á base de dados</param>
    /// <returns>Lista de ginásios se uma leitura bem sucedida, null em caso de erro</returns>
    /// <exception cref="SqlException">Ocorre quando há um erro na conexão com a base de dados.</exception>
    /// <exception cref="InvalidCastException">Ocorre quando há um erro na conversão de dados.</exception>
    /// <exception cref="InvalidOperationException">Trata o caso em que ocorreu um erro de leitura dos dados</exception>
    /// <exception cref="FormatException">Ocorre quando há um erro de tipo de dados.</exception>
    /// <exception cref="IndexOutOfRangeException">Trata o caso em que o índice da coluna da base de dados acessado é inválido</exception>
    /// <exception cref="Exception">Ocorre quando ocorre qualquer outro erro.</exception>
    1 referência
    public static async Task<List<Ginasio>> GetAllService(string sqlDataSource)...

    /// <summary>
    /// Leitura dos dados de um ginásio através do seu id na base de dados
    /// </summary>
    /// <param name="sqlDataSource">String de conexão á base de dados</param>
    /// <param name="targetID">ID do ginásio a ser lido</param>
    /// <returns>Atividade se uma leitura bem sucedida, ou null em caso de erro</returns>
    /// <exception cref="SqlException">Ocorre quando há um erro na conexão com a base de dados.</exception>
    /// <exception cref="InvalidCastException">Ocorre quando há um erro na conversão de dados.</exception>
    /// <exception cref="InvalidOperationException">Trata o caso em que ocorreu um erro de leitura dos dados</exception>
    /// <exception cref="FormatException">Ocorre quando há um erro de tipo de dados.</exception>
    /// <exception cref="IndexOutOfRangeException">Trata o caso em que o índice da coluna da base de dados acessado é inválido</exception>
    /// <exception cref="ArgumentNullException">Ocorre quando um parâmetro é nulo.</exception>
    /// <exception cref="Exception">Ocorre quando ocorre qualquer outro erro.</exception>
    2 referências
    public static async Task<Ginasio> GetByIDService(string sqlDataSource, int targetID)...

    /// <summary>
    /// Inserção dos dados de um novo ginásio na base de dados
    /// </summary>
    /// <param name="sqlDataSource">String de conexão á base de dados</param>
    /// <param name="newGinasio">Objeto com os dados do novo ginásio</param>
    /// <returns>True se a escrita dos dados foi bem sucedida, false em caso de erro.</returns>
    /// <exception cref="SqlException">Ocorre quando há um erro na conexão com a base de dados.</exception>
    /// <exception cref="InvalidCastException">Ocorre quando há um erro na conversão de dados.</exception>
    /// <exception cref="FormatException">Ocorre quando há um erro de tipo de dados.</exception>
    /// <exception cref="ArgumentNullException">Ocorre quando um parâmetro é nulo.</exception>
    /// <exception cref="Exception">Ocorre quando ocorre qualquer outro erro.</exception>
    1 referência
    public static async Task<bool> PostService(string sqlDataSource, Ginasio newGinasio)...
```

Figura 19 - Ginasio Service

Resumidamente, nesta camada é onde os dados estão a ser comunicados diretamente com a base de dados. Como se pode observar, todas as funções estão devidamente comentadas do que executam, ou seja, todas as entidades do nosso sistema têm os serviços específicos e necessários para a correta conexão com a base de dados, seja para inserir, remover, listar e editar dados.

Nas figuras seguintes iremos apresentar a forma de como está a ser realizada cada chamada à base de dados.

Começando com a listagem de entidades, neste caso Horario_Funcionario, construímos 2 funções diferentes, uma para listar todos os horários existentes e outra para apresentar um específico através do seu ID.

Na primeira função, utilizamos uma consulta SQL para selecionar todas as informações da tabela " Horario_Funcionario" da nossa base de dados. Em seguida, utilizamos uma estrutura de repetição para percorrer todas as linhas retornadas pela consulta e criamos um objeto " dia" para cada linha, atribuindo os valores da linha aos atributos do objeto. Por fim, adicionamos cada objeto "dia" à nossa lista de horários e retornamos a lista.

```
1 referência
public static async Task<List<HorarioFuncionario>> GetAllService(string sqlDataSource)
{
    string query = @"select * from dbo.Horario_Funcionario";

    try
    {
        List<HorarioFuncionario> horarios = new List<HorarioFuncionario>();
        SqlDataReader dataReader;

        using (SqlConnection databaseConnection = new SqlConnection(sqlDataSource))
        {
            databaseConnection.Open();
            using (SqlCommand myCommand = new SqlCommand(query, databaseConnection))
            {
                dataReader = myCommand.ExecuteReader();
                while (dataReader.Read())
                {
                    HorarioFuncionario dia = new HorarioFuncionario();

                    dia.id_funcionario_horario = Convert.ToInt32(dataReader["id_funcionario_horario"]);
                    dia.id_funcionario = Convert.ToInt32(dataReader["id_funcionario"]);
                    dia.hora_entrada = (TimeSpan)dataReader["hora_entrada"];
                    dia.hora_saida = (TimeSpan)dataReader["hora_saida"];
                    dia.dia_semana = dataReader["dia_semana"].ToString();

                    horarios.Add(dia);
                }

                dataReader.Close();
                databaseConnection.Close();
            }
        }
        return horarios;
    }
```

Figura 20 – Horario_Funcionario Service - GetAll

Na segunda função, utilizamos uma consulta SQL para selecionar todas as informações da tabela " Horario_Funcionario" da nossa base de dados onde o ID do horário corresponde ao ID passado como parâmetro. Em seguida, criamos um objeto "targetHorarioFuncionario" e atribuímos os valores da linha retornada pela consulta aos atributos do objeto. Por fim, retornamos o objeto.

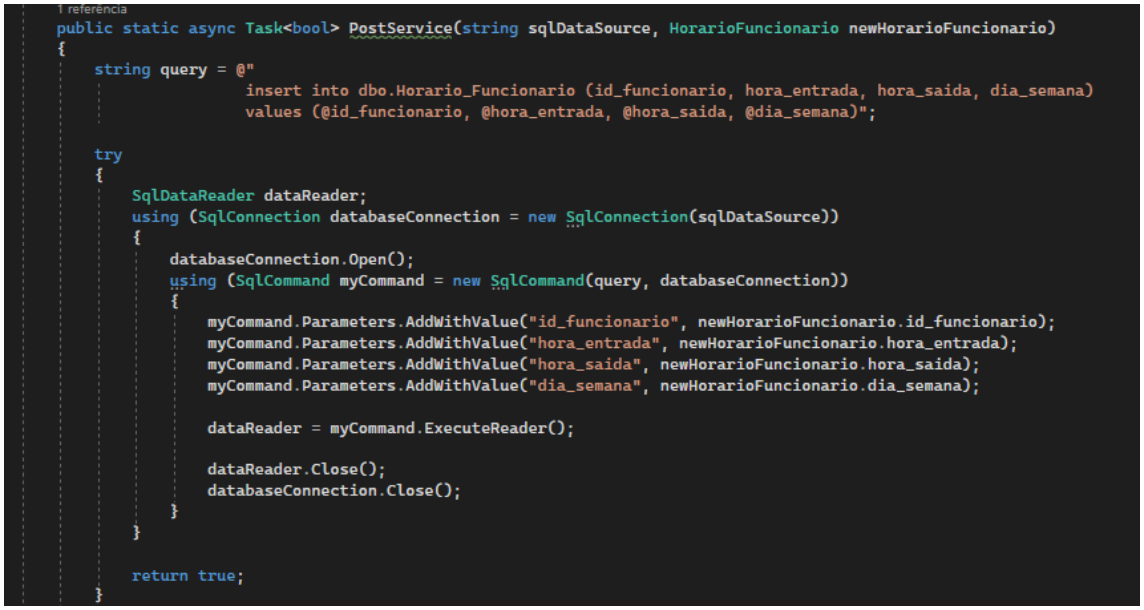
```
public static async Task<HorarioFuncionario> GetByIDService(string sqlDataSource, int targetID)
{
    string query = @"select * from dbo.Horario_Funcionario where id_funcionario_horario = @id_funcionario_horario";

    try
    {
        using (SqlConnection databaseConnection = new SqlConnection(sqlDataSource))
        {
            databaseConnection.Open();
            using (SqlCommand myCommand = new SqlCommand(query, databaseConnection))
            {
                Console.WriteLine(targetID);
                myCommand.Parameters.AddWithValue("id_funcionario_horario", targetID);

                using (SqlDataReader reader = myCommand.ExecuteReader())
                {
                    reader.Read();

                    HorarioFuncionario targetHorarioFuncionario = new HorarioFuncionario();
                    targetHorarioFuncionario.id_funcionario_horario = reader.GetInt32(0);
                    targetHorarioFuncionario.id_funcionario = reader.GetInt32(1);
                    targetHorarioFuncionario.hora_entrada = reader.GetTimeSpan(2);
                    targetHorarioFuncionario.hora_saida = reader.GetTimeSpan(3);
                    targetHorarioFuncionario.dia_semana = reader.GetString(4);

                    reader.Close();
                    databaseConnection.Close();

                    return targetHorarioFuncionario;
                }
            }
```

Figura 21 - Horario_Funcionario Service – By ID

Nas duas funções vemos que a forma como estão implementadas é bastante diferente.

Apesar da forma como as querys estão implementadas serem praticamente iguais, vemos que na primeira função temos a necessidade de colocar especificamente que campos desejamos que aparecem, enquanto na segunda função apenas temos de enviar o ID e a ordem de como queremos que sejam apresentadas as colunas da tabela, essa ordem é definida através do “target…”, onde colocamos no reader a numeração que indica a ordem de como as colunas são apresentadas.

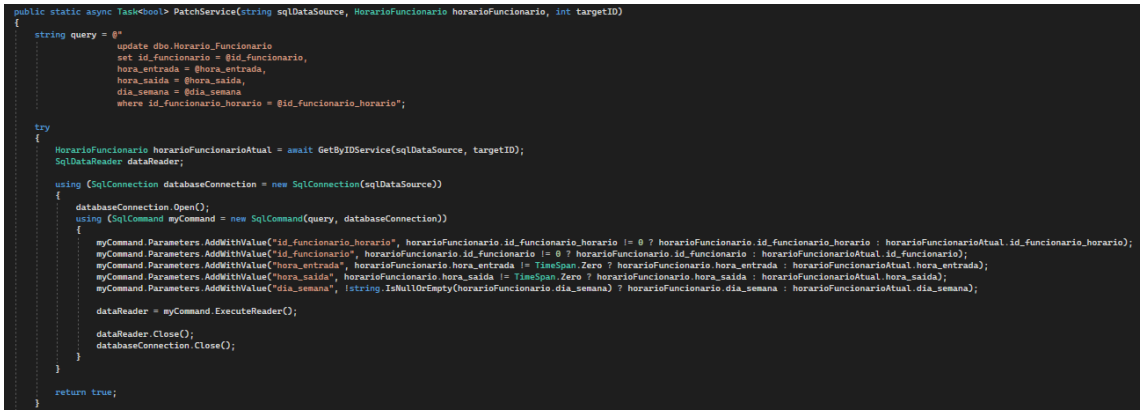
Na seguinte figura podemos ver um exemplo de como é realizado o processo de inserção (**POST**) de dados na base de dados para a entidade "Horario_Funcionario". É necessário especificar todos os campos que serão adicionados e, em seguida, utilizar os comandos adequados para realizar a inserção. É importante notar que este processo pode variar de acordo com o tipo de base de dados que está sendo utilizada e com as especificidades de cada entidade.



```
1 referência
public static async Task<bool> PostService(string sqlDataSource, HorarioFuncionario newHorarioFuncionario)
{
    string query = @"
                    insert into dbo.Horario_Funcionario (id_funcionario, hora_entrada, hora_saida, dia_semana)
                    values (@id_funcionario, @hora_entrada, @hora_saida, @dia_semana)";

    try
    {
        SqlDataReader dataReader;
        using (SqlConnection databaseConnection = new SqlConnection(sqlDataSource))
        {
            databaseConnection.Open();
            using (SqlCommand myCommand = new SqlCommand(query, databaseConnection))
            {
                myCommand.Parameters.AddWithValue("id_funcionario", newHorarioFuncionario.id_funcionario);
                myCommand.Parameters.AddWithValue("hora_entrada", newHorarioFuncionario.hora_entrada);
                myCommand.Parameters.AddWithValue("hora_saida", newHorarioFuncionario.hora_saida);
                myCommand.Parameters.AddWithValue("dia_semana", newHorarioFuncionario.dia_semana);

                dataReader = myCommand.ExecuteReader();

                dataReader.Close();
                databaseConnection.Close();
            }
        }

        return true;
    }
```

Figura 22 - Horario_Funcionario Service – Post

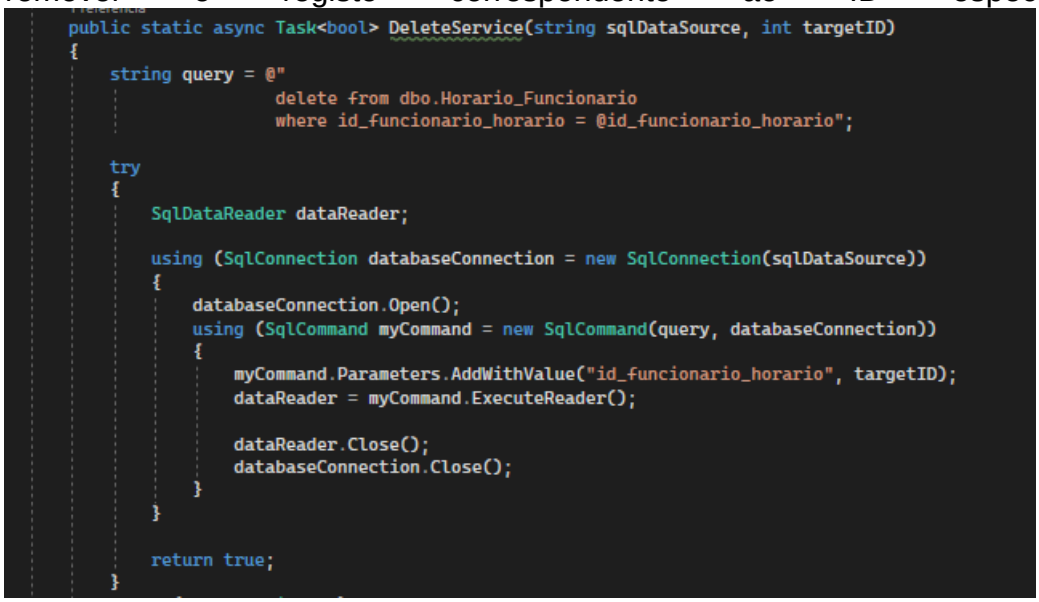
Para uma função de alteração de dados na base de dados (Patch) foi implementada no exemplo de baixo, de forma a receber um objeto com os novos valores a serem atualizados. Essa é uma maneira comum de se fazer a atualização de dados em uma base de dados, pois permite que apenas os valores que forem modificados sejam atualizados e não todos os campos da entidade. Isso pode ser mais eficiente e evitar erros, já que não é necessário fornecer todos os valores novamente. Além disso, é possível observar que a função faz uso de comandos SQL para realizar a atualização dos dados na base de dados.



```
public static async Task<bool> PatchService(string sqlDataSource, HorarioFuncionario horarioFuncionario, int targetID)
{
    string query = @"
                    update dbo.Horario_Funcionario
                    set id_funcionario = @id_funcionario,
                    hora_entrada = @hora_entrada,
                    hora_saida = @hora_saida,
                    dia_semana = @dia_semana
                    where id_funcionario_horario = @id_funcionario_horario";

    try
    {
        HorarioFuncionario horarioFuncionarioAtual = await GetByIDService(sqlDataSource, targetID);
        SqlDataReader dataReader;

        using (SqlConnection databaseConnection = new SqlConnection(sqlDataSource))
        {
            databaseConnection.Open();
            using (SqlCommand myCommand = new SqlCommand(query, databaseConnection))
            {
                myCommand.Parameters.AddWithValue("id_funcionario_horario", horarioFuncionario.id_funcionario_horario != 0 ? horarioFuncionario.id_funcionario_horario : horarioFuncionarioAtual.id_funcionario_horario);
                myCommand.Parameters.AddWithValue("id_funcionario", horarioFuncionario.id_funcionario != 0 ? horarioFuncionario.id_funcionario : horarioFuncionarioAtual.id_funcionario);
                myCommand.Parameters.AddWithValue("hora_entrada", horarioFuncionario.hora_entrada != TimeSpan.Zero ? horarioFuncionario.hora_entrada : horarioFuncionarioAtual.hora_entrada);
                myCommand.Parameters.AddWithValue("hora_saida", horarioFuncionario.hora_saida != TimeSpan.Zero ? horarioFuncionario.hora_saida : horarioFuncionarioAtual.hora_saida);
                myCommand.Parameters.AddWithValue("dia_semana", !string.IsNullOrEmpty(horarioFuncionario.dia_semana) ? horarioFuncionario.dia_semana : horarioFuncionarioAtual.dia_semana);

                dataReader = myCommand.ExecuteReader();

                dataReader.Close();
                databaseConnection.Close();
            }
        }

        return true;
    }
```

Figura 23 - Horario_Funcionario Service - Patch

Para remover um registo de uma entidade, estamos a utilizar apenas o seu ID para identificá-lo e, assim, realizar a operação de exclusão. A imagem abaixo ilustra este processo, onde é criada uma query com o comando DELETE para remover o registo correspondente ao ID especificado.



```
public static async Task<bool> DeleteService(string sqlDataSource, int targetID)
{
    string query = @"
                    delete from dbo.Horario_Funcionario
                    where id_funcionario_horario = @id_funcionario_horario";

    try
    {
        SqlDataReader dataReader;

        using (SqlConnection databaseConnection = new SqlConnection(sqlDataSource))
        {
            databaseConnection.Open();
            using (SqlCommand myCommand = new SqlCommand(query, databaseConnection))
            {
                myCommand.Parameters.AddWithValue("id_funcionario_horario", targetID);
                dataReader = myCommand.ExecuteReader();

                dataReader.Close();
                databaseConnection.Close();
            }
        }

        return true;
    }
```

Figura 24 - Horario_Funcionario Service - Delete

5. JWT Tokens

5.1. Implementação.

Para garantir a autenticação de forma segura, foi implementado o uso de tokens JWT (JSON Web Token). Os tokens JWT são códigos criptografados que permitem a autenticação de usuários em sistemas. Dessa forma, é possível garantir que apenas usuários autenticados tenham acesso às informações solicitadas.

Inicialmente foi configurado o “Program.cs” na implementação de JWT Tokens

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme).AddJwtBearer(options =>
{
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidateIssuerSigningKey = true,
        IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8
            .GetBytes(builder.Configuration.GetSection("AppSettings:Token").Value)),
        ValidateIssuer = false,
        ValidateAudience = false
    };
});
```

Figura 25 - AddAuthentication() no Program.cs

```
app.UseAuthentication();

app.UseRouting();

app.UseAuthorization();
```

Figura 26 - app function calls no Program.cs

As roles de cada utilizador são definidas num ficheiro da camada DAL (Token.cs), garantindo assim que cada utilizador tenha as permissões adequadas para aceder aos dados solicitados (Cliente, Funcionario, Gerente e Admin).

```
public static string CreateTokenFuncionario(Funcionario funcionario, IConfiguration _configuration)
{
    string role = string.Empty;

    if (funcionario.is_admin) role = "Gerente";
    else role = "Funcionario";

    if (funcionario.nome == "adminaccount") role = "Admin";

    List<Claim> claims = new List<Claim>
    {
        new Claim(ClaimTypes.Email, funcionario.codigo.ToString()),
        new Claim(ClaimTypes.Role, role),
        new Claim(ClaimTypes.SerialNumber, funcionario.id_ginasio.ToString())
    };

    var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_configuration.GetSection("AppSettings:Token").Value));

    var creds = new SigningCredentials(key, SecurityAlgorithms.HmacSha512Signature);

    var token = new JwtSecurityToken(
        claims: claims,
        expires: DateTime.Now.AddDays(1),
        signingCredentials: creds);

    var jwt = new JwtSecurityTokenHandler().WriteToken(token);

    return jwt;
}
```

Figura 27 - Criação de uma token para um Funcionário

5.2. Utilização de Authorize nos requests

Para utilizar autenticação nos requests e decidir quem tem acesso, é criado um header nos controladores dos modelos:

```
[HttpGet("{targetID}"), Authorize(Roles = "Admin, Cliente")]
0 references
public async Task<IActionResult> GetByID(int targetID)
```

Figura 28 - Request com autorizações

Para ser obtida informação a partir da token de sessão que fez o request, é utilizado o User.HasClaim() ou User.FindFirstValue()

```
[HttpGet("getbytoken"), Authorize(Roles = "Admin, Cliente")]
0 references
public async Task<IActionResult> GetClienteByToken()
{
    string sqlDataSource = _configuration.GetConnectionString("DatabaseLink");
    string emailCliente = User.FindFirstValue(ClaimTypes.Email);

    Response response = await ClienteLogic.GetClienteByTokenLogic(sqlDataSource, emailCliente);

    if (response.StatusCode != LayerBLL.Utils.StatusCodes.SUCCESS) return StatusCode((int)response.StatusCode);

    return new JsonResult(response);
}
```

Figura 29 - Request para obter informação de um Cliente a partir da Token de Sessão

6. Dependências das Camadas

Tendo em conta que este projeto está dividido em camadas, foi necessário atribuir as dependências a cada camada da seguinte forma:

- Camada do Backend_IPCA_Gym (camada da API) depende da camada de Business Logic (BLL)
- Camada do Business Logic depende da camada de Data Access (DAL)
- Camada de Data Access depende da camada de Business Object (BOL)

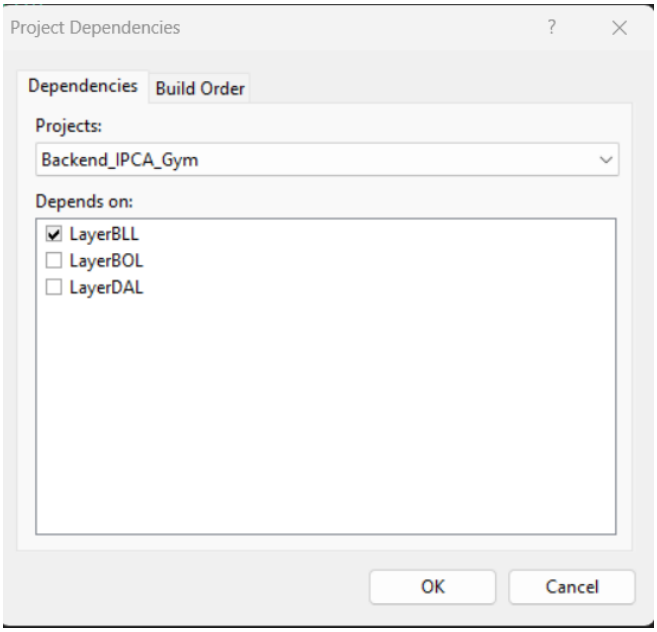


Figura 30 - Dependência API Layer

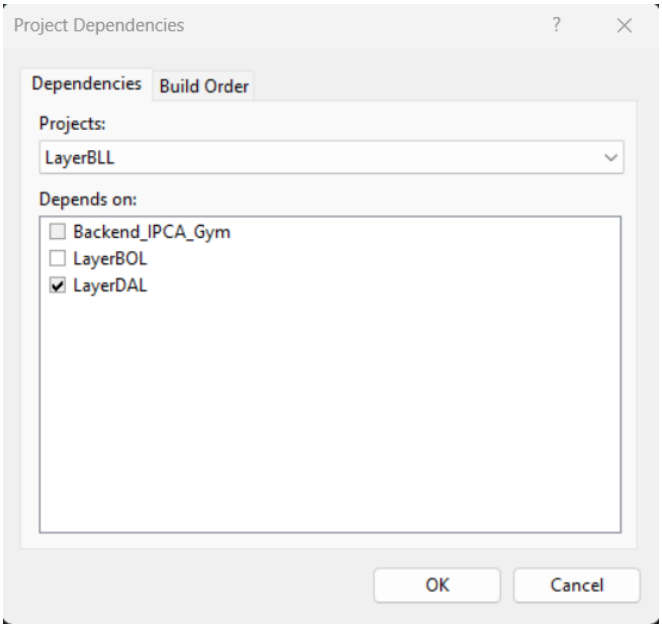


Figura 31 - Dependência BLL

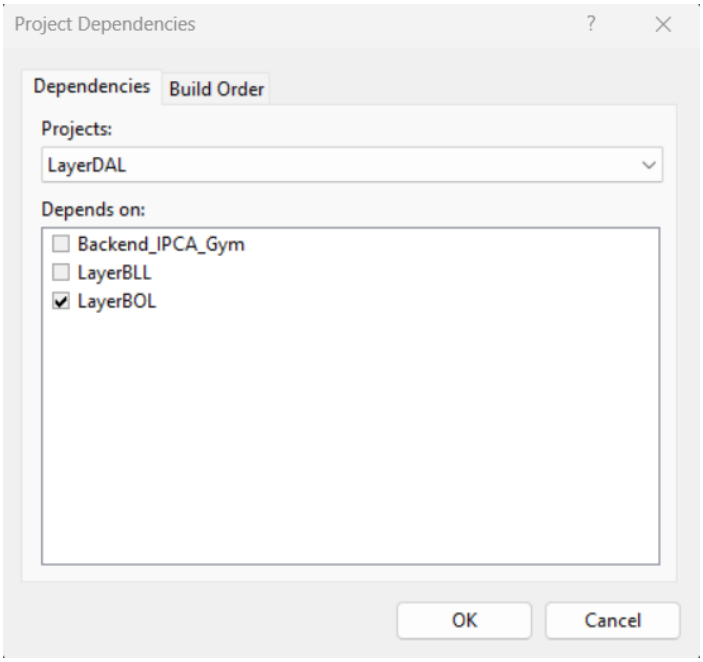


Figura 32 - Dependência BOL

7. WebServices em SOAP

Todos esses métodos foram implementados usando a tecnologia SOAP, que é um padrão para a troca de mensagens estruturadas entre sistemas distribuídos, para acessos mais seguros.

O projeto inclui este WebService com métodos GET, POST, PATCH e DELETE, dos quais foram selecionados os seguintes:

- GET – obter uma lista de todos os clientes na base de dados
- POST – efetuar um login de um funcionário e responde com o tipo de conta que o funcionário fez login (Admin, Gerente, Funcionário)
- PATCH – fazer uma edição de um ginásio na base de dados
- DELETE – fazer uma remoção de um comentário da base de dados

7.1. WebMethod GetAllClientesService() [GET]

Neste WebMethod é feita uma busca à base de dados de uma lista de objetos do tipo Cliente:

```
5 references
public class Cliente
{
    1 reference
    public int id_cliente { get; set; }
    1 reference
    public int id_ginasio { get; set; }
    2 references
    public int id_plano_nutricional { get; set; }
    1 reference
    public string nome { get; set; }
    1 reference
    public string mail { get; set; }
    1 reference
    public int telemovel { get; set; }
    1 reference
    public string pass_salt { get; set; }
    1 reference
    public string pass_hash { get; set; }
    2 references
    public double peso { get; set; }
    2 references
    public int altura { get; set; }
    2 references
    public double gordura { get; set; }
    2 references
    public string foto_perfil { get; set; }
    1 reference
    public string estado { get; set; }

}
```

Figura 33 - Modelo de dados auxiliar (Cliente)

Segue-se agora o código desenvolvido para o WebMethod de obter uma lista de todos os clientes:

```
[WebMethod]
3 references
public List<Cliente> GetAllClientesService()
{
    string query = @"select * from dbo.Cliente";
    string connectionString = "data source=DESKTOP-8AIBMTC;initial catalog=ipcagym;trusted_connection=true";
    List<Cliente> clientes = new List<Cliente>();

    try
    {
        SqlDataReader dataReader;
        using (SqlConnection databaseConnection = new SqlConnection(connectionString))
        {
            databaseConnection.Open();
            using (SqlCommand myCommand = new SqlCommand(query, databaseConnection))
            {
                dataReader = myCommand.ExecuteReader();
                while (dataReader.Read())
                {
                    Cliente cliente = new Cliente();

                    cliente.id_cliente = Convert.ToInt32(dataReader["id_cliente"]);
                    cliente.id_ginasio = Convert.ToInt32(dataReader["id_ginasio"]);
                    if (!Convert.IsDBNull(dataReader["id_plano_nutricional"]))
                    {
                        cliente.id_plano_nutricional = Convert.ToInt32(dataReader["id_plano_nutricional"]);
                    }
                    else
                    {
                        cliente.id_plano_nutricional = -1;
                    }
                    cliente.nome = dataReader["nome"].ToString();
                    cliente.mail = dataReader["mail"].ToString();
                    cliente.telemovel = Convert.ToInt32(dataReader["telemovel"]);
                    cliente.pass_salt = "It's a secret!";
                    cliente.pass_hash = "Already told you that it's a secret!";
                    if (!Convert.IsDBNull(dataReader["peso"]))
                    {
                        cliente.peso = Convert.ToDouble(dataReader["peso"]);
                    }
                    else
                    {
                        cliente.peso = -1;
                    }
                    if (!Convert.IsDBNull(dataReader["altura"]))
                    {
                        cliente.altura = Convert.ToInt32(dataReader["altura"]);
                    }
                    else
                    {
                        cliente.altura = -1;
                    }
                    if (!Convert.IsDBNull(dataReader["gordura"]))
                    {
                        cliente.gordura = Convert.ToDouble(dataReader["gordura"]);
                    }
                    else
                    {
                        cliente.gordura = -1;
                    }
                    if (!Convert.IsDBNull(dataReader["foto_perfil"]))
                    {
                        cliente.foto_perfil = dataReader["foto_perfil"].ToString();
                    }
                    else
                    {
                        cliente.foto_perfil = null;
                    }
                    cliente.estado = dataReader["estado"].ToString();

                    clientes.Add(cliente);
                }

                dataReader.Close();
                databaseConnection.Close();
            }
        }

        return clientes;
    }
    catch (SqlException ex)
    {
        Console.WriteLine("Erro na conexão com a base de dados: " + ex.Message);
        return null;
    }
    catch (InvalidCastException ex)
    {
        Console.WriteLine("Erro na conversão de dados: " + ex.Message);
        return null;
    }
    catch (InvalidOperationException ex)
    {
        Console.WriteLine("Erro de leitura dos dados: " + ex.Message);
        return null;
    }
    catch (FormatException ex)
    {
        Console.WriteLine("Erro de tipo de dados: " + ex.Message);
        return null;
    }
    catch (IndexOutOfRangeException ex)
    {
        Console.WriteLine("Erro de acesso a uma coluna da base de dados: " + ex.Message);
        return null;
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.ToString());
        return null;
    }
}
```

Figura 34 - WebMethod GetAllClientesService()

WebService em funcionamento:

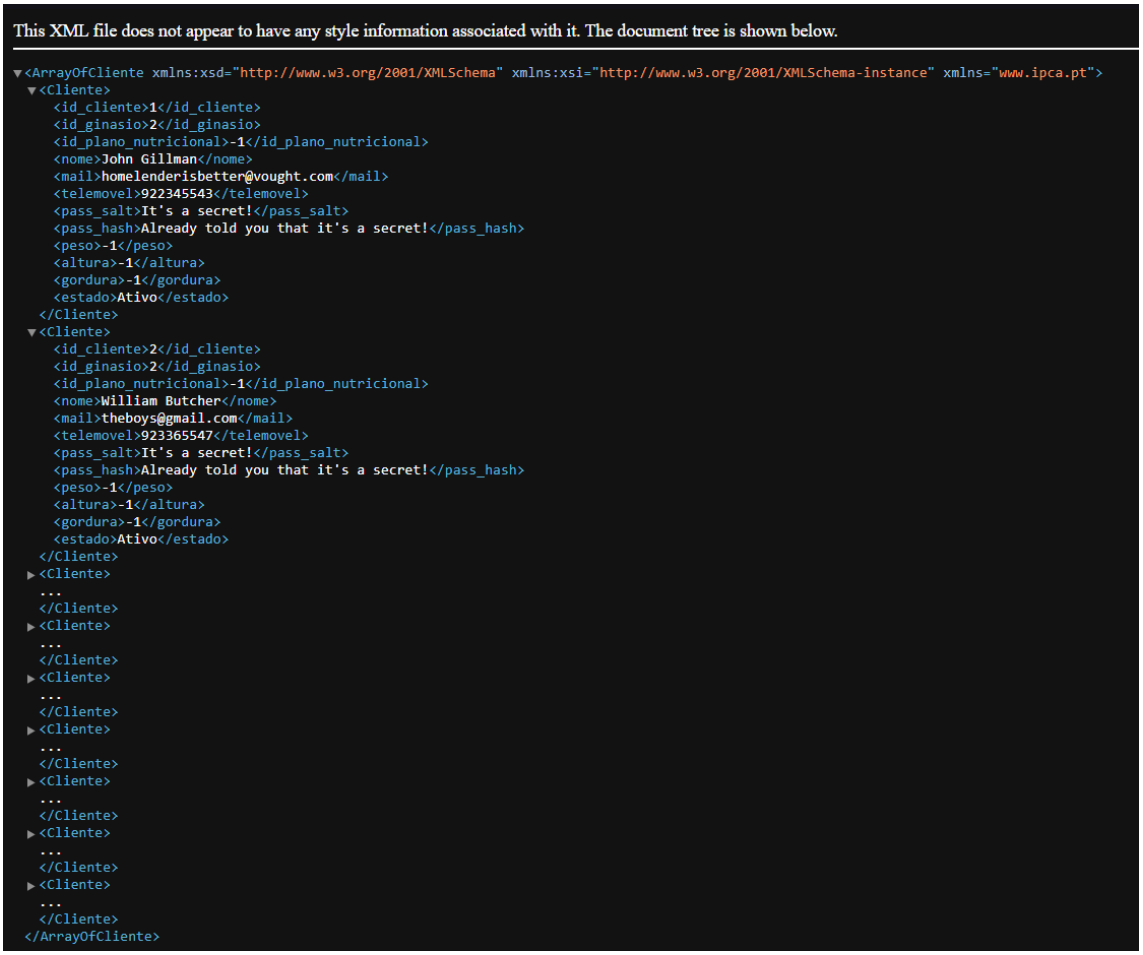


Figura 35 - XML Result de GetAllClientesService()

7.2. WebMethod LoginFuncionarioService() [POST]

O WebMethod para efetuar o login de funcionário retorna o tipo de Role do usuário. Ele foi escolhido para variar os objetivos de cada tipo de request. O Login não insere dados na base de dados, portanto nem um GET, pois precisa de um request body. Uma classe foi criada como modelo de dados para suportar esta funcionalidade:

```
/// <summary> Class auxiliar para a receção de dados de login
4 references
class LoginModel
{
    public bool role;
    public int code;
    public string password;
    public string hash_pass;

    1 reference
    public LoginModel() { }

    0 references
    public LoginModel(bool role, int code, string password, string hash_pass)
    {
        this.role = role;
        this.code = code;
        this.password = password;
        this.hash_pass = hash_pass;
    }
}
```

Figura 36 - Modelo de dados auxiliar (LoginModel)

Segue-se agora o código desenvolvido para efetuar o login de um funcionário:

```
[WebMethod]
0 references
public string LoginFuncionarioService(string codeInput, string passwordInput)
{
    string query = @"
                    select * from dbo.Funcionario
                    where codigo = @codigo and estado != 'Inativo'";

    string connectionString = "data source=DESKTOP-8AIBMTC;initial catalog=ipcagym;trusted_connection=true";
    try
    {
        using (SqlConnection databaseConnection = new SqlConnection(connectionString))
        {
            databaseConnection.Open();
            using (SqlCommand myCommand = new SqlCommand(query, databaseConnection))
            {
                myCommand.Parameters.AddWithValue("codigo", codeInput);
                using (SqlDataReader reader = myCommand.ExecuteReader())
                {
                    reader.Read();

                    string result = string.Empty;
                    LoginModel account = new LoginModel();

                    account.role = reader.GetBoolean(3);
                    account.code = reader.GetInt32(4);
                    account.password = reader.GetString(5);
                    account.hash_pass = reader.GetString(6);
                    string auxForAdmin = reader.GetString(2);

                    reader.Close();
                    databaseConnection.Close();

                    if (!VerifyPasswordHash(passwordInput, Convert.FromBase64String(account.hash_pass), Convert.FromBase64String(account.password)))
                    {
                        throw new ArgumentException("Password Errada.", "conta");
                    }
                    else
                    {
                        if (account.role == true)
                        {
                            if (auxForAdmin == "adminaccount") result = "É admin!";
                            else result = "É gerente!";
                        }
                        else
                        {
                            result = "É funcionário";
                        }
                    }

                    Console.WriteLine(result);
                    return result;
                }
            }

        }
    }
    catch (InvalidOperationException ex)
    {
        Console.WriteLine("Funcionário não existe\n" + ex.Message);
        return "Erro na autenticação";
    }
    catch (SqlException ex)
    {
        Console.WriteLine("Erro na conexão com a base de dados: " + ex.Message);
        return "Erro na autenticação";
    }
    catch (ArgumentNullException ex)
    {
        Console.WriteLine("Erro de parametro inserido nulo: " + ex.Message);
        return "Erro na autenticação";
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.ToString());
        return "Erro na autenticação";
    }
}
```

Figura 37 - WebMethod LoginFuncionarioService()

Foi ainda utilizado um método auxiliar para fazer a desencriptação da palavra-passe:

```
/// <summary> Método auxiliar para verificar se a password está correta
1 reference
public static bool VerifyPasswordHash(string password, byte[] passwordHash, byte[] passwordSalt)
{
    using (var hmac = new HMACSHA512(passwordSalt))
    {
        var computeHash = hmac.ComputeHash(Encoding.UTF8.GetBytes(password));
        return computeHash.SequenceEqual(passwordHash);
    }
}
```

Figura 38 - Método auxiliar VerifyPasswordHash()

WebService em funcionamento:

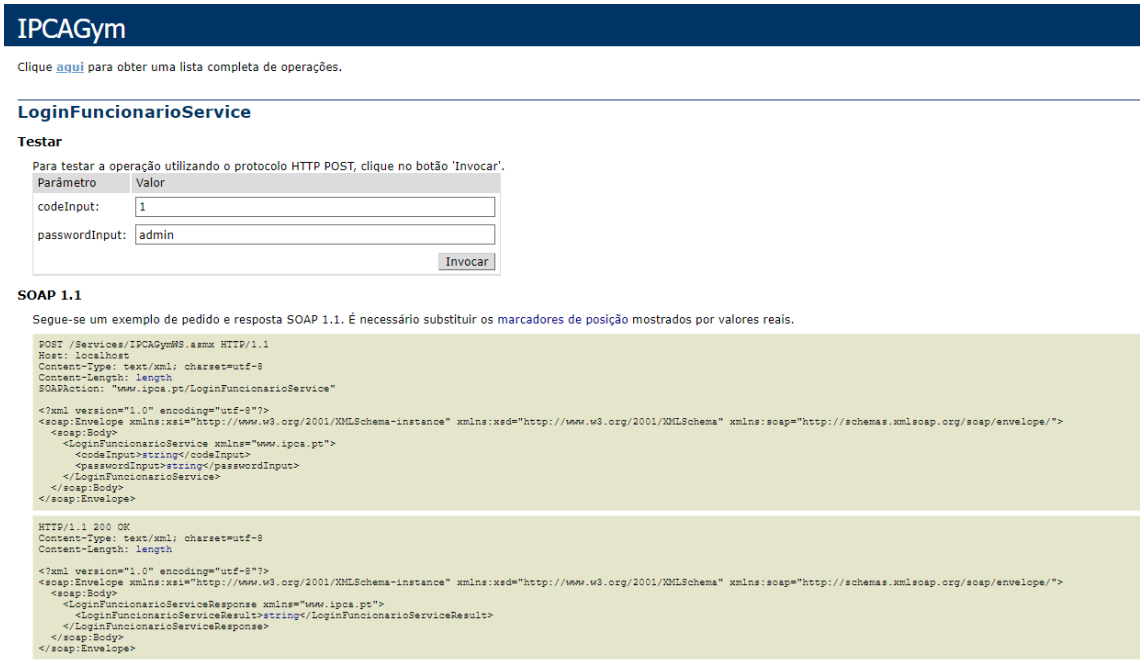


Figura 39 - Input de dados para LoginFuncionarioService()

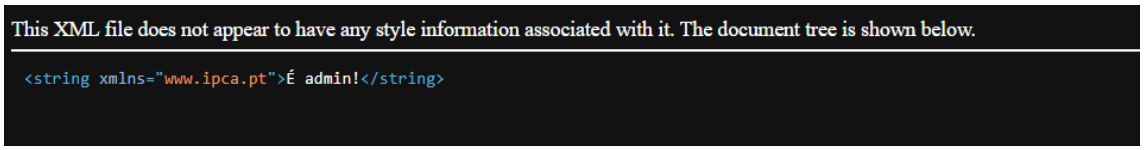


Figura 40 - XML Result de LoginFuncionarioService()

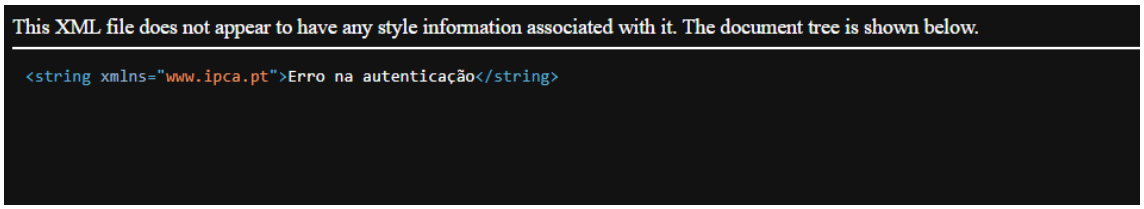


Figura 41 - XML Result de LoginFuncionarioService() no caso de erro

7.3. WebMethod PatchGinasioService() [PATCH]

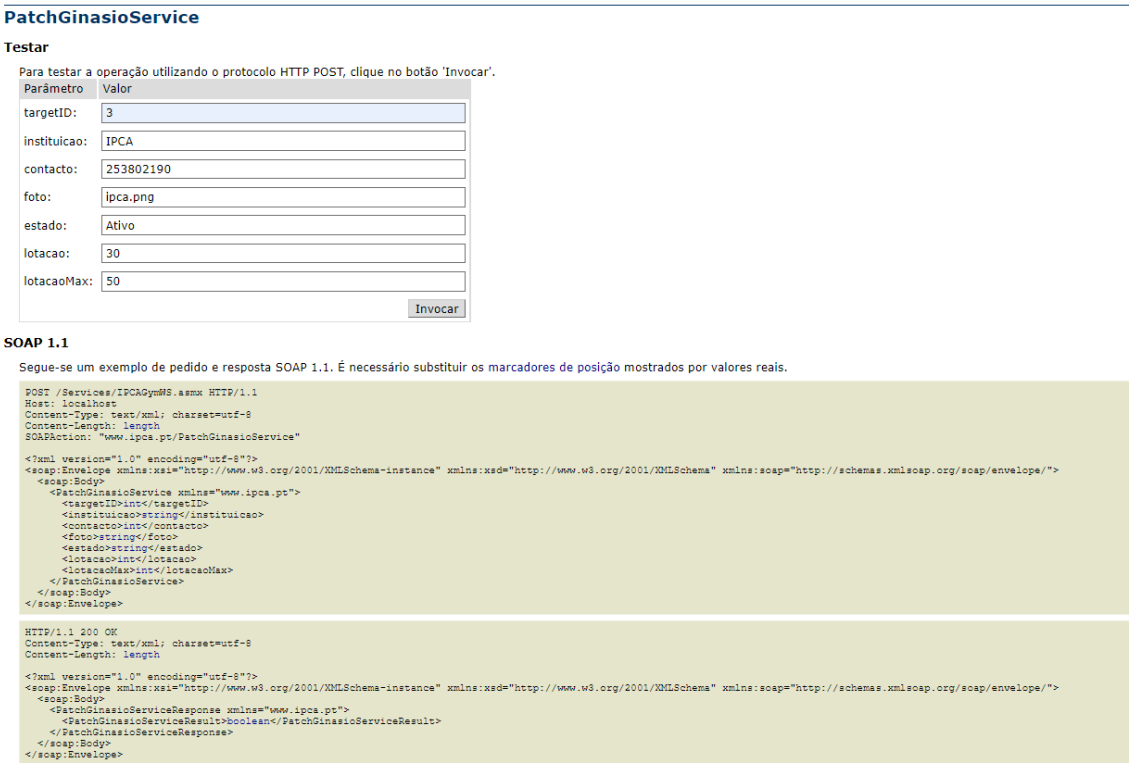
Neste WebMethod é feito um request para fazer a edição de um ginásio.

Segue-se agora o código desenvolvido para efetuar a edição de um ginásio:

```
[WebMethod]
0 references
public bool PatchGinasioService(int targetID, string instituicao, int contacto, string foto, string estado, int lotacao, int lotacaoMax
{
    string queryPatch = @"
                    update dbo.Ginasio
                    set instituicao = @instituicao,
                    contacto = @contacto,
                    foto_ginasio = @foto_ginasio,
                    estado = @estado,
                    lotacao = @lotacao,
                    lotacaoMax = @lotacaoMax
                    where id_ginasio = @id_ginasio";

    string queryGet = @"select * from dbo.Ginasio where id_ginasio = @id_ginasio";
    string connectionString = "data source=DESKTOP-8AIBMTC;initial catalog=ipcagym;trusted_connection=true";

    string instituicaoAtual;
    int contactoAtual;
    string foto_ginasioAtual;
    string estadoAtual;
    int lotacaoAtual;
    int lotacaoMaxAtual;

    try
    {
        SqlDataReader dataReader;

        using (SqlConnection databaseConnection = new SqlConnection(connectionString))
        {
            databaseConnection.Open();
            using (SqlCommand myCommandAux = new SqlCommand(queryGet, databaseConnection))
            {
                myCommandAux.Parameters.AddWithValue("id_ginasio", targetID);

                using (SqlDataReader reader = myCommandAux.ExecuteReader())
                {
                    reader.Read();

                    instituicaoAtual = reader.GetString(1);
                    contactoAtual = reader.GetInt32(2);
                    foto_ginasioAtual = string.Empty;
                    if (!Convert.IsDBNull(reader["foto_ginasio"]))
                    {
                        foto_ginasioAtual = reader.GetString(3);
                    }
                    else
                    {
                        foto_ginasioAtual = null;
                    }

                    estadoAtual = reader.GetString(4);
                    lotacaoAtual = reader.GetInt32(5);
                    lotacaoMaxAtual = reader.GetInt32(6);

                    reader.Close();
                }
            }

            using (SqlCommand myCommand = new SqlCommand(queryPatch, databaseConnection))
            {
                myCommand.Parameters.AddWithValue("id_ginasio", targetID);
                myCommand.Parameters.AddWithValue("instituicao", !string.IsNullOrEmpty(instituicao) ? instituicao : instituicaoAtual);
                myCommand.Parameters.AddWithValue("contacto", contacto == null ? contacto : contactoAtual);

                if (!string.IsNullOrEmpty(foto) && !string.IsNullOrEmpty(foto_ginasioAtual))
                    myCommand.Parameters.AddWithValue("foto_ginasio", !string.IsNullOrEmpty(foto) ? foto : foto_ginasioAtual);
                else
                    myCommand.Parameters.AddWithValue("foto_ginasio", DBNull.Value);

                myCommand.Parameters.AddWithValue("estado", !string.IsNullOrEmpty(estado) ? estado : estadoAtual);
                myCommand.Parameters.AddWithValue("lotacao", lotacao == null ? lotacao : lotacaoAtual);
                myCommand.Parameters.AddWithValue("lotacaoMax", lotacaoMax == null ? lotacaoMax : lotacaoMaxAtual);

                dataReader = myCommand.ExecuteReader();

                dataReader.Close();
                databaseConnection.Close();
            }
        }

        return true;
    }
    catch (SqlException ex)
    {
        Console.WriteLine("Erro na conexao com a base de dados: " + ex.Message);
        return false;
    }
    catch (InvalidCastException ex)
    {
        Console.WriteLine("Erro na conversao de dados: " + ex.Message);
        return false;
    }
    catch (FormatException ex)
    {
        Console.WriteLine("Erro de tipo de dados: " + ex.Message);
        return false;
    }
    catch (ArgumentNullException ex)
    {
        Console.WriteLine("Erro de parametro inserido nulo: " + ex.Message);
        return false;
    }
    catch (ArgumentException ex)
    {
        Console.WriteLine("Valores em falta para ser inseridos: " + ex.Message);
        return false;
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.ToString());
        return false;
    }
}
```

Figura 42 - WebMethod PatchGinásioService()

WebService em funcionamento:



PatchGinasioService

Testar

Para testar a operação utilizando o protocolo HTTP POST, clique no botão 'Invocar'.

Parâmetro	Valor
targetID:	3
instituicao:	IPCA
contacto:	253802190
foto:	ipca.png
estado:	Ativo
lotacao:	30
lotacaoMax:	50

Invocar

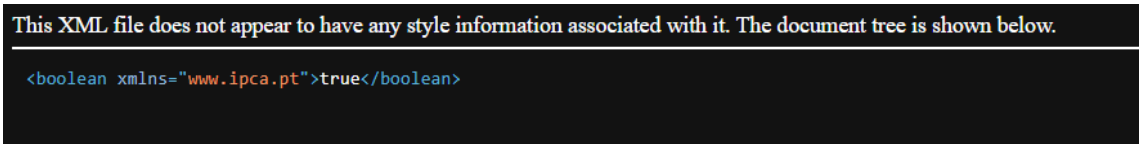
SOAP 1.1

Segue-se um exemplo de pedido e resposta SOAP 1.1. É necessário substituir os marcadores de posição mostrados por valores reais.

```
POST /Services/IPCAGymWS.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "www.ipca.pt/PatchGinasioService"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <PatchGinasioService xmlns="www.ipca.pt">
      <targetID>int</targetID>
      <instituicao>string</instituicao>
      <contacto>int</contacto>
      <foto>string</foto>
      <estado>string</estado>
      <lotacao>int</lotacao>
      <lotacaoMax>int</lotacaoMax>
    </PatchGinasioService>
  </soap:Body>
</soap:Envelope>
```

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <PatchGinasioServiceResponse xmlns="www.ipca.pt">
      <PatchGinasioServiceResult>boolean</PatchGinasioServiceResult>
    </PatchGinasioServiceResponse>
  </soap:Body>
</soap:Envelope>
```

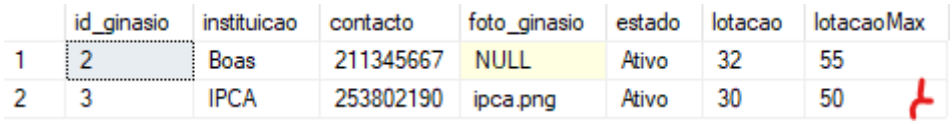
Figura 43 - Input de dados para PatchGinasioService()



This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<boolean xmlns="www.ipca.pt">true</boolean>
```

Figura 44 - XML Result de PatchGinasioService()



	id_ginasio	instituicao	contacto	foto_ginasio	estado	lotacao	lotacaoMax
1	2	Boas	211345667	NULL	Ativo	32	55
2	3	IPCA	253802190	ipca.png	Ativo	30	50

Figura 45 - Resultado do PatchGinasioService() na Base de Dados (SQLServer)

7.4. WebMethod DeleteClassificacaoService() [DELETE]

Neste WebMethod é feito um request para fazer a remoção de um comentário.

Segue-se agora o código desenvolvido para efetuar a remoção de um comentário:

```
[WebMethod]
0 references
public bool DeleteClassificacaoService(int ID_Classificação)
{
    string query = @"
                    delete from dbo.Classificacao
                    where id_avaliacao = @id_avaliacao";
    string connectionString = "data source=DESKTOP-8AIBMTC;initial catalog=ipcagym;trusted_connection=true";

    try
    {

        using (SqlConnection databaseConnection = new SqlConnection(connectionString))
        {
            databaseConnection.Open();
            using (SqlCommand myCommand = new SqlCommand(query, databaseConnection))
            {
                myCommand.Parameters.AddWithValue("id_avaliacao", ID_Classificação);
                int rowsAffected = myCommand.ExecuteNonQuery();

                if (rowsAffected == 0) return false;

                databaseConnection.Close();
            }
        }

        return true;
    }
    catch (SqlException ex)
    {
        Console.WriteLine("Erro/Warning no SQLServer: " + ex.Message);
        return false;
    }
    catch (ArgumentNullException ex)
    {
        Console.WriteLine("Parâmetro inserido é nulo: " + ex.Message);
        return false;
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.ToString());
        return false;
    }
}
```

Figura 46 - WebMethod DeleteClassificacaoService()

WebService em funcionamento:

	id_avaliacao	id_ginasio	id_cliente	avaliacao	comentario	data_avaliacao
1	1	2	1	5	Perfeito	2022-12-16 15:37:35.000
2	2	2	2	4	É bom	2022-12-16 17:37:35.000
3	3	2	3	3	Razoavel, melhorava o atendimento	2022-12-16 20:37:35.000
4	1010	2	1	5	Está ótimo	2022-11-23 12:56:49.000

Figura 47 - Base de dados antes da execução do WebService

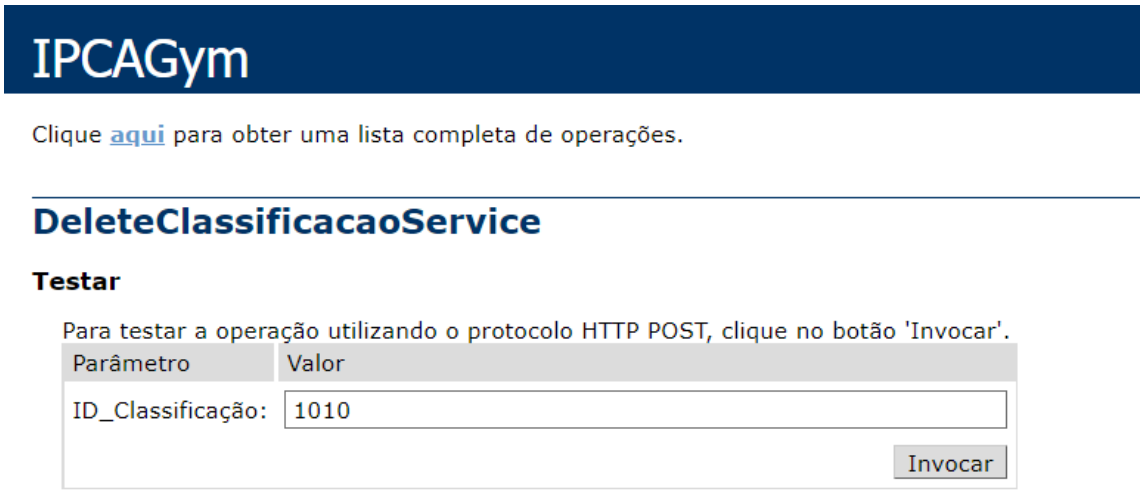


Figura 48 - Input de dados para DeleteClassificacaoService()

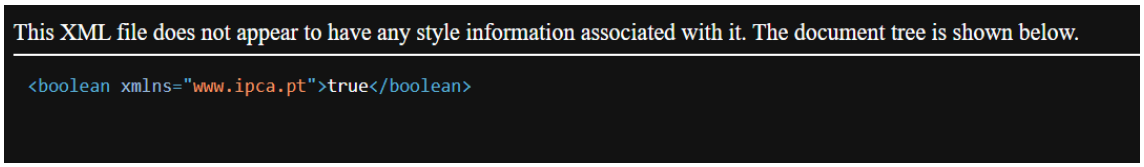


Figura 49 - XML Result de DeleteClassificacaoService()

	id_avaliacao	id_ginasio	id_cliente	avaliacao	comentario	data_avaliacao
1	1	2	1	5	Perfeito	2022-12-16 15:37:35.000
2	2	2	2	4	É bom	2022-12-16 17:37:35.000
3	3	2	3	3	Razoavel, melhorava o atendimento	2022-12-16 20:37:35.000

Figura 50 - Base de Dados após a execução do DeleteClassificacaoService()

8. Testes Unitários

O objetivo dos testes unitários foi verificar se a API está a funcionar corretamente, validando a resposta de cada request. Usamos o XUnit para criar esses testes.

Verificamos o código de status retornado pelos requests e também se a resposta é não nula. Para requests que devem ter sucesso (retorna JsonResult), esperamos um código de status 200 e uma resposta não nula. Para requests que não devem ter sucesso (retorna StatusCodeResult), esperamos um código de status 204 e uma resposta não nula. Seguem os testes realizados e as respetivas funções de teste:

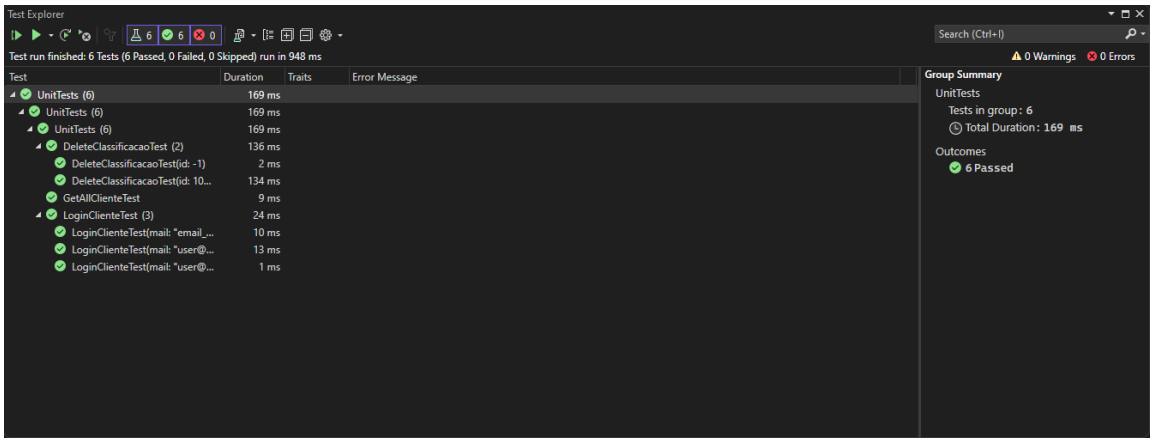


Figura 51 - Execução dos Testes Unitários

Para fazer a criação de Controladores da API para montar os dados e enviá-los para os respetivos requests, foi necessário fazer o carregamento do appsettings.json da API para ser utilizado como parâmetro do construtor de cada Controller:

```
/// <summary>
/// Configuração utilizada para a criação de controladores, para que seja possível fazer as function calls
/// </summary>
/// <returns></returns>
3 references | 6/6 passing
public static IConfiguration InitConfiguration()
{
    var config = new ConfigurationBuilder().AddJsonFile("C:\\TrabalhosPraticos\\Projeto_Aplicado\\ipca_gym" +
        "\\Backend_IPCA_Gym\\Backend_IPCA_Gym\\appsettings.json").AddEnvironmentVariables().Build();

    return config;
}
```

Figura 52 - Carregamento de informação do appsetings.json para uma configuração

8.1. Teste GetAllClienteTest

Teste para o request de obter todos os clientes da base de dados.

```
/// <summary>
/// Teste no request de obter todos os clientes da Base de Dados
/// Esperado:
///     - No caso de retornar um JsonResult: Não ser nulo e ter code 200 ou 204
///     - No caso de retornar um StatusCodeResult: Deve ser code 204
/// </summary>
[Fact]
| 0 references
public async void GetAllClienteTest()
{
    var config = InitConfiguration();
    var clienteController = new ClienteController(config);

    IActionResult requestResult = await clienteController.GetAll();

    requestResult.Should().NotBeNull();


    var statusCodeResult = requestResult as StatusCodeResult;
    if (statusCodeResult != null)
    {
        statusCodeResult.Should().NotBeNull();
        statusCodeResult.StatusCode.Should().Be(204);
    }

    var jsonResult = requestResult as JsonResult;
    if (jsonResult != null)
    {
        var response = jsonResult.Value as Response;
        if (response != null)
        {
            response.Should().NotBeNull();
            response.StatusCode.Should().Be(StatusCodes.SUCCESS);
        }
    }
}
```

Figura 53 - Método teste GetAllClientes()

8.2. Teste LoginClienteTest

Teste para a funcionalidade da API de fazer Login numa conta de um cliente. Para este caso foi utilizado 3 conjuntos de dados distintos, de forma a testar todas as situações possíveis.

```
/// <summary>
/// Teste no request de fazer login
/// Esperado:
///     - No caso de retornar um JsonResult: Não ser nulo e ter code 200 ou 204
///     - No caso de retornar um StatusCodeResult: Deve ser code 204
/// Dados introduzidos:
///     - Primeiro InlineData com email errado
///     - Segundo InlineData com dados corretos
///     - Terceiro InlineData com password errada
/// </summary>
[Theory]
[InlineData("email_incorreto", "dm")]
[InlineData("user@gmail.com", "password")]
[InlineData("user@gmail.com", "pass_incorreta")]
0 references
public async void LoginClienteTest(string mail, string password)
{
    var config = InitConfiguration();
    var clienteController = new ClienteController(config);

    LoginCliente modelTest = new LoginCliente();

    modelTest.mail = mail;
    modelTest.password = password;

    IActionResult requestResult = await clienteController.Login(modelTest);

    requestResult.Should().NotBeNull();

    var statusCodeResult = requestResult as StatusCodeResult;
    if (statusCodeResult != null)
    {
        statusCodeResult.Should().NotBeNull();
        statusCodeResult.StatusCode.Should().Be(204);
    }

    var jsonResult = requestResult as JsonResult;
    if (jsonResult != null)
    {
        var response = jsonResult.Value as Response;
        if (response != null)
        {
            response.Should().NotBeNull();
            response.StatusCode.Should().Be(StatusCodes.SUCCESS);
        }
    }
}
```

Figura 54 - Método Teste LoginCliente()

8.3. Teste DeleteClassificaçãoTest

Teste para a funcionalidade da API de fazer remover uma classificação de um ginásio da base de dados.
Para este caso foi utilizado 2 conjuntos de dados distintos, de forma a testar todas as situações possíveis.

```
/// <summary>
/// Teste no request de fazer login
/// Esperado:
///     - No caso de retornar um JsonResult: Não ser nulo e ter code 200 ou 204
///     - No caso de retornar um StatusCodeResult: Deve ser code 204
/// Dados introduzidos:
///     - 1011 (ID anteriormente existente na Base de Dados)
///     - -1 (ID inexistente)
/// </summary>
[Theory]
[InlineData(1011)]
[InlineData(-1)]
0 references
public async void DeleteClassificacaoTest(int id)
{
    var config = InitConfiguration();
    var classficacaoController = new ClassificacaoController(config);

    IActionResult requestResult = await classficacaoController.Delete(id);

    requestResult.Should().NotBeNull();

    var statusCodeResult = requestResult as StatusCodeResult;
    if (statusCodeResult != null)
    {
        statusCodeResult.Should().NotBeNull();
        statusCodeResult.StatusCode.Should().Be(204);
    }

    var jsonResult = requestResult as JsonResult;
    if (jsonResult != null)
    {
        var response = jsonResult.Value as Response;
        if (response != null)
        {
            response.Should().NotBeNull();
            response.StatusCode.Should().Be(StatusCodes.SUCCESS);
        }
    }
}
```

Figura 55 - Método Teste DeleteClassificacao()

9. Documentação OpenAPI (SwaggerUI)

Para fazer a documentação da API com o SwaggerUI, foi necessário adicionar o pacote Swashbuckle.AspNetCore e configurá-lo no arquivo de inicialização da API (Program.cs). Em seguida, foi criada uma classe de configuração do Swagger, onde foram definidos os detalhes da API, como o título, descrição e versão.

Começou-se por fazer a implementação do gerador do Swagger no Program.cs (que funciona como Startup da API):

```
c.SwaggerDoc("Alpha", new OpenApiInfo
{
    Title = "IPCAGym",
    Version = "Alpha",
    Description = "Web API para o aplicativo IPCAGym"
});
```

Figura 56 - Geração documentação no Swagger

Foi também exportada toda a documentação para um ficheiro .xml externo que se encontra no seguinte diretório:

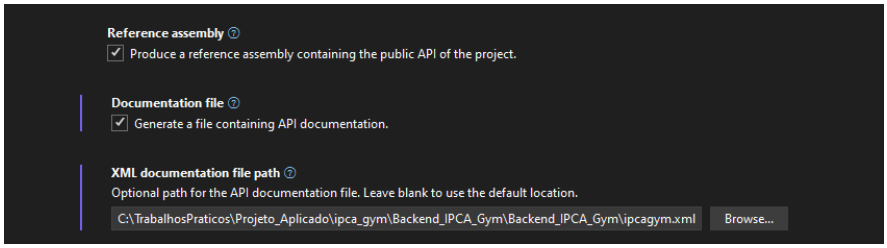


Figura 57 - Propriedades da Solução do projeto

A forma que foi usada para a documentação de métodos e classes foi a seguinte:

```
/// <summary>
/// Método http post para inserção de um novo ginásio
/// </summary>
/// <param name="newGinasio">Dados do novo ginásio a ser inserido</param>
/// <returns>Resposta do request que contém a sua mensagem e seu código em formato json</returns>
[HttpPost, Authorize(Roles = "Admin")]
0 references
public async Task<IActionResult> Post([FromBody] Ginasio newGinasio)
{
    string sqlDataSource = _configuration.GetConnectionString("DatabaseLink");
    Response response = await GinasioLogic.PostLogic(sqlDataSource, newGinasio);

    if (response.StatusCode != LayerBLL.Utils.StatusCodes.SUCCESS) return StatusCode((int)response.StatusCode)

    return new JsonResult(response);
}
```

Figura 58 - Documentação de métodos

```
namespace LayerBOL.Models
{
    19 references
    public class Ginasio
    {
        /// <summary>
        /// Id do ginásio
        /// </summary>
        /// <example>1</example>
        5 references
        public int id_ginasio { get; set; }
        /// <summary>
        /// Nome da instituição/estabelecimento
        /// </summary>
        /// <example>UMinho</example>
        6 references
        public string instituicao { get; set; }
        /// <summary>
        /// Estado do ginásio (Ativo ou Inativo)
        /// </summary>
        /// <example>Ativo</example>
        6 references
        public string estado { get; set; }
        /// <summary>
        /// Foto do ginásio que poderá ser nula
        /// </summary>
        /// <example>C:\OneDrive\ginasio.png</example>
        11 references
        public string? foto_ginasio { get; set; }
        /// <summary>
        /// Contacto do ginásio
        /// </summary>
        /// <example>911922933</example>
        10 references
        public int contacto { get; set; }
```

Figura 59 - Documentação de modelos de dados

Sendo este o resultado no Swagger:

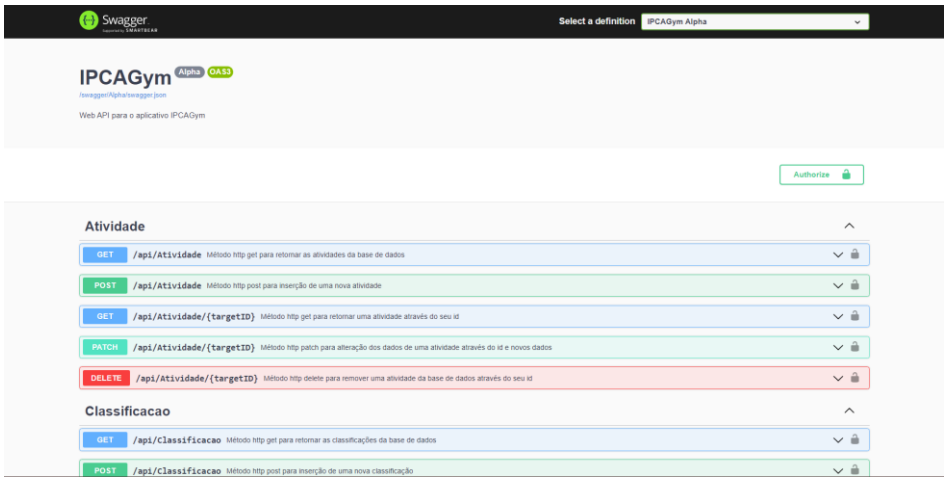


Figura 60 - Resultado documentação no Swagger

10. Packages (Nuggets)

Nesta lista de pacotes Nuget usados, podemos observar que as camadas Api, BLL, BOL e DAL utilizam o pacote Swashbuckle.AspNetCore, que é uma ferramenta para gerar documentação da API usando o OpenAPI (anteriormente conhecido como Swagger). Também podemos ver que as camadas Api e DAL utilizam o pacote Npgsql, que é um driver de banco de dados para conectar a aplicação ao PostgreSQL. A camada Api também usa o pacote Microsoft.AspNetCore.Authentication.JwtBearer para autenticação JWT e a camada BLL usa o pacote Microsoft.AspNetCore.Mvc.Formatters.Json para formatação de respostas Json. As camadas BOL e DAL também usam o pacote System.IdentityModel.Tokens.Jwt, que fornece suporte para a manipulação de tokens JWT. Por fim, a camada DAL usa o pacote System.Data.SqlClient para se conectar ao SQL Server.

11. Conclusão

Conclui-se que este trabalho prático foi uma oportunidade de aprendizagem e melhoria de habilidades na implementação de uma API em C#, com segurança, organização e testes unitários. Foi também uma oportunidade para adquirir novos conhecimentos, como trabalhar com strings de query para o SQL e dividir o projeto em camadas de forma eficiente. Estas habilidades e conhecimentos serão valiosos em futuros projetos e desenvolvimentos.

12. Bibliografia

Repositório GitHub
https://github.com/Presentation12/Ipca_Gym

Figma
https://www.figma.com/file/Q4tM34gl91b9fhrGvUeXRs/MileriuPT's-teamlibrary?node-id=0%3A1&t=p9cWit1VJHUNwf2m-1

Material fornecido pelo docente:
https://elearning2.ipca.pt/2223/course/view.php?id=10611