2ª entrega

Entrega intermédia

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, dezembro de 2022

IPCA GYM

Resumo

Este trabalho prático, relativo à unidade curricular de **Integração de sistemas de informação,** propende a melhorar a performance de trabalho em equipa num desafio que irá explorar as necessidades de um smart campus, no IPCA e demonstrar técnicas e conceitos abordados inter e extracurricular.

A ideia principal do projeto será um sistema para um ginásio e uma aplicação para os utilizadores. Nesta unidade curricular em específico iremos abordar a construção e interação com a API que irá sustentar o nosso projeto.

Conteúdo

Índice de figuras

1. Introdução

1.1. Contextualização

Provindo da ideia do projeto inicial, esta unidade curricular tem como propósito a construção da arquitetura do sistema, implementando a API do projeto que irá executar serviços web.

1.2. Motivação e Objetivos

A ideia de um sistema para o ginásio foi originada pela ideia de futuramente o IPCA vir a ter mais instalações à medida que este vai crescendo e desta forma existir uma forma de gerir o mesmo e ainda ajudar os clientes.

Temos por objetivos pessoais:

- Cimentar conhecimentos obtidos ao longo do percurso académico;

Objetivos do projeto:

- Construir a arquitetura do sistema
- Montagem de uma API que suporte serviços web, incluído:
  - Swagger;
  - Base de dados;
  - Autenticação;

1.3. Estrutura do Documento

O documento está estruturado de forma que seja de simples leitura. Existe recurso a referências de material fornecido pelo professor Óscar Ribeiro e/ou referências a excertos de Web grafia.

Este trabalho encontra-se também dividido em grupos, de forma a facilitar a procura e associação face ao material fornecido pelo docente.

2. Produto

2.1. Visão do Produto

Dentro dos subtópicos possíveis encaixados no Smart Campus vai ser abordado a Saúde. Foi decidido toda uma construção em torno do desenvolvimento android que visa à nossa universidade acompanhar a vida saudável e atlética dos estudantes.

O IPCA GYM nasce após notar-se a necessidade desse mesmo acompanhamento e a falta de um setor que permita a atividade aos jovens, no sentido de incentivar aos estudantes a realizar um estilo de vida saudável.

Será então possível aos estudantes terem um acompanhamento mobile da sua atividade física, tal como os diferentes exercícios que pode fazer ao longo do seu treino.

Os gestores do ginásio conseguirão fazer uma monitorização de todas as pessoas inscritas no ginásio, já que, em conjunto com outra unidade curricular, irá ser implementado um sistema externo para gestão de acesso através de um chip/cartão eletrónico.

Este projeto visa alcançar este objetivo através da implementação de uma aplicação Mobile e de hardware de gestão de acessos para que se torne mais cómoda a utilização da mesma.

3. Diagramas

3.1. Diagrama Entidade-Relação

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

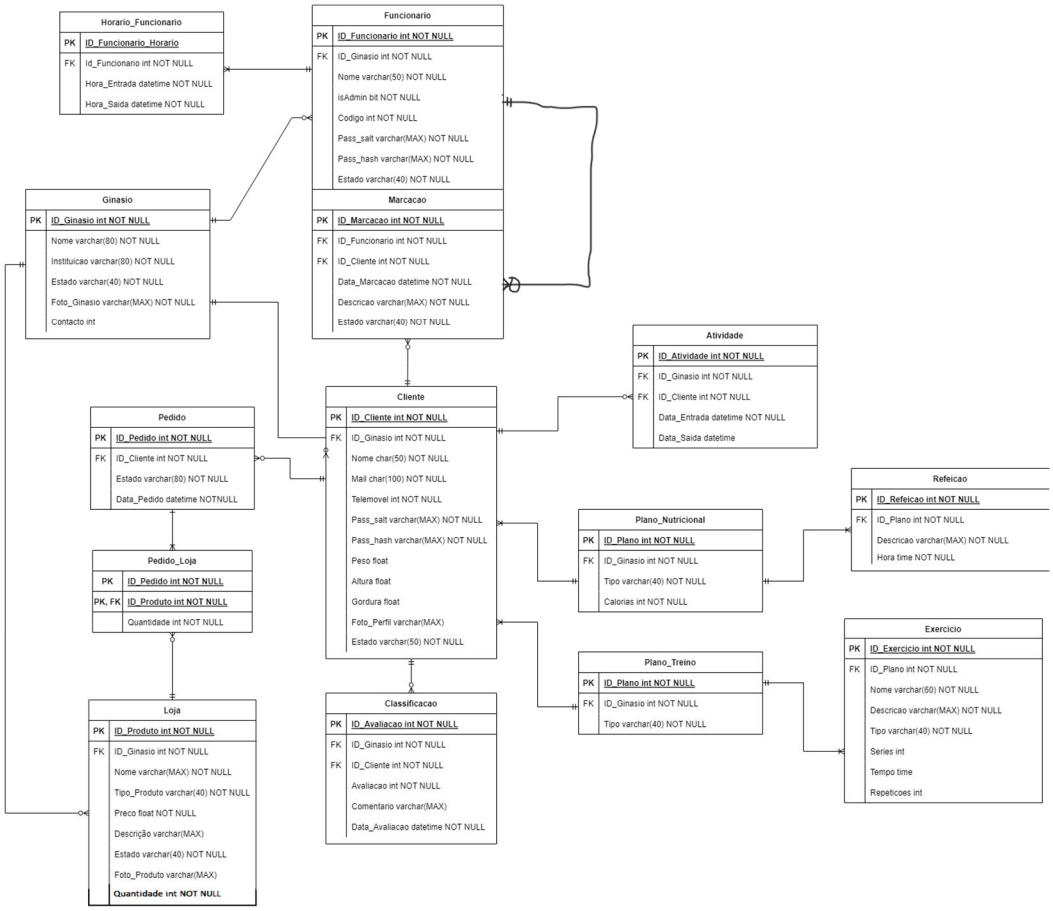


Figura 1 - Diagrama de Entidade-Relação

Como entidades principais este diagrama possuí:

- **Cliente** – dados de um cliente que está a utilizar a aplicação;
- **Funcionário** – dados de um funcionário do ginásio em causa, possuí o atributo "isAdmin" para determinar se este tem como role Gerente ou não;
- **Ginásio** – dados do ginásio em causa, entidade criada de forma que o projeto, mais tarde, tenha suporte para várias instituições académicas
- **Loja** – possui dados de todos os produtos disponíveis e indisponíveis na loja de cada ginásio
- **Atividade** – entidade criada com o propósito de analisar as entradas e saídas de cada cliente no ginásio (recebe informação do Arduíno)

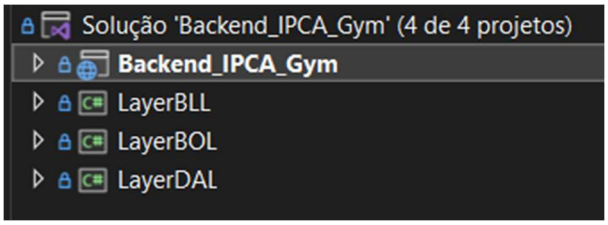
De forma que fosse possível suportar alguns dados sobre outras funcionalidades, foram adicionadas as seguintes entidades:

- **Plano_Nutricional** e Refeição – entidades que possuem dados sobre diferentes refeições e seus horários, cada ginásio define o seu plano nutricional;
- **Plano_Treino** e Exercício – entidades que possuem dados sobre diferentes exercícios e suas descrições, cada ginásio define o seu plano de treino;
- Pedido e **Pedido_Loja** – entidade que possui dados de cada encomenda feita pelo utilizador na loja do ginásio no qual este está inscrito;
- **Horario_Funcionario** – regista o horário de cada funcionário, de forma a verificar a sua disponibilidade para as diferentes marcações;
- **Marcação** – possui a informação de todas as marcações marcadas pelo cliente com o funcionário, associadas a cada ginásio;
- **Classificação** – contém todas as avaliações feitas pelos clientes a cada ginásio.

4. Código

4.1. Programação por Camadas

Neste projeto, como forma de organizar o nosso código, decidimos programar em 4 camadas, isto para que que o código fique mais organizado, com melhor performance e mais seguro. A nós permite-nos também detetar anomalias e corrigir problemas de forma mais simples e direta, tudo isto porque é possível substituir partes das camadas (ou a camada toda) sem que o sistema fique todo ele comprometido.



1ª Camada - Backend API
2ª Camada - BLL
3ª Camada - BOL
4ª Camada - DAL

Figura 2 - Camadas

4.1.1. Backend_IPCA_Gym

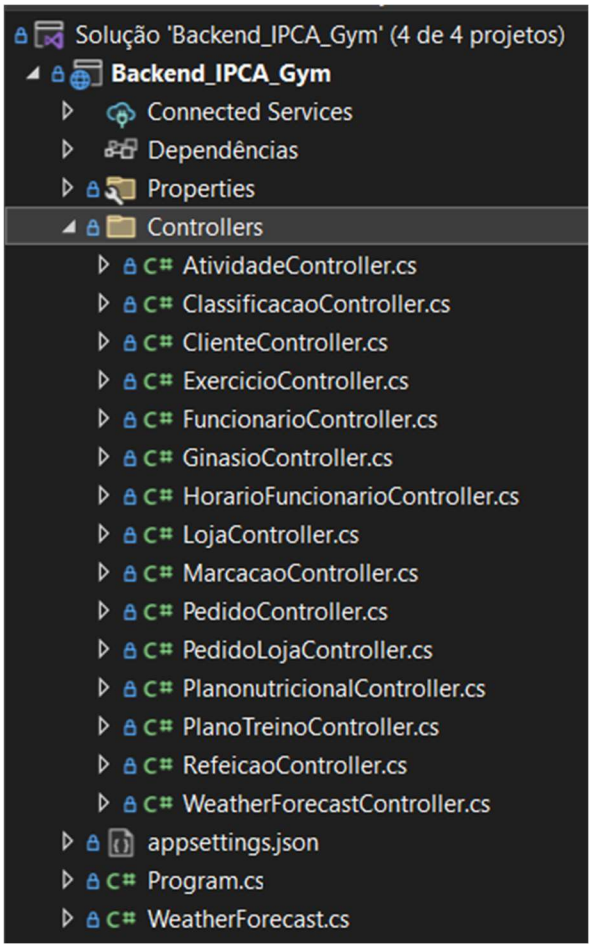


Figura 3 - Lista dos controladores

Na camada "Backend_IPCA_Gym" estão os controllers respetivos para todas as entidades do nosso sistema. Esta camada utiliza funções logic, sendo estas funções provindas da camada DAL.
Aqui é onde são executadas as chamadas à API. Também é nesta camada que está o nosso main, que é chamado quando executamos o nosso sistema.

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

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

Como referido na página anterior, em cada controlador corresponde a cada parte do nosso projeto e é onde são chamadas as funções que irão fazer a conexão com a bases de dados. Dentro dos controladores temos que colocar (antes de executarmos a chamada), o tipo de request que pretendemos fazer (httpget, httppost,hettpdelete…).

- 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. LayerBLL

"BLL" significa "Business Logic Layer" e é outro termo comum usado no desenvolvimento de software para se referir a uma camada na arquitetura de uma aplicação. A camada BLL normalmente é responsável por implementar a lógica de negócios. Isso pode incluir tarefas como validar a entrada do utilizador, realizar cálculos e interagir com a camada de acesso a dados (DAL) para recuperar e armazenar dados e muito mais.
No C#, a camada de lógica de negócios foi implementada como um conjunto de classes que contém os métodos que executam a lógica de negócios da aplicação. Esses métodos são chamados pela camada de apresentação (como uma interface de utilizador) ou por outros componentes na aplicação para executar determinadas tarefas.

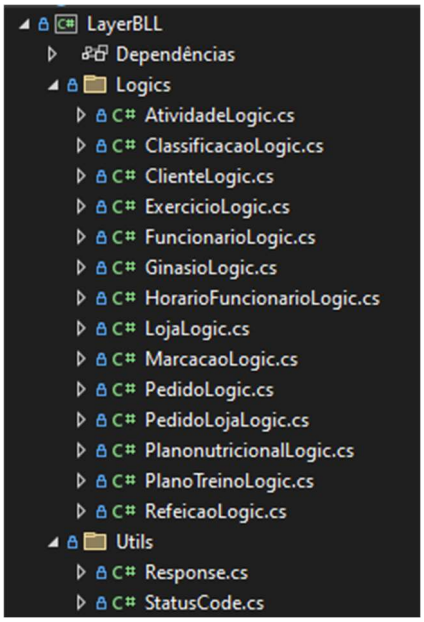


Figura 10 - Camada BLL

- Na parte *Logics* temos as funções logic (funções referidas anteriormente chamadas na camada Backend.
- Na parte *Utils* temos o código que nos dá informação request está ou não a funcionar 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

Apresento agora um exemplo, 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. LayerBOL

A camada "BOL" significa "Business Object Layer" e é um termo comum usado no desenvolvimento de software referir-se a uma camada que é responsável por representar entidades de negócios e os seus relacionamentos.
O "BOL" normalmente fica entre a camada de apresentação (como uma interface de utilizador) e a camada de acesso a dados (que é responsável pela comunicação com a base de dados ou outro armazenamento de dados).
No C#, a camada do objeto de negócios foi também implementada como um conjunto de classes respetivos a cada entidade do negócio e os relacionamentos entre elas. Nestas classes podemos observar também as propriedades que correspondem aos atributos das entidades de negócios e métodos que executam a lógica de negócios.

Em baixo, e com o código já devidamente comentado, podemos observar as propriedades que a entidade Ginásio tem e estão a ser chamados, ao lado da figura podemos também observar na base de dados do projeto que os campos são os mesmos da tabela.

```
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

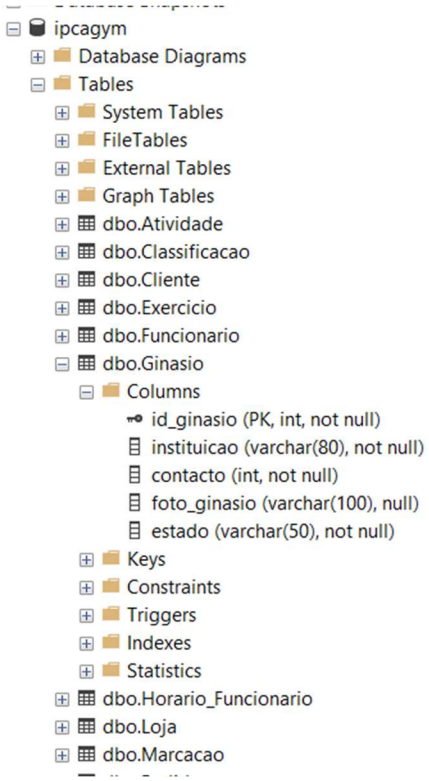
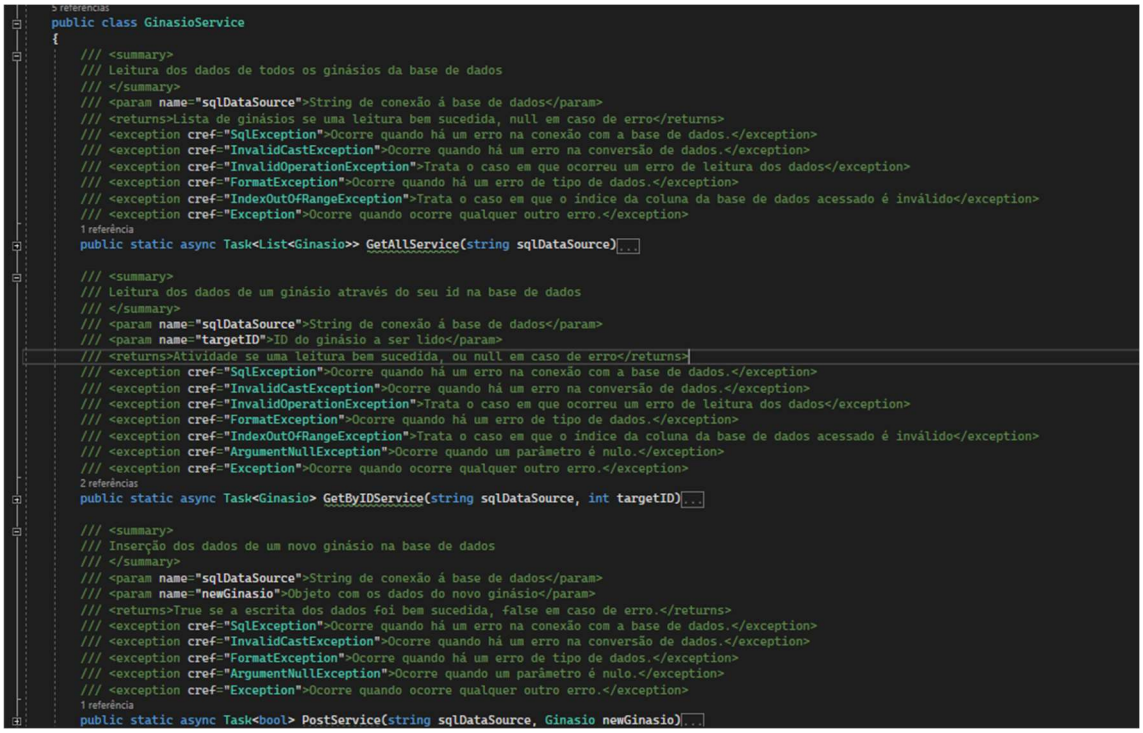


Figura 17 - Propriedades Ginásio DB

4.1.4. LayerDAL

"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 ginásio, construímos 2 funções diferentes, uma para listar todos os ginásios existentes e outra para apresentar apenas um ginásio específico através do seu ID.

```
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 - Ginasio Service – GetAll

```
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 - Ginasio 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.

Quanto ao método Post (inserção de valores na base de dados), podemos observar na query que temos a necessidade de colocar todos os campos que vamos adicionar a uma entidade especifica. Após definidos os campos basta apenas fazer a inserção dos campos através de comandos predefinidos no C#.

```
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 23 - Ginasio Service - Post

```
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 22 - Ginasio Service - Patch

Na imagem acima vemos como está implementada a função do Patch (alteração de dados relativos a uma entidade). Começamos por criar um objeto que irá ficar temporariamente a ser utilizado, esse objeto é atribuído a um específico através do seu ID onde serão alterados os dados.
Após a alteração de dados é feita a comunicação com a base de dados e alterados os dados correspondentes ao mesmo ID.

```
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 - Ginasio Service - Delete

Para a remoção de um registo relativo a uma entidade estamos a utilizar apenas o ID.

5. Dependências

Tendo em conta que este projeto está dividido em camadas, foi necessário atribuir as dependências a cada camada.

A atribuição de dependências foi a seguinte:

- 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)

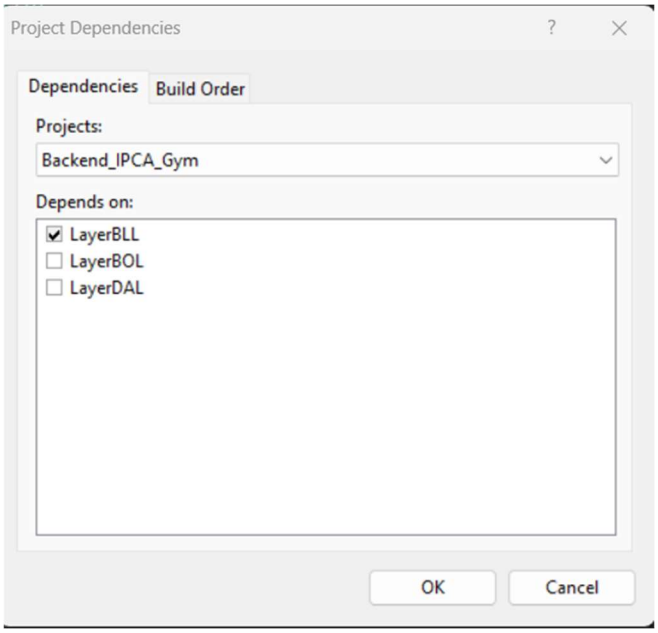


Figura 25 - Dependêcia API Layer

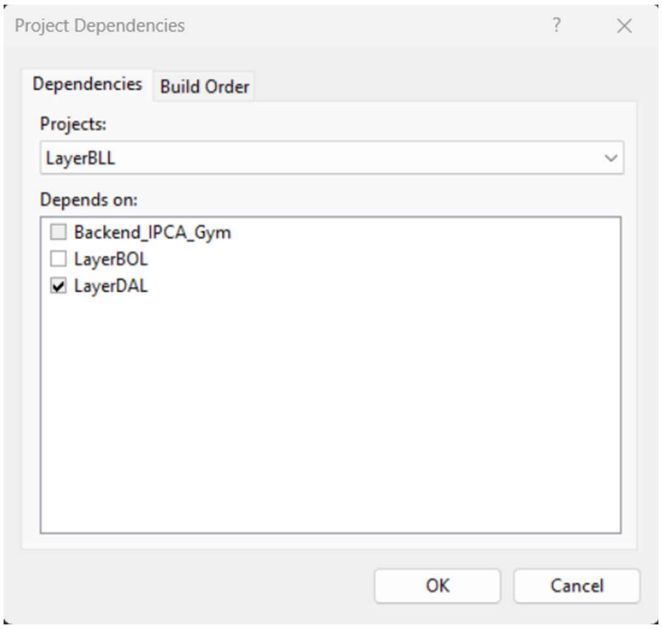


Figura 26 - Dependêcia BLL

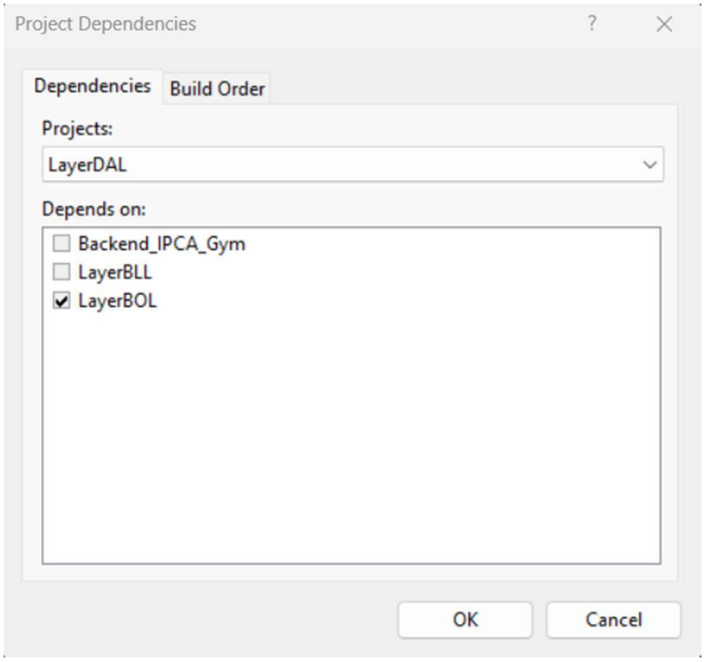


Figura 27 - Dependêcia BOL

6. Conclusão

A primeira etapa deste progresso atual do projeto foi possível com o recurso a ferramentas de suporte, com uma distribuição por tarefas e gestão do tempo semanal, foi também possível pelo companheirismo de todos os membros que se cumpriram e entre ajudaram. Com esta entrega foi possível fazer uma reflexão do estado atual do projeto e aspetos a serem melhorados.

Apesar de não estar mencionado no trabalho, o grupo encontra-se de momento a fazer implementação de **JWT Bearer Tokens**, bem como a autenticação que vem com as mesmas.

7. 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