Programação Orientada a Objetos

Relatório de Trabalho Prático

Arthur Fellipe Cerqueira Gomes - 24200

Júlia Dória Rodrigues - 24204

Engenharia de Sistemas Informáticos

Janeiro de 2024

ESCOLA
SUPERIOR
DE TECNOLOGIA
IPCA

Afirmo por minha honra que não recebi qualquer apoio não autorizado na realização deste trabalho prático. Afirmo igualmente que não copiei qualquer material de livro, artigo, documento web ou de qualquer outra fonte exceto onde a origem estiver expressamente citada.

Arthur Fellipe Cerqueira Gomes - 24200

Júlia Dória Rodrigues - 24204

Índice

INTRODUÇÃO 5

ESTRUTURA DO CÓDIGO 6

Classes e Subclasses 6

1. Program.cs 6
2. Utilizador.cs 7
3. Morador.cs 10
4. Gestor.cs e Funcionario.cs 11
5. Quarto.cs 13
6. Reserva.cs 14
7. Servico.cs 16
8. ControladorUtilizador.cs 17
9. ControladorMorador.cs 20
10. ControladorQuarto.cs 22
11. ControladorGestor.cs e ControladorFuncionario.cs 24
12. ControladorReserva.cs 26
13. ImenuUtilizador.cs 28
14. MenuInicial.cs 30
15. ViewReserva.cs 32

DOCUMENTAÇÃO (DOCFX) 36

GITHUB 36

Lista de Figuras

Figura 1 - Program.cs 7
Figura 2 - Classe Utilizador 8
Figura 3 - Construtor Utilizador 9
Figura 4 - Get/Set Utilizador 9
Figura 5 - Manipulação JSON Utilizador 10
Figura 6 - SubClasse Morador 10
Figura 7 - Construtor Morador 10
Figura 8 - Get/Set Morador 11
Figura 9 - Manipulação JSON do Morador.cs 11
Figura 10 - Construtor Gestor 12
Figura 11 - Método para salvar lista de gestores no JSON 12
Figura 12 - Método para carregar a lista de gestores do ficheiro JSON 13
Figura 13 - Classe Quarto 14
Figura 14 - Construtor Quarto 14
Figura 15 - Get/Set Quarto 14
Figura 16 - Classe Reserva 15
Figura 17 - Construtor Reserva 15
Figura 18 - Manipulação JSON da Reserva.cs 16
Figura 19 - Método CriarFicheiroJson da classe Reserva.cs 17
Figura 20 - ControladorUtilizador.cs 17
Figura 21 - Region Imprimir Uitlizador 18
Figura 22 - Region Autenticar Utilizador 19
Figura 23 - Métodos Auxiliares do ControladorUtilizador.cs 19
Figura 24 – Método para Criar Morador 20
Figura 25 - Método para Adicionar Morador à lista 20
Figura 26 - Método para imprimir lista de moradores 21

Figura 27 - Método para validar morador na lista 21
Figura 28 - Classe ControladorQuarto 22
Figura 29 - Método ImprimirListaDeQuartos 22
Figura 30 - Método ObterTipoQuartoAleatório 23
Figura 31 - Método ObterCapacidadePorTipoQuarto 23
Figura 32 - Método ObterPrecoBasePorTipoQuarto 23
Figura 33 - Método para calcular preço da renda 23
Figura 34 - Método ObterDisponibilidadeAleatoria 24
Figura 35 - Métodos para criar gestor e adicionar a lista 24
Figura 36 - Métodos para imprimir llista de gestores 25
Figura 37 - Método auxiliar para validar gestor na lista 25
Figura 38 - Método para criar reserva 26
Figura 39 - Método para buscar uma reserva 26
Figura 40 - Método para editar campos de uma reserva 27
Figura 41 - Método para excluir uma reserva existente 27
Figura 42 - Método ExibirMenuAposLogin 32
Figura 43 - Menu para Criar Reserva 33
Figura 44 - Menu para editar reserva 35

Introdução

Esta fase final do projeto apresenta os esforços dedicados ao desenvolvimento do sistema de gestão de reservas em residências estudantis, inicialmente concebido na fase anterior. Neste estágio, foi alcançado um marco significativo, com a implementação completa do código que abrange os métodos essenciais, menus interativos, interfaces intuitivas, adoção de uma arquitetura de camadas MVC (Model-View-Controller), e a integração eficaz de controladores.

O sistema opera de maneira eficiente, proporcionando a utilização de todas as funções pensadas para todos os usuários, sejam eles clientes, gestores ou funcionários. A plataforma, devidamente refinada, não apenas permite o cadastro de usuários, mas também oferece uma visão dos quartos disponíveis, destacando as características distintivas de cada acomodação, permite a realização de reservas, alterações, cancelamentos, assim como realização de cadastro de utilizadores, alteração de cadastro e exclusão.

Esta documentação não apenas detalha as classes e estruturas de dados implementadas, mas também fornece uma compreensão aprofundada da lógica por trás de cada funcionalidade, consolidando assim o desenvolvimento completo e bem-sucedido do sistema de gestão de reservas em residências estudantis com a programação orientada a objetos em C#.

Estrutura do Código

O código fonte da aplicação contém os seguintes ficheiros:

- Program.cs
- Utilizador.cs
- Morador.cs
- Funcionario.cs
- Gestor.cs
- Quarto.cs
- Reserva.cs
- Servico.cs
- ControladorUtilizador.cs
- ControladorMorador.cs
- ControladorQuarto.cs
- ControladorGestor.cs
- ControladorFuncionario.cs
- ControladorReserva
- ImenuUtilizador
- MenuInicial.cs
- ViewReserva.cs

Classes e Subclasses

1. Program.cs

A classe Program desempenha um papel crucial no funcionamento do programa, uma vez que contém o método *Main* que serve como ponto de entrada para a execução do sistema. Dentro deste método, uma série de operações são realizadas para configurar o ambiente inicial e permitir a interação do usuário.

Primeiramente, são iniciados os processos de criação de arquivos JSON para armazenar informações sobre utilizadores, moradores, gestores, funcionários e reservas. Isso é realizado chamando os métodos CriarFicheiroJson da classe Servico, que garantem a existência desses arquivos ou os criam vazios, se necessário.

Em seguida, o método estático CriarListaDeQuartos da classe ControladorQuarto é chamado para gerar uma lista de quartos com informações aleatórias. Essa lista é armazenada na variável Quarto.listaDeQuartos.

Por fim, é invocado o método estático ExibirMenuInicial da classe MenuInicial, proporcionando ao usuário uma interface interativa. Este método contém um loop que permite ao usuário escolher entre diversas opções, incluindo o registro de utilizadores, listagem de moradores, criação de reservas, entre outras. O loop continua até que o usuário opte por sair do sistema.

```
class Program
{
    0 referências
    static void Main(string[] args)
    {
        ControladorUtilizador controlador = new ControladorUtilizador();
        MenuInicial menuInicial = new MenuInicial(controlador);

        Servico.CriarFicheiroJson("utilizador.json");
        Servico.CriarFicheiroJson("morador.json");
        Servico.CriarFicheiroJson("gestor.json");
        Servico.CriarFicheiroJson("funcionario.json");
        Servico.CriarFicheiroJson("reserva.json");
        ControladorQuarto.CriarListaDeQuartos();

        menuInicial.ExibirMenuInicial();
    }
}
```

Figura 1 - Program.cs

Dessa forma, a classe Program desempenha um papel central na inicialização e execução do programa, garantindo a preparação adequada do ambiente antes de entrar no fluxo principal da aplicação.

2. Utilizador.cs

O código define uma classe pública chamada Utilizador, que representa todo e qualquer pessoa que faça registo na aplicação.

Dentro da classe, há uma lista pública estática chamada listaDeUtilizadores que armazena todos os utilizadores criados. As propriedades da classe, todas definidas como protegidas, incluem informações típicas de um utilizador, como ID, nome, email, etc. A propriedade ultimoId é auxiliar e usada para manter o último ID de utilizador.

O código utiliza o atributo [JsonProperty("NomePropriedade")] para mapear propriedades C# para chaves específicas durante a serialização ou desserialização de JSON.

```
public static List<Utilizador> listaDeUtilizadores = new List<Utilizador>(); // Variável que guarda a lista de utilizadores

[JsonProperty("UtiId")]
protected int UtiId;  // Propriedade que guarda o id do utilizador
[JsonProperty("NomeUti")]
protected string NomeUti; // Propriedade que guarda o nome do utilizador
[JsonProperty("Email")]
protected string Email; // Propriedade que guarda o email do utilizador
[JsonProperty("Password")]
protected string Password; // Propriedade que guarda a password do utilizador
[JsonProperty("DataNascimento")]
protected DateTime DataNascimento; // Propriedade que guarda a data de nascimento do utilizador
[JsonProperty("Morada")]
protected string Morada; // Propriedade que guarda a morada do utilizador
[JsonProperty("CodigoPostal")]
protected string CodigoPostal; // Propriedade que guarda o código postal do utilizador
[JsonProperty("Localidade")]
protected string Localidade; // Propriedade que guarda a localidade do utilizador
[JsonProperty("ContactoTelefone")]
protected string ContactoTelefone; // Propriedade que guarda o contacto telefónico do utilizador
[JsonProperty("DocIdentificacao")]
protected string DocIdentificacao; // Propriedade que guarda o documento de identificação do utilizador
[JsonProperty("TipoDocIdentificacao")]
protected string TipoDocIdentificacao; // Propriedade que guarda o tipo de documento de identificação do utilizador
[JsonProperty("IBAN")]
protected string IBAN; // Propriedade que guarda o IBAN do utilizador
[JsonProperty("TipoUtilizador")]
protected string TipoUtilizador; // Propriedade que guarda o tipo de utilizador
[JsonProperty("IsAtivo")]
protected bool IsAtivo; // Propriedade que indica se o utilizador está ativo ou não
[JsonProperty("DataRegisto")]
protected DateTime DataRegisto; // Propriedade que guarda a data de registo do utilizador

protected static int ultimoId = 0; // Variável que guarda o último id de utilizador
```

Figura 2 - Classe Utilizador

O construtor da classe inicializa um objeto Utilizador com as respetivas propriedades e define que alguns valores já serão inicializados com valores padrão. Nomeadamente, estabelece que a data de registo padrão é a data atual e que o utilizador inicia com estado inativo.

```
public Utilizador(int utiId,
                  string nomeUti,
                  string email,
                  string password,
                  DateTime dataNascimento,
                  string morada,
                  string codigoPostal,
                  string localidade,
                  string contactoTelefone,
                  string docIdentificacao,
                  string tipoDocIdentificacao,
                  string iban,
                  string tipoUtilizador,
                  bool isAtivo = false, // Por defeito, o utilizador não está ativo
                  DateTime dataRegisto = default(DateTime)) // Por defeito, a data de registo é a data atual

{
    UtiId = utiId;
    NomeUti = nomeUti;
    Email = email;
    Password = password;
    DataNascimento = dataNascimento;
    Morada = morada;
    CodigoPostal = codigoPostal;
    Localidade = localidade;
    ContactoTelefone = contactoTelefone;
    DocIdentificacao = docIdentificacao;
    TipoDocIdentificacao = tipoDocIdentificacao;
    IBAN = iban;
    TipoUtilizador = tipoUtilizador;
    IsAtivo = isAtivo;
    DataRegisto = (dataRegisto == default(DateTime)) ? DateTime.Now : dataRegisto; // Se a data de registo não for definida, assume a data atual
}
```

Figura 3 - Construtor Utilizador

Há também métodos Get e Set para cada propriedade, permitindo acesso e modificação desses atributos.

```
#region Getters e Setters
3 references
public int GetUtiId()
{
    return UtiId;
}

0 references
public void SetUtiId(int utiId)
{
    UtiId = utiId;
}
```

Figura 4 - Get/Set Utilizador

Por fim, foi criada também uma região com alguns métodos para salvar e manipular o arquivo JSON dos utilizadores. Os dois métodos listados abaixo, por exemplo, são utilizados para gerenciar a persistência de dados de utilizadores em formato JSON. O método CriarListaUtilizador cria uma lista vazia de utilizadores e a salva em um arquivo "utilizador.json", enquanto o método SalvarListaFicheiro serializa a lista, incluindo o último ID, e escreve no arquivo JSON especificado.

```
public void CriarListaUtilizador()
{
    List<Utilizador> listaDeUtilizadores = new List<Utilizador>();
    SalvarListaFicheiro("utilizador.json", listaDeUtilizadores);
}
/// <summary>
/// Método para salvar a lista de utilizadores no ficheiro JSON
/// </summary>
/// <param name="caminhoArquivo"></param>
/// <param name="listaUtilizadores"></param>
2 referências
public static void SalvarListaFicheiro(string caminhoArquivo, List<Utilizador> listaUtilizadores)
{
    string json = JsonConvert.SerializeObject(new { UltimoId = LerUltimoIdDoJson(caminhoArquivo),
        Utilizadores = listaUtilizadores }, Newtonsoft.Json.Formatting.Indented);
    File.WriteAllText(caminhoArquivo, json);
}
```

Figura 5 - Manipulação JSON Utilizador

3. Morador.cs

O código define uma subclasse chamada Morador, que herda da classe base Utilizador, e representa um morador de uma residência. A propriedade *listaDeMoradores* é uma lista estática que armazena todos os moradores criados. A propriedade *IsAdimplente*, específica da subclasse, indica se o morador está adimplente ou não.

```
public class Morador : Utilizador
{

    [JsonProperty("isAdimplente")]
    protected bool IsAdimplente; // Propriedade que indica se o morador está adimplente ou não
```

Figura 6 - SubClasse Morador

O construtor da classe Morador chama o construtor da classe base Utilizador, passando os parâmetros necessários e inicializando a propriedade específica *IsAdimplente* com valor verdadeiro por defeito, assim como *tipoUtilizador* passa a receber a string padrão “Morador”.

```
public Morador(
int utilId,
string nomeUti,
string email,
string password,
DateTime dataNascimento,
string morada,
string codigoPostal,
string localidade,
string contactoTelefone,
string docIdentificacao,
string tipoDocIdentificacao,
string iban,
bool isAtivo,
DateTime dataRegisto,
bool isAdimplente = true) // O morador é adimplente por defeito
: base(utilId, nomeUti, email, password, dataNascimento, morada, codigoPostal, localidade, contactoTelefone, docIdentificacao, tipoDocIdentificacao, iban, "Morador", isAtivo, dataRegisto)
{
    IsAdimplente = isAdimplente;
}
#endregion
```

Figura 7 - Construtor Morador

Há métodos Get e Set para obter e modificar a propriedade *IsAdimplente*.

```
#region Getters e Setters
2 referências
public bool GetIsAdimplente()
{
    return IsAdimplente;
}
0 referências
public void SetAdimplente(bool isAdimplente)
{
    IsAdimplente = isAdimplente;
}
#endregion
```

Figura 8 - Get/Set Morador

Há ainda dois métodos relacionados à persistência de dados de moradores em formato JSON. O método SalvarListaFicheiro serializa uma lista de moradores e a escreve em um arquivo específico, enquanto o método CarregarListaDeMoradores lê um arquivo JSON, verifica se contém dados válidos e retorna a lista de moradores correspondente.

```
public static void SalvarListaFicheiro(string caminhoArquivo, List<Morador> listaMoradores)
{
    // Serializar lista de moradores
    string json = JsonConvert.SerializeObject(listaMoradores, Newtonsoft.Json.Formatting.Indented);
    File.WriteAllText(caminhoArquivo, json); // Escrever no ficheiro
}

/// <summary> Método para carregar a lista de moradores do ficheiro JSON
2 referências
public static List<Morador> CarregarListaDeMoradores(string caminhoArquivo)
{
    if (File.Exists(caminhoArquivo)) // Verificar se o ficheiro existe
    {
        string json = File.ReadAllText(caminhoArquivo); // Ler o ficheiro

        if (!string.IsNullOrEmpty(json)) // Verificar se o JSON não está vazio
        {
            return JsonConvert.DeserializeObject<List<Morador>>(json);
        }
    }

    return new List<Morador>(); // Se o ficheiro não existir ou estiver vazio, retorna uma lista vazia
}
```

Figura 9 - Manipulação JSON do Morador.cs

4. Gestor.cs e Funcionario.cs

A classe Gestor é uma subclasse de Utilizador, herdando todos os atributos e métodos dessa classe base. Ela representa um tipo específico de utilizador no sistema, identificado como gestor.

O construtor da classe Gestor é responsável por inicializar os atributos específicos de um gestor. Ele recebe parâmetros como utiId, nomeUti, email, password, dataNascimento, entre outros. Esses valores são então repassados ao construtor da classe base Utilizador por meio da palavra-chave base. Essa abordagem é adotada para garantir que todos os atributos da classe base sejam devidamente inicializados.

```
public class Gestor : Utilizador
{
    1 referência
    public Gestor(
        int utiId,
        string nomeUti,
        string email,
        string password,
        DateTime dataNascimento,
        string morada,
        string codigoPostal,
        string localidade,
        string contactoTelefone,
        string docIdentificacao,
        string tipoDocIdentificacao,
        string iban,
        bool isAtivo,
        DateTime dataRegisto)
        : base(utiId, nomeUti, email, password, dataNascimento, morada, codigoPostal, localidade,
              contactoTelefone, docIdentificacao, tipoDocIdentificacao, iban, "Gestor", isAtivo, dataRegisto)
    {

    }
```

Figura 10 - Construtor Gestor

Além disso, a classe Gestor implementa métodos relacionados à persistência em formato JSON. O método SalvarListaFicheiro permite salvar uma lista de gestores em um arquivo JSON. Ele recebe o caminho do arquivo e a lista de gestores como parâmetros, serializa a lista para o formato JSON com formatação indentada e escreve o conteúdo no arquivo.

```
public static void SalvarListaFicheiro(string caminhoArquivo, List<Gestor> listaGestores)
{
    string json = JsonConvert.SerializeObject(listaGestores, Newtonsoft.Json.Formatting.Indented);
    File.WriteAllText(caminhoArquivo, json); // Escrever no ficheiro
}
```

Figura 11 - Método para salvar lista de gestores no JSON

O método CarregarListaDeGestores tem a função oposta: carregar uma lista de gestores a partir de um arquivo JSON. Ele verifica se o arquivo existe, lê o conteúdo do arquivo e realiza a desserialização do JSON para obter a lista de gestores. Se o arquivo não existir ou estiver vazio, o método retorna uma lista vazia.

```
public static List<Gestor> CarregarListaDeGestores(string caminhoArquivo)
{
    if (File.Exists(caminhoArquivo)) // Verificar se o ficheiro existe
    {
        string json = File.ReadAllText(caminhoArquivo); // Ler o ficheiro

        if (!string.IsNullOrEmpty(json)) // Verificar se o JSON não está vazio
        {
            return JsonConvert.DeserializeObject<List<Gestor>>(json);
        }
    }

    return new List<Gestor>(); // Se o ficheiro não existir ou estiver vazio, retorna uma lista vazia
}
```

Figura 12 - Método para carregar a lista de gestores do ficheiro JSON

Para a classe Funcionario é possível observar que ela segue uma estrutura semelhante à classe Gestor. No entanto, para evitar redundância na explicação, é importante ressaltar que a classe Funcionario compartilha as mesmas características fundamentais da classe Gestor, uma vez que ambas são subclasses de Utilizador. Portanto, os métodos e funcionalidades associados à persistência em JSON, como SalvarListaFicheiro e CarregarListaDeFuncionarios, podem ser encontrados de forma análoga na classe Funcionario, sem a necessidade de uma descrição detalhada.

5. Quarto.cs

O código define uma classe chamada Quarto, que representa um quarto em uma residência. A propriedade listaDeQuartos é uma lista estática que armazena todos os quartos criados.

A classe possui propriedades protegidas para armazenar informações sobre o quarto, como QuartoId (indicando o ID do quarto), TipoQuarto (indicando o tipo de quarto), Andar (indicando o andar onde o quarto está localizado), Capacidade (indicando a capacidade de pessoas do quarto), PrecoRenda (indicando o preço da renda do quarto) e Disponibilidade (indicando se o quarto está disponível).

O código utiliza o atributo [JsonProperty("NomePropriedade")] para mapear propriedades C# para chaves específicas durante a serialização ou desserialização de JSON.

```
public class Quarto
{
    public static List<Quarto> listaDeQuartos = new List<Quarto>(); // Variável que guarda a lista de quartos da residência

    [JsonProperty("quartoId")]
    protected int QuartoId; // Propriedade que indica o id do quarto
    [JsonProperty("tipoQuarto")]
    protected string TipoQuarto; // Propriedade que indica o tipo de quarto (individual, duplo, triplo, etc.)
    [JsonProperty("andar")]
    protected int Andar; // Propriedade que indica o andar onde se localiza o quarto
    [JsonProperty("capacidade")]
    protected int Capacidade; // Propriedade que indica a capacidade de pessoas do quarto
    [JsonProperty("precoRenda")]
    protected float PrecoRenda; // Propriedade que indica o preço da renda do quarto
    [JsonProperty("disponibilidade")]
    protected bool Disponibilidade; // Propriedade que indica se o quarto está disponível ou não
```

Figura 13 - Classe Quarto

O construtor da classe Quarto é responsável por inicializar essas propriedades com os valores passados como parâmetros.

```
public Quarto(int quartoId, string tipoQuarto, int andar, int capacidade, float precoRenda, bool disponibilidade)
{
    QuartoId = quartoId;
    TipoQuarto = tipoQuarto;
    Andar = andar;
    Capacidade = capacidade;
    PrecoRenda = precoRenda;
    Disponibilidade = disponibilidade;
}
```

Figura 14 - Construtor Quarto

Métodos Get e Set são fornecidos para aceder e modificar as propriedades do quarto.

```
#region Getters e Setters
1 reference
public int GetQuartoId()
{
    return QuartoId;
}
0 references
private void SetQuartoId(int quartoId)
{
    QuartoId = quartoId;
}
```

Figura 15 - Get/Set Quarto

6. Reserva.cs

O código apresenta uma classe chamada Reserva, que representa uma reserva de um quarto em uma residência. A classe possui propriedades públicas para armazenar informações sobre a reserva, como ReservaId (indicando o ID da reserva), QuartoId (indicando o ID do quarto reservado), UtiId (indicando o ID do utilizador que fez a reserva), DataEntrada (indicando a

data de entrada na reserva), DataSaida (indicando a data de saída da reserva), PrecoCaucao (indicando o preço da caução da reserva), IsAtivo (indicando se a reserva está ativa ou não) e DataReserva (indicando a data da reserva).

O código utiliza o atributo [JsonProperty("NomePropriedade")] para mapear propriedades C# para chaves específicas durante a serialização ou desserialização de JSON.

```
public class Reserva
{
    [JsonProperty("reservaId")]
    1 reference
    public int ReservaId { get; set; } // Propriedade que indica o id da reserva
    [JsonProperty("quartoId")]
    1 reference
    public int QuartoId { get; set; } // Propriedade que indica o id do quarto reservado
    [JsonProperty("utiId")]
    1 reference
    public int UtiId { get; set; } // Propriedade que indica o id do utilizador que fez a reserva
    [JsonProperty("dataEntrada")]
    1 reference
    public DateTime DataEntrada { get; set; } // Propriedade que indica a data de entrada na reserva
    [JsonProperty("dataSaida")]
    1 reference
    public DateTime DataSaida { get; set; } // Propriedade que indica a data de saída da reserva
    [JsonProperty("precoCaucao")]
    1 reference
    public float PrecoCaucao { get; set; } // Propriedade que indica o preço da caução da reserva
    [JsonProperty("isAtivo")]
    1 reference
    public bool IsAtivo { get; set; } // Propriedade que indica se a reserva está ativa ou não
    [JsonProperty("dataReserva")]
    1 reference
    public DateTime DataReserva { get; set; } // Propriedade que indica a data da reserva
```

Figura 16 - Classe Reserva

Os métodos de obtenção e modificação das propriedades são criados de forma automática, com a sintaxe { get; set; }.

O construtor da classe Reserva é responsável por inicializar essas propriedades com os valores passados como parâmetros. Ele também define um valor padrão para DataReserva como a data atual, caso não seja especificada.

```
#region Construtor
0 references
public Reserva(
    int reservaId,
    int quartoId,
    int utiId,
    DateTime dataEntrada,
    DateTime dataSaida,
    float precoCaucao,
    bool isAtivo,
    DateTime dataReserva = default(DateTime)) // A data da reserva é a data atual por defeito
{
    ReservaId = reservaId;
    QuartoId = quartoId;
    UtiId = utiId;
    DataEntrada = dataEntrada;
    DataSaida = dataSaida;
    PrecoCaucao = precoCaucao;
    IsAtivo = isAtivo;
    DataReserva = (dataReserva == default(DateTime)) ? DateTime.Now : dataReserva; // Se a data da reserva não for especificada, é a data atual
}
#endregion
```

Figura 17 - Construtor Reserva

Além das funcionalidades já apresentadas, a classe Reserva.cs inclui métodos dedicados à manipulação de dados em formato JSON. Dois desses métodos notáveis são o LerUltimoIdReserva e o AtualizarUltimoIdNoJson. O primeiro, LerUltimoIdReserva, é responsável por ler o último ID de reserva a partir de um arquivo JSON, proporcionando uma referência para a criação subsequente de novas reservas. Caso o arquivo ou o conteúdo não existam, o método retorna o valor padrão zero. Por sua vez, o segundo método, AtualizarUltimoIdNoJson, recebe um novo ID como parâmetro e o utiliza para atualizar o último ID no arquivo JSON correspondente, garantindo a consistência e integridade dos dados de reservas no sistema.

```
public static int LerUltimoIdReserva(string caminhoArquivo)
{
    dynamic jsonData = JsonConvert.DeserializeObject(File.ReadAllText(caminhoArquivo));

    if (jsonData != null && jsonData.UltimoIdReserva != null)
    {
        return (int)jsonData.UltimoIdReserva;
    }

    return 0; // Valor padrão se o ficheiro ou conteúdo não existir
}

/// <summary>
/// Método que atualiza o último id de reserva no ficheiro JSON
/// </summary>
/// <param name="novoUltimoId"></param>
1 referência
public static void AtualizarUltimoIdNoJson(int novoUltimoId, string caminhoArquivo)
{
    List<Reserva> listaExistente = CarregarListaDeReservas(caminhoArquivo);

    string json = JsonConvert.SerializeObject(new { UltimoIdReserva = novoUltimoId,
        Reservas = listaExistente }, Newtonsoft.Json.Formatting.Indented);
    File.WriteAllText(caminhoArquivo, json);
}
```

Figura 18 - Manipulação JSON da Reserva.cs

7. Servico.cs

A classe Servico.cs desempenha um papel crucial ao oferecer funcionalidades utilitárias comuns no contexto do programa. O método CriarFicheiroJson se encarrega de verificar a existência de um arquivo JSON no caminho especificado e, caso não exista, cria um novo arquivo vazio.

```
public static void CriarFicheiroJson(string caminhoArquivo)
{
    if (!File.Exists(caminhoArquivo))
    {
        File.Create(caminhoArquivo).Close();
    }
}
```

Figura 19 - Método CriarFicheiroJson da classe Reserva.cs

Além disso, a classe disponibiliza uma série de métodos de leitura de dados, como LerString, LerInteiro, LerData, LerFloat e LerBooleano. Estes métodos são projetados para simplificar e padronizar a entrada de dados fornecida pelo usuário, promovendo a consistência e evitando entradas inválidas. Essa abordagem facilita a interação dos usuários com o sistema, tornando a entrada de dados mais intuitiva e menos suscetível a erros, contribuindo para a eficiência e usabilidade geral do programa.

8. ControladorUtilizador.cs

A classe ControladorUtilizador tem um papel central no gerenciamento de diferentes tipos de utilizadores no sistema. Ela possui instâncias dos controladores ControladorMorador, ControladorGestor e ControladorFuncionario.

Os métodos SetControladorMorador, SetControladorGestor e SetControladorFuncionario são responsáveis por instanciar objetos desses controladores, possibilitando a interação direta com os dados específicos de moradores, gestores e funcionários.

```
public class ControladorUtilizador
{
    private ControladorMorador controladorMorador;
    private ControladorGestor controladorGestor;
    private ControladorFuncionario controladorFuncionario;
    /// <summary> Instancia um objeto de ControladorMorador
    0 referências
    public void SetControladorMorador(ControladorMorador controladorMorador)
    {
        this.controladorMorador = controladorMorador;
    }
    /// <summary> Instancia um objeto de ControladorFuncionario
    0 referências
    public void SetControladorGestor(ControladorGestor controladorGestor)
    {
        this.controladorGestor = controladorGestor;
    }
    /// <summary> Instancia um objeto de ControladorFuncionario
    0 referências
    public void SetControladorFuncionario(ControladorFuncionario controladorFuncionario)
    {
        this.controladorFuncionario = controladorFuncionario;
    }
```

Figura 20 - ControladorUtilizador.cs

Essa estrutura modular permite uma gestão eficiente dos diferentes tipos de utilizadores no sistema, permitindo a coordenação adequada entre as entidades envolvidas. Ao associar cada controlador com sua respectiva classe de utilizador, o ControladorUtilizador facilita a implementação de funcionalidades específicas para cada tipo de utilizador no contexto do sistema de gestão de reservas em residências estudantis.

Em seguida, o ControladorUtilizador.cs apresenta o CRUD do utilizador, com as funções para criar, buscar, editar e excluir um utilizador. Cada uma dessas funções possuem seus métodos próprios, com menus para que o utilizador possa interagir.

Por fim, há ainda três regiões distintas de métodos nesta classe. A primeira diz respeito as Funcionalidades de Impressão e Exibição de Utilizadores, onde o conjunto de métodos ImprimirListaDeUtilizadores, ExibirDetalhesListaUtilizador e ExibirDetalhesUtilizador desempenha o papel de visualização das informações dos utilizadores no sistema. O método ImprimirListaDeUtilizadores inicia o processo carregando a lista de utilizadores a partir de um arquivo JSON. Em seguida, invoca o método ExibirDetalhesListaUtilizador, que formata e apresenta de forma legível as informações contidas na lista. Além disso, o método ExibirDetalhesUtilizador é responsável por mostrar os detalhes de um único utilizador quando necessário.

```
public void ImprimirListaDeUtilizadores()
{

    List<Utilizador> listaDeUtilizadoresAtual =
        Utilizador.CarregarListaDeUtilizadores("utilizador.json");

    ExibirDetalhesListaUtilizador(listaDeUtilizadoresAtual);

}

/// <summary> Método para exibir os detalhes de uma lista de utilizadores
2 referências
public void ExibirDetalhesListaUtilizador(List<Utilizador> listaDeUtilizadoresAtual)
{
    if (listaDeUtilizadoresAtual == null)
    {
        Console.WriteLine("Não há utilizadores registados");
    }
    else
    {
        Console.WriteLine("Lista de Utilizadores");
        Console.WriteLine("-------------------------------------------");
        foreach (Utilizador utilizador in listaDeUtilizadoresAtual)
        {
            Console.WriteLine($"ID: {utilizador.GetUtiId()}, Nome: {utilizador.GetNomeUti()}, Email: {u
        }
```

Figura 21 - Region Imprimir Uitlizador

O bloco de código sob a região "Autenticação" inclui métodos relacionados à autenticação de utilizadores, notavelmente AutenticarUtilizador e Login. O método AutenticarUtilizador verifica se as credenciais fornecidas correspondem a algum utilizador na lista carregada do arquivo JSON. Por sua vez, o método Login interage com o utilizador, solicitando email e senha, e retorna o utilizador autenticado ou nulo, dependendo do sucesso da autenticação.

```
public static Utilizador AutenticarUtilizador(string email, string senha)
{
    // Carregar a lista de utilizadores do arquivo JSON
    List<Utilizador> listaDeUtilizadores = Utilizador.CarregarListaDeUtilizadores("utilizador.json");

    // Verificar se as credenciais correspondem a algum usuário na lista
    return listaDeUtilizadores.FirstOrDefault(u => u.GetEmail() == email && u.GetPassword() == senha);
}

1 referência
public static Utilizador Login()
{

    string email = Servico.LerString("Email: ");
    string senha = Servico.LerString("Senha: ");

    Utilizador utilizador = AutenticarUtilizador(email, senha);

    if (utilizador != null)
    {
        Console.WriteLine($"Bem-vindo, {utilizador.GetNomeUti()}!");
        return utilizador;
    }

    else
    {
        return null;
    }
```

Figura 22 - Region Autenticar Utilizador

Por fim, o bloco sob "Métodos Auxiliares" oferece funcionalidades de validação. O método ValidarUtilizador impede a duplicidade de utilizadores verificando se um novo utilizador já existe na lista. Por outro lado, ValidarTipoUtilizador valida se o tipo de utilizador é válido, sendo "morador", "funcionario" ou "gestor". A enumeração TipoCampo define os campos que podem ser editados, contribuindo para a consistência dos dados.

```
public bool ValidarUtilizador(Utilizador novoUtilizador, List<Utilizador> listaExistente)
{
    // Verificar se o utilizador já existe na lista
    if (listaExistente.Exists(u =>
                novoUtilizador.GetDocIdentificacao() == u.GetDocIdentificacao() &&
                                novoUtilizador.GetTipoDocIdentificacao() ==
                                u.GetTipoDocIdentificacao()))
    {
        Console.WriteLine("O utilizador já existe na lista.");
        return false;
    }
    else
    {
        return true;
    }
}

/// <summary> Método de validação para o tipo de utilizador
6 referências
public bool ValidarTipoUtilizador(string tipoUtilizador)
{
    return tipoUtilizador.Equals("morador", StringComparison.OrdinalIgnoreCase) ||
           tipoUtilizador.Equals("funcionario", StringComparison.OrdinalIgnoreCase) ||
           tipoUtilizador.Equals("gestor", StringComparison.OrdinalIgnoreCase);
}
```

Figura 23 - Métodos Auxiliares do ControladorUtilizador.cs

Esses conjuntos de funcionalidades proporcionam uma base sólida para a manipulação, exibição e validação dos utilizadores no sistema, garantindo um ambiente coeso e eficiente.

9. ControladorMorador.cs

A classe ControladorMorador atua como o controlador central para operações relacionadas aos moradores no sistema. Esta classe integra métodos para criar um novo morador, adicionar moradores à lista, imprimir e exibir detalhes da lista de moradores. A criação de um morador é feita com base nas informações de um utilizador existente, garantindo consistência nos dados.

```
public Morador CriarMorador(Utilizador utilizador)
{
    Morador morador = new Morador(
                utilizador.GetUtiId(),
                utilizador.GetNomeUti(),
                utilizador.GetEmail(),
                utilizador.GetPassword(),
                utilizador.GetDataNascimento(),
                utilizador.GetMorada(),
                utilizador.GetCodigoPostal(),
                utilizador.GetLocalidade(),
                utilizador.GetContactoTelefone(),
                utilizador.GetDocIdentificacao(),
                utilizador.GetTipoDocIdentificacao(),
                utilizador.GetIBAN(),
                utilizador.GetIsAtivo(),
                utilizador.GetDataRegisto());
    AdicionarMorador(morador); // Adicionar morador à lista de moradores
    return morador;

}
```

Figura 24 – Método para Criar Morador

O método AdicionarMorador valida a inclusão de um novo morador, evitando duplicidades na lista existente, e, se a validação for bem-sucedida, adiciona o morador à lista e salva as alterações em um arquivo JSON.

```
public bool AdicionarMorador(Morador novoMorador)
{
    // Carregar a lista existente do arquivo
    List<Morador> listaExistente = Morador.CarregarListaDeMoradores("morador.json");

    // Verificar se o morador já existe na lista através do documento de identificação
    if (!ValidarMorador(novoMorador, listaExistente))
    {
        return false;
    }
    else
    {
        listaExistente.Add(novoMorador); // Adicionar morador à lista de moradores
        Morador.SalvarListaFicheiro("morador.json", listaExistente); // Salvar lista de moradores no ficheiro
        return true;
    }

}
```

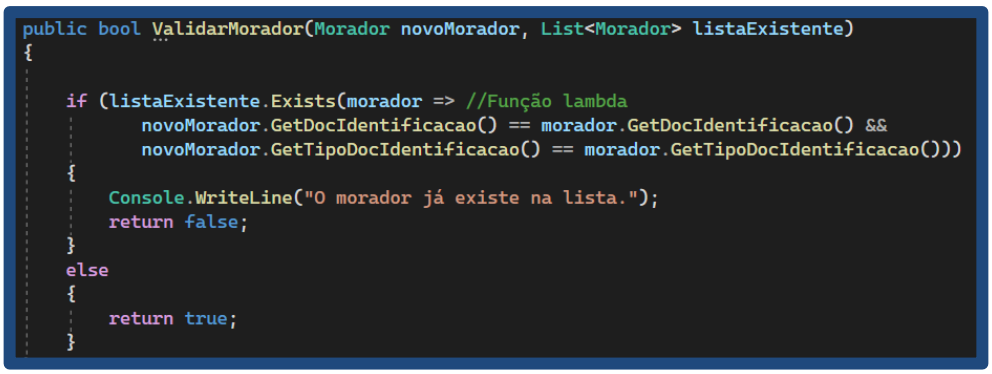
Figura 25 - Método para Adicionar Morador à lista

A visualização da lista de moradores é possível por meio dos métodos ImprimirListaDeMoradores e ExibirDetalhesListaMorador, que apresentam as informações de cada morador de forma organizada no console. Adicionalmente, o método ExibirDetalhesMorador permite visualizar os detalhes de um único morador.

```
public void ImprimirListaDeMoradores()
{
    List<Morador> listaDeMoradoresAtual = Morador.CarregarListaDeMoradores("morador.json"); // Carregar lista de m

    ExibirDetalhesListaMorador(listaDeMoradoresAtual);

}
 1 referência
public void ExibirDetalhesListaMorador(List<Morador> listaDeMoradoresAtual)
{
    if (listaDeMoradoresAtual == null)
    {
        Console.WriteLine("Não há moradores registados");
    }
    else
    {
        Console.WriteLine("Lista de Moradores");
        Console.WriteLine("----------------------------------------");
        foreach (Morador morador in listaDeMoradoresAtual)
        {
            Console.WriteLine($"ID: {morador.GetUtiId()}, Nome: {morador.GetNomeUti()}, " +
                $"Email: {morador.GetEmail()}, Data de Nascimento: {morador.GetDataNascimento():dd-MM-yyyy}, " +
                $"Morada: {morador.GetMorada()}, Código Postal: {morador.GetCodigoPostal()}, " +
                $"Localidade: {morador.GetLocalidade()}, Contacto Telefone: {morador.GetContactoTelefone()}, " +
                $"Documento de Identificação: {morador.GetDocIdentificacao()}, " +
                $"Tipo de Documento de Identificação: {morador.GetTipoDocIdentificacao()}, " +
                $"IBAN: {morador.GetIBAN()}, Tipo de Utilizador: {morador.GetTipoUtilizador()}, " +
                $"Ativo: {morador.GetIsAtivo()} Data de Registo: {morador.GetDataRegisto():dd-MM-yyyy}, " +
                $"Adimplente: {morador.GetIsAdimplente()}");
```

Figura 26 - Método para imprimir lista de moradores

A classe também conta com o método auxiliar `ValidarMorador`, responsável por evitar a inclusão de moradores duplicados na lista, verificando a existência de moradores com base no documento de identificação e tipo de documento.



```
public bool ValidarMorador(Morador novoMorador, List<Morador> listaExistente)
{

    if (listaExistente.Exists(morador => //Função lambda
            novoMorador.GetDocIdentificacao() == morador.GetDocIdentificacao() &&
            novoMorador.GetTipoDocIdentificacao() == morador.GetTipoDocIdentificacao()))
    {
        Console.WriteLine("O morador já existe na lista.");
        return false;
    }
    else
    {
        return true;
    }
```

Figura 27 - Método para validar morador na lista

Globalmente, a ControladorMorado.cs facilita a gestão eficiente e consistente das operações relacionadas aos moradores, promovendo a integridade dos dados no sistema.

10. ControladorQuarto.cs

A classe ControladorQuarto é um controlador que gerencia operações relacionadas aos quartos de uma residência.

Método CriarListaDeQuartos(): Este método estático é responsável por criar e retornar uma lista de quartos. Ele itera pelos andares (de 0 a 2) e pelos números dos quartos (de 1 a 19), atribuindo informações aleatórias para cada quarto, como tipo de quarto, capacidade, preço base, preço de renda e disponibilidade, produzidas por métodos auxiliares. Os quartos são criados e adicionados à lista listaDeQuartos. A lista completa é, então, retornada.

```
public static List<Quarto> CriarListaDeQuartos()
{

    // Itera pelos andares (0 a 2) e pelos números dos quartos (1 a 19)
    for (int i = 0; i <= 2; i++)
    {
        for (int j = 1; j < 20; j++)
        {
            // Obtém informações aleatórias para criar um quarto
            string tipoQuarto = ObterTipoQuartoAleatorio();
            int capacidade = ObterCapacidadePorTipoQuarto(tipoQuarto);
            float precoBase = ObterPrecoBasePorTipoQuarto(tipoQuarto);
            float precoRenda = CalcularPrecoRenda(i, precoBase);
            bool disponibilidadeAleatoria = ObterDisponibilidadeAleatoria();

            // Cria um objeto Quarto com as informações obtidas
            Quarto quarto = new Quarto(
                j + i * 100, // ID único do quarto
                tipoQuarto,
                i, // Andar
                capacidade,
                precoRenda,
                disponibilidadeAleatoria
            );

            Quarto.listaDeQuartos.Add(quarto); // Adiciona o quarto à lista de quartos
        }
    }

    return Quarto.listaDeQuartos; // Retorna a lista de quartos criada
}
```

Figura 28 - Classe ControladorQuarto

Método ImprimirListaDeQuartos(List<Quarto> listaDeQuartos): Este método imprime na consola as informações detalhadas de cada quarto presente na lista fornecida como argumento. As informações incluem o ID do quarto, tipo, andar, capacidade, preço da renda e disponibilidade.

```
public static void ImprimirListaDeQuartos(List<Quarto> listaDeQuartos)
{
    Console.WriteLine("Lista de Quartos");
    Console.WriteLine("-----------------------------------------");
    foreach (Quarto quarto in listaDeQuartos)
    {
        Console.WriteLine($"ID: {quarto.GetQuartoId().ToString("D3")}, Tipo: {quarto.GetTipoQuarto()}, Andar: {quarto.GetAndar()}, Capacidade: {quarto.GetCapacidade()}");
    }
}
```

Figura 29 - Método ImprimirListaDeQuartos

Métodos auxiliares de Lista de Quartos:

ObterTipoQuartoAleatorio(): Retorna aleatoriamente um tipo de quarto entre "Individual", "Duplo" e "Studio".

```
private static string ObterTipoQuartoAleatorio()
{
    string[] tiposQuarto = { "Individual", "Duplo", "Studio" }; // Array com os tipos de quarto
    Random random = new Random(); // Objeto Random para gerar números aleatórios
    int indiceAleatorio = random.Next(tiposQuarto.Length); // Gera um número aleatório entre 0 e o tamanho do array
    return tiposQuarto[indiceAleatorio]; // Retorna o tipo de quarto correspondente ao indice gerado
}
```

Figura 30 - Método ObterTipoQuartoAleatório

ObterCapacidadePorTipoQuarto(string tipoQuarto): Retorna a capacidade correspondente a um tipo de quarto (1 para "Individual" e "Studio", 2 para "Duplo").

```
private static int ObterCapacidadePorTipoQuarto(string tipoQuarto)
{
    switch (tipoQuarto)
    {
        case "Individual":
        case "Studio":
            return 1;
        case "Duplo":
            return 2;
        default:
            return 0; // Caso de fallback, caso seja um tipo de quarto desconhecido
    }
```

Figura 31 - Método ObterCapacidadePorTipoQuarto

ObterPrecoBasePorTipoQuarto(string tipoQuarto): Retorna o preço base correspondente a um tipo de quarto.

```
private static float ObterPrecoBasePorTipoQuarto(string tipoQuarto)
{
    switch (tipoQuarto)
    {
        case "Individual":
            return 220;
        case "Duplo":
            return 400;
        case "Studio":
            return 350;
        default:
            return 0; // Caso de fallback, caso seja um tipo de quarto desconhecido
    }
}
```

Figura 32 - Método ObterPrecoBasePorTipoQuarto

CalcularPrecoRenda(int andar, float precoBase): Calcula e retorna o preço da renda de um quarto com base no andar e no preço base, aplicando um acréscimo de 5% por andar.

```
private static float CalcularPrecoRenda(int andar, float precoBase)
{
    return precoBase * (1.0f + 0.05f * andar); // Retorna o preço de renda com base no preço base e no andar, com um acréscimo de 5% por andar
}
```

Figura 33 - Método para calcular preço da renda

ObterDisponibilidadeAleatoria(): Retorna aleatoriamente true ou false para indicar a disponibilidade de um quarto.

```
private static bool ObterDisponibilidadeAleatoria()
{
    Random random = new Random();
    return random.NextDouble() < 0.5; // Retorna true ou false aleatoriamente
}
```

Figura 34 - Método ObterDisponibilidadeAleatoria

11. ControladorGestor.cs e ControladorFuncionario.cs

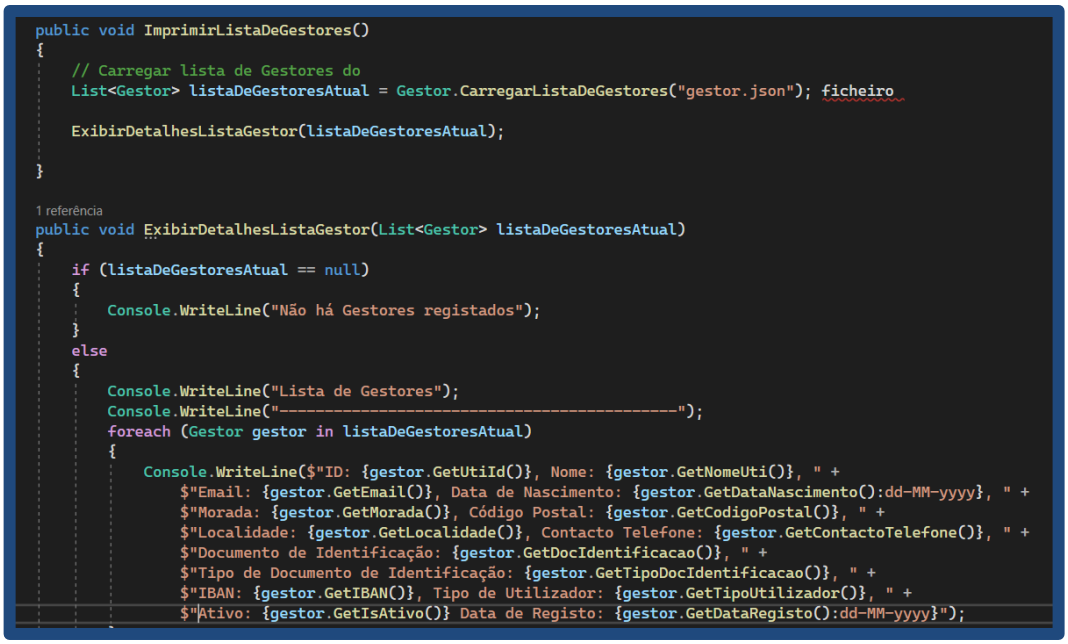
A classe ControladorGestor desempenha o papel de controlador para operações relacionadas aos gestores no sistema. Seus principais métodos incluem a criação de um novo gestor, adição desse gestor à lista existente, impressão da lista de gestores e exibição detalhada dessa lista no console.

O método CriarGestor utiliza as informações de um utilizador existente para criar um novo objeto gestor, preservando a consistência dos dados. Em seguida, o gestor é adicionado à lista de gestores por meio do método AdicionarGestor. Este último método realiza a validação para evitar a inclusão de gestores duplicados, verificando a existência do gestor na lista com base no documento de identificação e tipo de documento.

```
public Gestor CriarGestor(Utilizador utilizador)
{
    Gestor gestor = new Gestor(
        utilizador.GetUtiId(), utilizador.GetNomeUti(),
        utilizador.GetEmail(), utilizador.GetPassword(),
        utilizador.GetDataNascimento(), utilizador.GetMorada(),
        utilizador.GetCodigoPostal(), utilizador.GetLocalidade(),
        utilizador.GetContactoTelefone(), utilizador.GetDocIdentificacao(),
        utilizador.GetTipoDocIdentificacao(), utilizador.GetIBAN(),
        utilizador.GetIsAtivo(),utilizador.GetDataRegisto());

    AdicionarGestor(gestor);
    return gestor;
}

1 referência
public bool AdicionarGestor(Gestor novoGestor)
{
    List<Gestor> listaExistente = Gestor.CarregarListaDeGestores("gestor.json");

    // Verificar se o gestor já existe na lista através do documento de identificação
    if (!ValidarGestor(novoGestor, listaExistente))
    {
        return false;
    }
    else
    {
        listaExistente.Add(novoGestor); // Adicionar gestor à lista de gestores
        Gestor.SalvarListaFicheiro("gestor.json", listaExistente); // Salvar lista de gestores no ficheiro
        return true;
    }
```

Figura 35 - Métodos para criar gestor e adicionar a lista

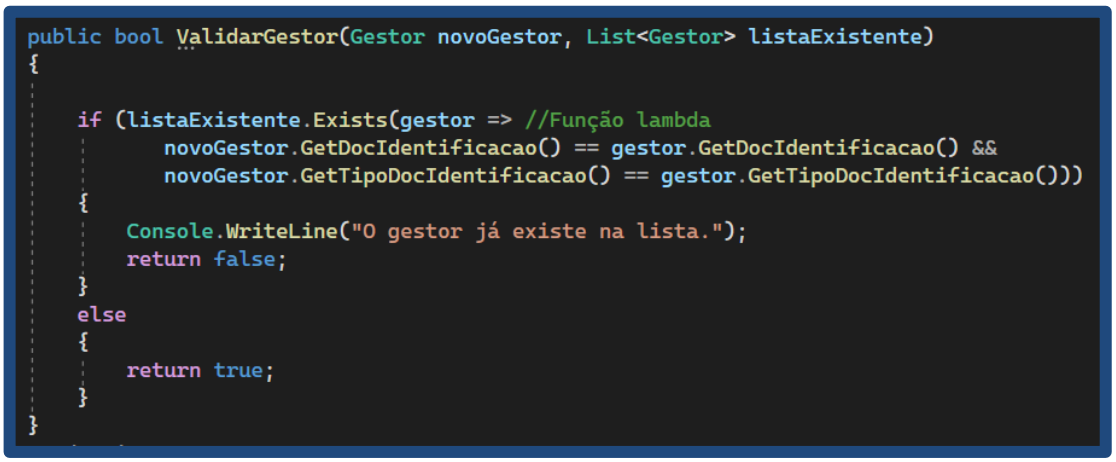
A visualização da lista de gestores é facilitada pelos métodos ImprimirListaDeGestores e ExibirDetalhesListaGestor, que apresentam de forma organizada as informações de cada gestor no console. Similarmente, o método `ExibirDetalhesGestor` permite visualizar os detalhes de um único gestor.



```
public void ImprimirListaDeGestores()
{
    // Carregar lista de Gestores do
    List<Gestor> listaDeGestoresAtual = Gestor.CarregarListaDeGestores("gestor.json"); ficheiro

    ExibirDetalhesListaGestor(listaDeGestoresAtual);

}

1 referência
public void ExibirDetalhesListaGestor(List<Gestor> listaDeGestoresAtual)
{
    if (listaDeGestoresAtual == null)
    {
        Console.WriteLine("Não há Gestores registados");
    }
    else
    {
        Console.WriteLine("Lista de Gestores");
        Console.WriteLine("-----------------------------------------");
        foreach (Gestor gestor in listaDeGestoresAtual)
        {
            Console.WriteLine($"ID: {gestor.GetUtiId()}, Nome: {gestor.GetNomeUti()}, " +
                $"Email: {gestor.GetEmail()}, Data de Nascimento: {gestor.GetDataNascimento():dd-MM-yyyy}, " +
                $"Morada: {gestor.GetMorada()}, Código Postal: {gestor.GetCodigoPostal()}, " +
                $"Localidade: {gestor.GetLocalidade()}, Contacto Telefone: {gestor.GetContactoTelefone()}, " +
                $"Documento de Identificação: {gestor.GetDocIdentificacao()}, " +
                $"Tipo de Documento de Identificação: {gestor.GetTipoDocIdentificacao()}, " +
                $"IBAN: {gestor.GetIBAN()}, Tipo de Utilizador: {gestor.GetTipoUtilizador()}, " +
                $"Ativo: {gestor.GetIsAtivo()} Data de Registo: {gestor.GetDataRegisto():dd-MM-yyyy}");
```

Figura 36 - Métodos para imprimir llista de gestores

Os métodos auxiliares, como ValidarGestor, desempenham um papel crucial na garantia da integridade dos dados, evitando duplicidades na lista de gestores.



```
public bool ValidarGestor(Gestor novoGestor, List<Gestor> listaExistente)
{

    if (listaExistente.Exists(gestor => //Função lambda
            novoGestor.GetDocIdentificacao() == gestor.GetDocIdentificacao() &&
            novoGestor.GetTipoDocIdentificacao() == gestor.GetTipoDocIdentificacao()))
    {
        Console.WriteLine("O gestor já existe na lista.");
        return false;
    }
    else
    {
        return true;
    }
}
```

Figura 37 - Método auxiliar para validar gestor na lista

Este padrão é replicado na classe relacionada ControladorFuncionario, para manter a coerência e facilitar a manutenção do código.

12. ControladorReserva.cs

A classe `ControladorReserva` atua como um controlador responsável por operações relacionadas às reservas no sistema. Seus principais métodos envolvem a criação, adição, busca, edição e exclusão de reservas, proporcionando uma gestão eficaz desse aspecto do sistema.

A funcionalidade de criação de reserva é realizada pelo método `CriarReserva`, que utiliza parâmetros como ID do quarto, ID do utilizador, datas de entrada e saída, e preço da caução. A criação é seguida pela adição da reserva à lista existente, onde são realizadas validações para garantir a consistência dos dados.

```
public Reserva CriarReserva(int quartoId, int utiId, DateTime dataEntrada, DateTime dataSaida, float precoCaucao)
{
    // Obtém o último ID de reserva armazenado no arquivo JSON
    int ultimoId = Reserva.LerUltimoIdReserva("reserva.json");

    // Incrementa o último ID para obter um novo ID único para a nova reserva
    int reservaId = ++ultimoId;

    // Cria uma nova instância de Reserva com os parâmetros fornecidos
    Reserva novaReserva = new Reserva(
        reservaId,
        quartoId,
        utiId,
        dataEntrada,
        dataSaida,
        precoCaucao);

    // Adiciona a nova reserva à lista existente e verifica se a adição foi bem-sucedida
    if (AdicionarReserva(novaReserva))
    {
        // Atualiza o último ID no arquivo JSON com o novo ID da reserva criada
        Reserva.AtualizarUltimoIdNoJson(reservaId, "reserva.json");

        // Retorna a nova reserva criada
        return novaReserva;
    }

    // Retorna null em caso de falha na adição da reserva
    return null;
}
```

Figura 38 - Método para criar reserva

O método `BuscarReserva` possibilita a busca de reservas com base em um predicado, oferecendo flexibilidade na obtenção de sublistas conforme critérios específicos.

```
public List<Reserva> BuscarReserva(Predicate<Reserva> predicado)
{
    List<Reserva> listaExistente = ObterListaAtual();

    if (predicado == null)
    {
        // Se o predicado for nulo, retorna null
        return null;
    }
    else
    {
        // Se o predicado não for nulo, retorna a lista de reservas que satisfazem o predicado
        List<Reserva> reservasEncontradas = listaExistente.FindAll(predicado);
        return reservasEncontradas;
    }
}
```

Figura 39 - Método para buscar uma reserva

Para a edição de reservas, o método `EditarReserva` permite selecionar o campo a ser editado, obtendo um novo valor e atualizando a reserva correspondente na lista existente. Validações são realizadas para assegurar a integridade dos dados.

```
public void EditarReserva(Reserva reserva)
{
    // Obtem a lista atual de reservas
    List<Reserva> listaExistente = ObterListaAtual();
    // Cria uma cópia da lista existente sem a reserva escolhida para fins de validação
    List<Reserva> listaValidacao = listaExistente.Where(r => r.ReservaId != reserva.ReservaId).ToList();
    // Selecionar campo para edição
    TipoCampo tipoCampo = viewReserva.MenuSelecionarCampo();
    // Obtém o novo valor e atualiza a reserva escolhida
    Reserva reservaAtualizada = viewReserva.ObterNovoValorReserva(reserva, tipoCampo);
    // Valida reserva
    if (ValidarReserva(reservaAtualizada, listaValidacao))
    {
        // Se a reserva for válida, atualiza os campos na lista existente
        Reserva reservaExistente = listaExistente.FirstOrDefault(r => r.ReservaId == reserva.ReservaId);
        if (reservaExistente != null)
        {
            // Atualiza os campos na reserva existente com os valores da reserva atualizada
            AtualizarCamposReserva(reservaExistente, reservaAtualizada);
        }
        // Salva as alterações no arquivo
        Reserva.SalvarListaFicheiro("reserva.json", listaExistente);
        Console.WriteLine("Atualização realizada com sucesso.");
    }
    else
    {
        Console.WriteLine("A reserva não é válida. A atualização não foi realizada.");
    }
}
```

Figura 40 - Método para editar campos de uma reserva

A exclusão de uma reserva é efetuada pelo método `ExcluirReserva`, verificando a existência da reserva na lista antes de prosseguir com a exclusão.

```
public void ExcluirReserva(Reserva reserva)
{
    // Obtem a lista atual de reservas
    List<Reserva> listaExistente = ObterListaAtual();

    // Verifica se a reserva existe na lista
    if (!listaExistente.Any(r => r.ReservaId == reserva.ReservaId))
    {
        Console.WriteLine($"Reserva com ID {reserva.ReservaId} não encontrada. A exclusão não foi realizada.");
        return;
    }
    else
    {
        // Se a reserva for válida, exclui a reserva da lista existente
        listaExistente.RemoveAll(r => r.ReservaId == reserva.ReservaId);


        // Salva as alterações no ficheiro
        Reserva.SalvarListaFicheiro("reserva.json", listaExistente);
        Console.WriteLine("Exclusão realizada com sucesso.");
    }

}
```

Figura 41 - Método para excluir uma reserva existente

Há ainda os métodos auxiliares, que de alguma forma já foram utilizados em outrass classes, como ValidarReserva para validações específicas, ObterListaAtual para obter a lista atualizada de reservas, e a enumeração TipoCampo para facilitar a seleção de campos em operações de edição, contribuem para a coesão e eficiência do código.

13. ImenuUtilizador.cs

A interface IMenuUtilizador define um contrato para classes que representam menus de interação para diferentes tipos de utilizadores. Ela contém um único método, `ExibirMenu()`, que é responsável por exibir as opções disponíveis para o utilizador e lidar com as interações correspondentes.

O método ExibirMenu() nas classes que implementam esta interface será responsável por apresentar de forma interativa as opções específicas de cada tipo de utilizador, como moradores, gestores e funcionários. Isso proporciona uma estrutura padronizada para a interação do utilizador, enquanto permite que cada classe de menu personalize suas opções e comportamentos conforme necessário.

```
public interface IMenuUtilizador
{
    7 referências
    void ExibirMenu();
}

/// <summary>
/// Classe que representa o menu de interação com operações relacionadas a cada tipo de utilizador
/// </summary>
7 referências
public abstract class MenuUtilizadorBase : IMenuUtilizador
{
    protected Utilizador UtilizadorAtual;

    3 referências
    public MenuUtilizadorBase(Utilizador utilizador)
    {
        UtilizadorAtual = utilizador;
    }

    7 referências
    public abstract void ExibirMenu();
```

A classe abstrata MenuUtilizadorBase serve como uma implementação parcial dessa interface, fornecendo uma base comum para os menus específicos de cada tipo de utilizador. Ela contém uma propriedade protegida UtilizadorAtual, que armazena o utilizador associado ao menu em questão. O construtor da classe recebe um utilizador como parâmetro e o atribui à propriedade UtilizadorAtual.

A implementação concreta, exemplificada pela classe MenuMorador, estende MenuUtilizadorBase. Nela, o método ExibirMenu() é sobrescrito para fornecer as opções de

menu específicas para moradores. O loop while permite que o menu seja exibido continuamente até que o utilizador escolha sair. Cada opção do menu corresponde a uma ação específica, como realizar, alterar ou cancelar uma reserva, buscar uma reserva, alterar o cadastro, voltar ao menu inicial ou sair.

```
public class MenuMorador : MenuUtilizadorBase
 {
     private ControladorUtilizador controladorUtilizador;
     private ControladorMorador controladorMorador;
     private ViewReserva viewReserva;
     private ControladorReserva controladorReserva;
     public MenuMorador(Utilizador utilizador) : base(utilizador)
     {
         controladorUtilizador = new ControladorUtilizador();
         controladorMorador = new ControladorMorador();
         controladorUtilizador.SetControladorMorador(controladorMorador);
         viewReserva = new ViewReserva();
         controladorReserva = new ControladorReserva();
         viewReserva.SetControladorReserva(controladorReserva);

     }

     public override void ExibirMenu()
     {
         bool continuar = true; // Variável para controlar a continuidade do
loop

         while (continuar)// Loop para exibir o menu
         {
             Console.WriteLine("Menu Morador:");
             Console.WriteLine("1. Realizar Reserva");
             Console.WriteLine("2. Alterar Reserva");
             Console.WriteLine("3. Cancelar Reserva");
             Console.WriteLine("4. Buscar Reserva");
             Console.WriteLine("5. Alterar Cadastro");
             Console.WriteLine("6. Voltar ao Menu Inicial");
             Console.WriteLine("7. Sair");

             Console.Write("Escolha uma opção: ");
             string opcao = Console.ReadLine();

             switch (opcao)
             {
                 case "1":
                     Console.WriteLine("Opção 1 - Realizar Reserva");
                     viewReserva.MenuCriarReserva();
                     break;

                 case "2":
                     Console.WriteLine("Opção 2 - Alterar Reserva");
                     viewReserva.MenuEditarReserva();
                     break;

                 case "3":
                     Console.WriteLine("Opção 3 - Cancelar Reserva");
                     viewReserva.MenuExcluirReserva();
                     break;
```

```
                case "4":
                    Console.WriteLine("Opção 4 - Buscar Reserva");
                    viewReserva.MenuBuscarReserva();
                    break;

                case "5":
                    Console.WriteLine("Opção 5 - Alterar Cadastro");
                    controladorUtilizador.MenuEditarUtilizador();
                    break;

                case "6":
                    Console.WriteLine("Opção 6 - Voltar ao Menu Inicial");
                    Console.Clear();
                    MenuInicial menuInicial = new
MenuInicial(controladorUtilizador);
                    menuInicial.ExibirMenuInicial();
                    continuar = false;
                    break;
                case "7":
                    Console.WriteLine("Saindo...");
                    continuar = false;
                    break;

                default:
                    Console.WriteLine("Opção inválida. Tente novamente.");
                    break;
            }

            if (continuar != false) // Aguardar o usuário pressionar Enter
para limpar o ecrã e continuar
            {
                Console.WriteLine("\nPressione Enter para voltar ao menu...");
                Console.ReadLine();
                Console.Clear();
            }

        }
    }

}
```

Essa estrutura é replicada nas classes MenuGestor e MenuFuncionario, adaptando-se às funcionalidades distintas associadas a gestores e funcionários, respectivamente. Essas classes específicas implementam seus próprios métodos para lidar com as ações relevantes a cada tipo de utilizador.

14. MenuInicial.cs

A classe `MenuInicial` é responsável pela exibição do menu principal do sistema, oferecendo opções como login, registro de utilizador e sair. A interação com o usuário é realizada por meio da entrada do console. Além disso, ela gerencia o redirecionamento para menus específicos com base no tipo de utilizador logado.

O método ExibirMenuInicial apresenta as opções disponíveis e, dependendo da escolha do usuário, executa a funcionalidade correspondente. No caso de login, é invocado o método Login da classe ControladorUtilizador. Em seguida, verifica-se se o login foi bem-sucedido, redirecionando para o menu apropriado caso positivo.

```
  public void ExibirMenuInicial()
  {
      Console.WriteLine("Bem-vindo! Escolha uma opção:");
      Console.WriteLine("1. Login");
      Console.WriteLine("2. Registrar");
      Console.WriteLine("3. Sair");

      string opcao = Console.ReadLine();

      switch (opcao)
      {
          case "1":
              Utilizador utilizadorLogado = ControladorUtilizador.Login();

              if (utilizadorLogado != null)
              {
                  ExibirMenuAposLogin(utilizadorLogado);
                  return;
              }
              else
              {
                  Console.WriteLine("Credenciais inválidas. Tente novamente.");
                  Console.WriteLine("\nPressione Enter para voltar ao menu
inicial...");
                  Console.ReadLine();
                  Console.Clear();
                  ExibirMenuInicial();
              }
              break;
          case "2":
              controladorUtilizador.CriarUtilizador();
              break;
          case "3":
              Console.WriteLine("Até logo!");
              Environment.Exit(0);
              break;
          default:
              Console.WriteLine("Opção inválida. Tente novamente.");
              ExibirMenuInicial();
              break;
      }
  }
```

Caso a escolha seja o registo de um novo utilizador, é chamado o método `CriarUtilizador` do `ControladorUtilizador`.

Para encerrar o programa, o usuário pode selecionar a opção de sair, resultando na chamada a Environment.Exit(0).

O método privado ExibirMenuAposLogin é responsável por redirecionar para menus específicos, dependendo do tipo de utilizador logado. Ele instanciará o menu adequado (como MenuMorador, MenuGestor, ou MenuFuncionario) e exibirá as opções específicas para cada tipo de utilizador.

```
private void ExibirMenuAposLogin(Utilizador utilizador)
{
    // Verifica o tipo de utilizador e exibe o menu apropriado
    switch (utilizador.GetTipoUtilizador().ToLower())
    {
        case "morador":
            new MenuMorador(utilizador).ExibirMenu();
            break;
        case "gestor":
            new MenuGestor(utilizador).ExibirMenu();
            break;
        case "funcionario":
            new MenuFuncionario(utilizador).ExibirMenu();
            break;
        default:
            //opção para identificar algum utilizador que por algum erro tenha sido salvo com um tipo não reconhecido
            Console.WriteLine("Tipo de utilizador não reconhecido.");
            break;
    }
}
```

Figura 42 - Método ExibirMenuAposLogin

15. ViewReserva.cs

A classe ViewReserva é parte do sistema de gerenciamento de reservas e desempenha o papl de interação com o usuário. Nela, foram inseridos métodos que abordam as operações fundamentais do sistema, como criar, buscar, editar, imprimir detalhes e excluir reservas.

O método MenuCriarReserva() é responsável por apresentar um menu para o usuário, solicitando informações essenciais para a criação de uma nova reserva, como ID do quarto, ID do utilizador, datas de entrada e saída, e preço da caução. Após a coleta dessas informações, o método invoca o `ControladorReserva` para efetuar a criação da reserva.

```
public void MenuCriarReserva()
{
    // Título e cabeçalho da operação
    Console.WriteLine("Criar Reserva");
    Console.WriteLine("-------------");
    Console.WriteLine();

    // Entrada de dados para criar uma nova reserva
    int quartoId = Servico.LerInteiro("Insira o ID do quarto: ");
    int utiId = Servico.LerInteiro("Insira o ID do utilizador: ");
    DateTime dataEntrada = Servico.LerData("Insira a data de entrada (dd-MM-yyyy): ");
    DateTime dataSaida = Servico.LerData("Insira a data de saída (dd-MM-yyyy): ");
    float precoCaucao = Servico.LerFloat("Insira o preço da caução: ");

    // Chama o controlador para criar a reserva
    controladorReserva.CriarReserva(quartoId, utiId, dataEntrada, dataSaida, precoCaucao);
}
```

Figura 43 - Menu para Criar Reserva

A busca de reservas é tratada pelo método MenuBuscarReserva(), que permite ao usuário escolher um critério de busca, como ID da Reserva, ID do Quarto, ID do Utilizador, Data de Entrada, Data de Saída ou Data da Reserva. Este método utiliza o `ControladorReserva` para buscar e exibir as reservas com base no critério escolhido.

```
        public Predicate<Reserva> MenuBuscarReserva()
        {
            // Inicializa o predicado como nulo
            Predicate<Reserva> predicado = null;
            // Obtém a lista de reservas atual do controlador
            List<Reserva> listaDeReservasAtual =
controladorReserva.ObterListaAtual();

            // Loop para escolher o critério de busca
            while (true)
            {
                Console.WriteLine("Escolha o critério de busca:");
                Console.WriteLine("1. Por ID da Reserva");
                Console.WriteLine("2. Por ID do Quarto");
                Console.WriteLine("3. Por ID do Utilizador");
                Console.WriteLine("4. Por Data de Entrada");
                Console.WriteLine("5. Por Data de Saída");
                Console.WriteLine("6. Por Data da Reserva");

                // Entrada de opção do usuário
                int opcao = Servico.LerInteiro("");

                int valorInteiro;
                DateTime valorData;

                // Switch para definir o predicado com base na opção escolhida
                switch (opcao)
                {
                    case 1:
                        valorInteiro = Servico.LerInteiro("Digite o ID da
Reserva:");
                        predicado = r => r.ReservaId == valorInteiro;
                        break;
```

```
                    case 2:
                        valorInteiro = Servico.LerInteiro("Digite o ID do
Quarto:");
                        predicado = r => r.QuartoId == valorInteiro;
                        break;
                    case 3:
                        valorInteiro = Servico.LerInteiro("Digite o ID do
Utilizador:");
                        predicado = r => r.UtiId == valorInteiro;
                        break;
                    case 4:
                        valorData = Servico.LerData("Digite a Data de Entrada
(dd-MM-yyyy):");
                        predicado = r => r.DataEntrada == valorData;
                        break;
                    case 5:
                        valorData = Servico.LerData("Digite a Data de Saída
(dd-MM-yyyy):");
                        predicado = r => r.DataSaida == valorData;
                        break;
                    case 6:
                        valorData = Servico.LerData("Digite a Data da Reserva
(dd-MM-yyyy):");
                        predicado = r => r.DataReserva == valorData;
                        break;
                    default:
                        Console.WriteLine("Opção inválida.");
                        break;
                }

                // Se uma opção válida foi escolhida, sai do loop
                if (opcao >= 1 && opcao <= 6)
                {
                    break;
                }
            }

            // Se um predicado foi definido, realiza a busca e exibe os
resultados
            if (predicado != null)
            {
                List<Reserva> listaDeReservasSelecionadas =
controladorReserva.BuscarReserva(predicado);
                ExibirDetalhesListaReserva(listaDeReservasSelecionadas);
            }

            return predicado;
        }
```

A edição de reservas é tratada pelo método MenuEditarReserva(), que utiliza o método de busca para encontrar a reserva desejada. Após a escolha do usuário, o método solicita a seleção do campo a ser editado (como ID do Quarto, ID do Utilizador, Data de Entrada, Data de Saída ou Ativo) e, em seguida, chama o ControladorReserva para realizar a edição.

```
public void MenuEditarReserva()
{
    // Obtém o predicado para buscar as reservas
    Predicate<Reserva> predicado = MenuBuscarReserva();
    // Busca as reservas com base no predicado
    List<Reserva> reservasEncontradas = controladorReserva.BuscarReserva(predicado);
    if (reservasEncontradas != null && reservasEncontradas.Count > 0)
    {
        // Solicita ao usuário que escolha o ID da reserva a ser atualizada
        int idEscolhido = Servico.LerInteiro("Escolha o ID da reserva que deseja atualizar: ");
        // Busca a reserva escolhida
        Reserva reservaEscolhida = reservasEncontradas.FirstOrDefault(r => r.ReservaId == idEscolhido);
        if (reservaEscolhida != null)
        {
            // Exibe detalhes da reserva escolhida
            ExibirDetalhesReserva(reservaEscolhida);
            // Chama o controlador para editar a reserva
            controladorReserva.EditarReserva(reservaEscolhida);
        }
        else
        {
            Console.WriteLine($"Reserva com ID {idEscolhido} não encontrada. A atualização não foi realizada.");
        }
    }
    else
    {
        Console.WriteLine("Reserva não encontrada. A atualização não foi realizada.");
    }
}
```

Figura 44 - Menu para editar reserva

A exclusão de reservas é realizada pelo método MenuExcluirReserva(). Ele também faz uso do método de busca para encontrar as reservas relevantes e, depois da escolha do usuário, invoca o ControladorReserva para realizar a exclusão.

```
public void MenuExcluirReserva()
{
    // Obtém um predicado para buscar as reservas com base nos critérios
escolhidos pelo usuário
    Predicate<Reserva> predicado = MenuBuscarReserva();

    // Busca as reservas com base no predicado
    List<Reserva> reservasEncontradas =
controladorReserva.BuscarReserva(predicado);

    if (reservasEncontradas != null && reservasEncontradas.Count > 0)
    {
        // Solicita ao usuário que escolha o ID da reserva a ser excluída
        int idEscolhido = Servico.LerInteiro("Escolha o ID da reserva que
deseja excluir: ");

        // Busca a reserva escolhida
        Reserva reservaEscolhida = reservasEncontradas.FirstOrDefault(r =>
r.ReservaId == idEscolhido);

        if (reservaEscolhida != null)
        {
            // Exibe detalhes da reserva escolhida
            ExibirDetalhesReserva(reservaEscolhida);

            // Chama o controlador para excluir a reserva
            controladorReserva.ExcluirReserva(reservaEscolhida);
        }
```

```
        else
        {
            // Se a reserva escolhida não foi encontrada, exibe uma mensagem
indicando que a exclusão não foi realizada
            Console.WriteLine($"Reserva com ID {idEscolhido} não encontrada. A
exclusão não foi realizada.");
        }
    }
    else
    {
        // Se nenhuma reserva foi encontrada com base nos critérios escolhidos,
exibe uma mensagem indicando que a exclusão não foi realizada
        Console.WriteLine("Reserva não encontrada. A exclusão não foi
realizada.");
    }
}
```

Esses métodos, integrados com o ControladorReserva e somados a outros métodos auxiliares ou complementares como o ObterNovoValorReserva, proporcionam a interface eficiente para as operações relacionadas a reservas no sistema.

Documentação (Docfx)

O código-fonte do projeto foi devidamente documentado utilizando a ferramenta DocFX. A documentação inclui comentários incorporados diretamente no código-fonte, seguindo o formato de documentação XML suportado pelo C#. Esses comentários foram estruturados de acordo com as convenções do DocFX para garantir uma documentação clara e coesa.

Github

O código-fonte e a documentação completos do sistema de gestão de reservas em residências estudantis desenvolvido como parte deste trabalho acadêmico estão disponíveis no seguinte repositório do GitHub: https://github.com/juliadoriar/TP_POO_24200_24204.git