Gestão de Alojamentos Turísticos

Programação Orientada a Objetos

Enrique George Rodrigues Nº 28602

Instituto Politécnico do Cávado e do Ave
Licenciatura Engenharia de Sistemas Informáticos

15 de Novembro de 2024

Resumo. Este projeto tem como objetivo o desenvolvimento de uma aplicação para a gestão de alojamentos turísticos, permitindo a administração eficiente de registos, reservas e consultas relacionadas com clientes e alojamentos. A solução foi implementada em C#, utilizando o paradigma de programação orientada a objetos, de forma a garantir uma estrutura modular e escalável. As principais funcionalidades incluem o registo de clientes, a gestão de reservas, o processamento de check-ins e check-outs, bem como o controlo de pagamentos. A aplicação destaca-se pelo uso de dicionários como estruturas de dados eficientes, permitindo um acesso rápido e otimizado a registos de clientes, reservas e alojamentos, com uma complexidade temporal de $O(1)$ para operações de consulta e inserção. A persistência de dados é gerida em formato JSON, proporcionando uma solução leve e prática para operadores turísticos.

Palavras-chave: registos, consultas, reservas, check-in, clientes, alojamentos

Índice

Amostras de Código

1 Introdução

1.1 Contexto e Motivação

O setor do turismo enfrenta desafios crescentes na gestão eficiente de alojamentos, exigindo soluções que garantam agilidade, precisão e uma experiência satisfatória para os clientes. Este projeto visa desenvolver um sistema de gestão de alojamentos turísticos que permita a administração eficaz de registos de clientes, reservas e alojamentos. A implementação foi realizada em C#, uma linguagem reconhecida pela sua robustez e flexibilidade, permitindo o desenvolvimento de aplicações escaláveis e eficientes.

1.2 Estrutura de Dados e Persistência

Para garantir um desempenho otimizado, o sistema utiliza dicionários como estruturas de dados principais, que se baseiam em tabelas de hash, proporcionando acesso rápido e eficiente a informações cruciais, com uma complexidade temporal de $O(1)$ para operações de consulta e inserção. Além disso, a persistência de dados é gerida em formato JSON, facilitando a integração e manipulação das informações.

1.3 Boas Práticas de Programação

Embora a eficiência e a funcionalidade sejam os pilares deste projeto, a aplicação também foi concebida com atenção às boas práticas de programação, promovendo um código limpo e manutenível. Desta forma, o sistema não apenas atende às necessidades atuais do setor, mas também está preparado para futuras evoluções e inovações na gestão de alojamentos turísticos.

1.4 Objetivos e Organização do Documento

Este documento descreve o trabalho prático da unidade curricular de Programação Orientada a Objetos, parte integrante da Licenciatura em Engenharia de Sistemas Informáticos no Instituto Politécnico do Cávado e do Ave. O enunciado deste trabalho prático pode ser encontrado no anexo titulado “Trabalho_POO_ESI_2024_2025“. O código fonte está disponível no Github e pode ser consultado aqui.

2 Padrões e Práticas de Programação

Para garantir a consistência, legibilidade e manutenção da base de código, este projeto segue os padrões de programação estabelecidos. Estes padrões estão alinhados com o Guia de Estilo da Microsoft para C# [1], promovendo uniformidade em todo o projeto.

2.1 Convenções de Nomeação

A adoção de uma convenção de nomeação clara melhora a clareza do código. As seguintes convenções foram utilizadas ao longo do projeto:

- **Classes e Structs**: `ClassName`
- **Métodos**: `MethodName()`
- **Propriedades**: `PropertyName`
- **Variáveis de Instância e Locais**: `variableName`
- **Constantes**: `CONSTANT_NAME`
- **Parâmetros**: `parameterName`
- **Enums**: `EnumName`

Evita-se o uso de abreviações excessivas e nomes confusos, promovendo a clareza no código.

2.2 Regras de Formatação

A formatação consistente é fundamental para a legibilidade:

- **Indentação**: 4 espaços por nível de indentação, evitando tabs.
- **Comprimento da Linha**: Linhas limitadas a um máximo de 120 caracteres.
- **Chaves**: As chaves de abertura são colocadas na próxima linha da declaração de controle.
- **Espaçamento**: Espaços em branco são usados para separar operadores e ao redor de chaves.
- **Comentários**: Comentários descritivos são utilizados para explicar seções complexas do código, seguindo um estilo consistente.

2.3 Práticas de Programação

Seguir as melhores práticas garante a qualidade do código:

- Utilizam-se nomes claros para variáveis e métodos, tornando o código auto-explicativo.
- Escreve-se código claro e conciso, evitando complexidade desnecessária.
- Segue-se o Guia de Estilo da Microsoft para C# [1] para formatação e estilo gerais, garantindo a conformidade com os padrões da indústria.

2.4 Controlo de Versões

Utiliza-se Git para o controle de versões, facilitando a gestão eficaz da base de código:

- Cria-se uma nova branch para cada funcionalidade ou correção de erro.
- As branches são criadas a partir da branch principal para desenvolvimento, com nomes descritivos que refletem a funcionalidade ou erro abordado.
- Evita-se fazer commits diretamente na branch principal, garantindo a estabilidade.
- As branches de funcionalidade são regularmente fundidas de volta na branch principal após testes completos.

2.5 Documentação

Utiliza-se *XML Documentation Comments* para documentar o código, permitindo a geração automática de documentação da API:

- Segue-se a sintaxe do XML para comentários, documentando métodos, propriedades e classes.
- A documentação inclui descrições breves, descrições de parâmetros, descrições de valores de retorno e exemplos de uso.
- Utilizam-se tags como `<summary>`, `<param>`, `<returns>` e `<example>` para estruturar corretamente os comentários.
- Define-se o autor nos comentários de cabeçalho de classes para novos ficheiros ou seções significativas de código.

3 Diagrama de Classes

3.1 Introdução ao Sistema

O diagrama de classes apresentado descreve a estrutura lógica do sistema de gestão de reservas para alojamentos turísticos, com uma arquitetura desenhada para centralizar e organizar informações essenciais sobre clientes, reservas, alojamentos e pagamentos. Este modelo, por além de otimizar o fluxo de trabalho, promove a extensibilidade e modularidade, facilitando tanto as expansões futuras como a manutenção do sistema.

3.2 Classes Principais

No diagrama, cada classe representa uma componente fundamental do sistema. A classe *Client* (cliente) armazena informações sobre um cliente individual, como nome, e-mail e endereço. Para gerir vários clientes, o sistema utiliza a classe *Clients*, que funciona como uma coleção para armazenar e organizar várias instâncias da classe *Client*. Esta coleção é implementada através de um dicionário, permitindo o acesso eficiente e rápido aos dados dos clientes.

A classe *Accommodation* (alojamento) representa as propriedades ou quartos disponíveis para reserva, com informações como o tipo de alojamento, a capacidade e o preço por noite. Parecida à classe *Clients*, o sistema utiliza a classe *Accommodations*, que também serve como uma coleção para armazenar e gerir vários alojamentos, sendo implementada através de um dicionário para facilitar a consulta e a manipulação dos dados.

A classe *Reservation* (reserva) liga os clientes aos alojamentos, registando os detalhes da estadia, incluindo as datas de check-in e check-out, o custo total e o estado do pagamento. Tal como as classes *Clients* e *Accommodations*, o sistema utiliza a classe *Reservations*, que guarda uma coleção de reservas, também gerida por um dicionário, permitindo a manipulação eficiente de múltiplas instâncias de reservas.

3.3 Processamento de Pagamentos

A classe *Payment* (pagamento) tem como objetivo guardar informações detalhadas sobre cada transação associada a uma reserva, incluindo o montante pago, a data do pagamento e o método utilizado. A classe *Reservation* mantém uma coleção de instâncias da classe *Payment*, permitindo registar todos os pagamentos realizados para uma determinada reserva.

3.4 BookingManager e a Interface ManageableEntity

A classe *BookingManager* (gestor de reservas) atua como o controlador principal do sistema, fornecendo métodos para adicionar e remover clientes, criar e cancelar reservas, gerir alojamentos e processar pagamentos.

O sistema introduz a interface *ManageableEntity*, implementada pelas classes *Clients*, *Reservations* e *Accommodations*. Esta interface unifica os métodos básicos de gestão, como *Add()*, *Remove()*, *Import()* e *Export()*, permitindo que o *BookingManager* interaja com estas classes de forma uniforme.

3.5 Padrão Facade

O *BookingManager* atua como a face do sistema, permitindo direcionar chamadas para a classe apropriada com base no tipo de entidade a ser manipulada. Isto segue o *Padrão Facade* [2], que simplifica a interação com o sistema ao fornecer uma interface única e centralizada. Este padrão oculta as complexidades internas do sistema, como a gestão dos dados em várias coleções, e permite que o utilizador ou outros componentes do sistema façam operações de forma mais intuitiva e eficiente. Ao centralizar a lógica de interação, o *Facade* promove uma maior modularidade e extensibilidade, uma vez que mudanças internas podem ser feitas sem impactar a interface externa. Desta forma, o *BookingManager* oferece um ponto único de acesso para a criação, manipulação e gestão das entidades, facilitando a manutenção e escalabilidade do sistema.

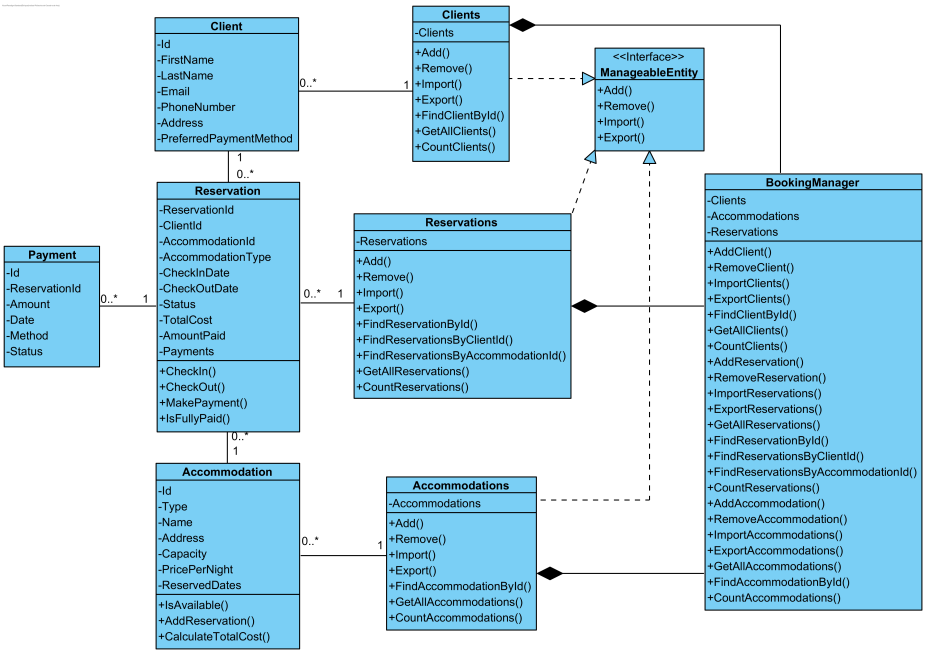

Fig. 1: Diagrama de Classes do Sistema de Gestão de Alojamentos Turísticos

4 Organização do Código

A arquitetura do sistema foi estruturada de acordo com um layout comum e amplamente utilizado em aplicações de software, que segue o padrão de separação de responsabilidades entre diferentes camadas. Este padrão facilita a manutenção, escalabilidade e testabilidade do sistema. A divisão em *Models*, *Repositories*, *Services* e *Utilities* organiza a aplicação de maneira modular e coesa, garantindo que cada componente tenha uma função clara e bem definida. Cada classe é guardada num ficheiro chamado de forma correspondente, reforçando a organização e facilitando a navegação no código.

4.1 Nomeação de Ficheiros

Para manter uma estrutura organizada e intuitiva, os ficheiros do sistema foram nomeados de acordo com as classes que eles contêm. Cada classe tem um ficheiro correspondente com o mesmo nome, garantindo uma correspondência direta entre a funcionalidade e o ficheiro onde ela está implementada. Esta abordagem facilita a localização de classes específicas e melhora a navegabilidade do projeto, uma vez que o nome de cada ficheiro reflete o conteúdo e a responsabilidade do mesmo. Por exemplo, a classe *Client* está localizada no ficheiro `Client.cs`, enquanto a classe *BookingManager* pode ser encontrada em `BookingManager.cs`.

4.2 Conformidade com a CLS

A *Common Language Specification (CLS)* define um conjunto de regras e padrões para garantir a interoperabilidade entre as diferentes linguagens que compõem o ecossistema .NET. Ao desenvolver o sistema, foi dada atenção à conformidade com a CLS para assegurar que o código seja acessível e utilizável por outras linguagens compatíveis com o .NET.

Dentro deste contexto, o tipo `decimal` foi escolhido para operações financeiras e monetárias devido à sua alta precisão e, também, por ser um tipo que segue as normas da CLS. Esta conformidade permite que o sistema mantenha uma representação exata de valores monetários e seja facilmente reutilizável em outros ambientes e linguagens que suportem .NET.

Cabe destacar que nem todos os tipos numéricos são compatíveis com a CLS — por exemplo, tipos inteiros não-assinados como `uint` e `ulong` não são CLS-compliant e podem apresentar problemas de interoperabilidade em algumas linguagens. A escolha de tipos compatíveis com a CLS reforça o compromisso com a robustez e a acessibilidade do sistema em um ambiente .NET multi-linguagem.

4.3 Models

A camada de *Models* é responsável por representar os dados essenciais do sistema. Nesta camada, as classes de modelo (como *Client*, *Accommodation* e *Reservation*) são definidas com propriedades e comportamentos relacionados aos dados que

representam. Esses modelos são frequentemente utilizados para mapear as entidades que serão manipuladas pela aplicação, sendo a base para as operações de armazenamento e recuperação de dados. Além disso, os modelos podem incluir validações de dados e regras de negócio específicas, facilitando a comunicação entre as diferentes camadas.

4.4 Repositories

A camada de *Repositories* no nosso sistema é responsável por guardar e organizar listas de objetos de diferentes tipos, como Clientes, Alojamentos e Reservas. Esta camada serve como um *container* centralizado para gerir estes conjuntos de dados durante a execução do programa, proporcionando um acesso estruturado e consistente às informações.

4.5 Services

A camada de *Services* contém a lógica de negócio principal da aplicação. Os serviços orquestram as interações entre os modelos e os repositórios, executando operações mais complexas que envolvem várias entidades ou ações. No contexto do nosso sistema de gestão de reservas, o *BookingManager* pode ser considerado um serviço, pois gere o processo de reservas, incluindo a criação, validação e processamento de pagamentos.

4.6 Utilities

A camada de *Utilities* contém classes e métodos auxiliares que oferecem funcionalidades que podem ser usadas em diferentes partes do sistema. Exemplos típicos incluem ferramentas de validação, manipulação de strings, loggers e geradores de relatórios. As *Utilities* ajudam a manter o código limpo e reutilizável, evitando a duplicação de lógica em várias partes da aplicação. Por exemplo, um validador de datas ou um utilitário de formatação de preços pode ser usado em diferentes componentes, como na criação de reservas e no processamento de pagamentos.

5 Estruturas de Dados Utilizadas

5.1 Utilização de Dictionary

As estruturas de dados escolhidas para o sistema foram definidas com o objetivo de otimizar a eficiência e o desempenho, especialmente em operações de busca, inserção e remoção. A principal estrutura de dados utilizada foi o *Dictionary* (dicionário), que oferece uma maneira eficiente de armazenar e aceder a entidades como clientes, reservas e alojamentos, por meio de chaves únicas. Esta estrutura garante um tempo de acesso quase constante para operações como inserção e busca, o que é essencial para o bom desempenho do sistema, especialmente à medida que o número de entidades cresce.

```
readonly Dictionary<int, Client> _clientDictionary = new
    Dictionary<int, Client>();
```

Amostra de Código 1: Uso de Dictionary

O uso do *Dictionary* garante que as operações de inserção e remoção de clientes sejam realizadas de forma rápida, com um tempo médio de execução constante $O(1)$, o que melhora a escalabilidade do sistema à medida que mais clientes são adicionados.

5.2 Utilização de List e Pesquisa Binária

Para a classe *Reservation*, foi utilizada uma *List* de tuplas para guardar intervalos de datas de reservas, o que permite gerir as datas de forma eficiente. A lista é ordenada, e para verificar a disponibilidade de um alojamento, utiliza-se a técnica de *pesquisa binária*, que permite encontrar rapidamente se há sobreposição de datas entre reservas. Abaixo está um exemplo de como o sistema verifica se existe disponibilidade:

```
public bool IsAvailable(DateTime startDate, DateTime endDate)
{
        if (endDate <= startDate)
        throw new ArgumentException("End date must be after the start
            date.");

        int index = _reservedDates.BinarySearch((startDate, endDate),
            new DateRangeComparer());
        if (index < 0)
        index = ~index; // A pesquisa binária retorna o complemento
            do índice de inserção

        if (index > 0 && _reservedDates[index - 1].End > startDate)
        {
                return false; // Sobreposição com a reserva anterior
```

```
13          }
14          if (index < _reservedDates.Count &&
                _reservedDates[index].Start < endDate)
15          {
16                  return false; // Sobreposição com a reserva seguinte
17          }
18
19          return true; // Não há sobreposição, o alojamento está
                disponível
20  }
21
22  return true; // Não há sobreposição, o alojamento está disponível
```

Amostra de Código 2: Uso de Pesquisa Binária

Ao utilizar a *List* de tuplas para armazenar intervalos de datas e a pesquisa binária para verificar a disponibilidade, o sistema é capaz de realizar verificações de disponibilidade de maneira muito mais eficiente do que uma busca linear. A *pesquisa binária* reduz o tempo de busca de $O(n)$ para $O(log(n))$, o que é um ganho significativo em termos de desempenho, principalmente em sistemas com muitas reservas.

5.3 Vantagens das Estruturas de Dados Escolhidas

As escolhas de *Dictionary* e *List* oferecem várias vantagens para o sistema:

- **Eficiência de Acesso**: O *Dictionary* permite acesso rápido às entidades, como clientes e reservas, através de suas chaves únicas. Isto melhora a performance, especialmente quando o sistema lida com um grande volume de dados.
- **Organização e Escalabilidade**: O uso do *Dictionary* para armazenar entidades e o *List* para reservas garante que o sistema seja facilmente escalável. À medida que o número de clientes ou reservas cresce, o desempenho do sistema permanece estável devido à eficiência destas estruturas.
- **Pesquisa Rápida de Intervalos de Datas**: A combinação de *List* com pesquisa binária permite que a verificação de disponibilidade de um alojamento seja realizada de forma muito mais eficiente do que outras abordagens. A inserção das reservas na lista ordenada mantém a integridade dos dados e facilita a verificação de conflitos de datas.

Estas escolhas refletem uma consideração cuidadosa da necessidade de desempenho e eficiência, ao mesmo tempo que garantem uma implementação simples e direta, essencial para manter o sistema modular e fácil de manter.

6 Uso de LINQ

No desenvolvimento do sistema, foi utilizado o LINQ (*Language Integrated Query*) para facilitar a consulta e manipulação de coleções de dados, proporcionando uma forma mais concisa e legível de trabalhar com dados em memória. O LINQ integra-se diretamente no código C#, permitindo escrever consultas diretamente nas coleções, sem necessidade de recorrer a linguagens de consulta externas ou a loops complicados.

6.1 Vantagens de LINQ

O LINQ oferece várias vantagens, entre as quais se destacam:

- **Sintaxe Concisa e Legível**: Permite escrever consultas de forma mais direta e legível, o que resulta em um código mais limpo e fácil de entender.
- **Integração com Coleções de Dados**: Pode ser utilizado diretamente sobre coleções de objetos como listas, arrays, e dicionários, o que facilita a manipulação de dados sem recorrer a ferramentas externas.
- **Redução de Erros e Complexidade**: Ao abstrair a complexidade da consulta e manipulação de dados, o LINQ reduz as possibilidades de erros e torna o código mais modular e menos propenso a falhas.
- **Execução Diferida**: O LINQ utiliza execução diferida, ou seja, a consulta só é executada quando necessário, o que pode melhorar o desempenho, especialmente quando se lida com um volume grande de dados.

6.2 Utilização de LINQ no Código

A seguir, são apresentados exemplos de como o LINQ foi utilizado no código para realizar operações de filtragem em coleções de dados.

O primeiro exemplo mostra como o LINQ é usado para encontrar todas as reservas associadas a um cliente, dado o seu ID único. A função '*FindReservationsByClientId()*' filtra o dicionário de reservas, retornando apenas as reservas que correspondem ao '*clientId*' fornecido.

```
1 public IEnumerable<Reservation> FindReservationsByClientId(int
      clientId)
2 {
3         return _reservationDictionary.Values.Where(r => r.ClientId ==
              clientId);
4 }
```

Amostra de Código 3: Encontra Reservas por ID de Cliente

No segundo exemplo, o LINQ é utilizado para encontrar todas as reservas associadas a um alojamento específico, filtrando as reservas pelo 'AccommodationId'. A função 'FindReservationsByAccommodationId' realiza uma consulta parecida, mas com base no ID do alojamento.

```
public IEnumerable<Reservation> FindReservationsByAccommodationId(int
    accommodationId)
{
        return _reservationDictionary.Values.Where(r =>
            r.AccommodationId == accommodationId);
}
```

Amostra de Código 4: Encontra Reservas por ID de Alojamento

Estes exemplos demonstram como o LINQ simplifica a filtragem de dados nas coleções, tornando o código mais legível e eficiente.

7 Tratamento de Exceções

A escolha de utilizar códigos de erro, ao invés de mensagens *hardcoded*, proporciona maior flexibilidade, reutilização e facilita a manutenção do código. Além disso, permite uma melhor adaptação a mudanças nas regras de negócio, sem a necessidade de alteração direta nas mensagens de erro.

7.1 Estrutura das Exceções

A base para o tratamento de exceções no sistema é a classe `ValidationException`. Esta classe personalizada herda da classe base `Exception` e é usada para encapsular falhas de validação que ocorrem durante o processamento dos dados.

A principal característica da `ValidationException` é que ela recebe um código de erro (do tipo `ValidationErrorCode`) e, ao ser instanciada, busca automaticamente a mensagem correspondente através da classe `ValidationErrorMessages`. O código abaixo apresenta a implementação básica desta exceção personalizada:

```
1  namespace SmartStay.Validation
2  {
3          public class ValidationException : Exception
4          {
5                  public ValidationException(ValidationErrorCode
                       errorCode)
6                  :
                       base(ValidationErrorMessages.GetErrorMessage(errorCode))
7                  {
8                      ErrorCode = errorCode;
9                  }
10
11                 public ValidationErrorCode ErrorCode { get; }
12         }
13 }
```

Amostra de Código 5: Classe ValidationException

Como pode ser observado, ao receber um `ValidationErrorCode`, a classe `ValidationException` utiliza o método `GetErrorMessage` da classe `ValidationErrorMessages` para obter a mensagem correspondente ao erro, evitando a necessidade de codificar as mensagens diretamente na aplicação.

7.2 Definição dos Códigos de Erro

A enumeração `ValidationErrorCode` é responsável por definir os diferentes tipos de erros que podem ocorrer durante a validação dos dados. Cada valor desta enumeração é associado a um código numérico único, o que permite identificar de forma eficiente qual validação falhou.

Abaixo, apresentamos um exemplo simplificado dessa enumeração:

```
namespace SmartStay.Validation
{
        public enum ValidationErrorCode
        {
                InvalidName = 1001,
                InvalidEmail = 1002,
                InvalidPhoneNumber = 1003
        }
}
```

Amostra de Código 6: Enumeração ValidationErrorCode

Cada valor da enumeração representa um tipo específico de erro de validação, como por exemplo, nome inválido, e-mail inválido, ou número de telefone inválido. Esta enumeração é usada na classe `ValidationException` para determinar o erro que ocorreu.

7.3 Mensagens de Erro Dinâmicas

A classe `ValidationErrorMessages` é responsável por mapear cada código de erro para uma mensagem descritiva correspondente. Em vez de codificar as mensagens diretamente nas classes de validação ou nas exceções, as mensagens são armazenadas em um dicionário, o que facilita a manutenção e a atualização das mensagens de erro.

Veja abaixo um exemplo simplificado dessa classe:

```
namespace SmartStay.Validation
{
        public static class ValidationErrorMessages
        {
                private static readonly
                    Dictionary<ValidationErrorCode, string>
                    ErrorMessages = new() {
                        { ValidationErrorCode.InvalidName, "O nome
                            deve ser uma string não vazia e não
                            exceder 50 caracteres." },
                        { ValidationErrorCode.InvalidEmail, "O
                            endereço de e-mail é inválido ou não
                            corresponde ao formato exigido." }
                };

                public static string
                    GetErrorMessage(ValidationErrorCode errorCode)
                {
                        return ErrorMessages.TryGetValue(errorCode,
                            out var message) ? message : "Erro de
                            validação desconhecido.";
                }
        }
```

```
15 }
```

Amostra de Código 7: Classe ValidationErrorMessages

Desta forma, quando uma exceção de validação é gerada, a mensagem correspondente ao código de erro é buscada dinamicamente a partir do dicionário. Caso o código de erro não esteja mapeado, uma mensagem padrão "Erro de validação desconhecido" será retornada, garantindo que todas as exceções sejam tratadas de forma consistente.

7.4 Benefícios da Abordagem

A utilização de códigos de erro e mensagens dinâmicas traz uma série de benefícios para o sistema:

- **Facilidade de Manutenção**: A manutenção das mensagens de erro torna-se mais simples, uma vez que as mensagens estão centralizadas num único local e não espalhadas pelo código.
- **Escalabilidade**: Novos erros podem ser facilmente adicionados à enumeração e ao dicionário de mensagens, sem a necessidade de reescrever as exceções ou a lógica de validação.
- **Internacionalização**: Como as mensagens de erro estão centralizadas, é mais fácil adaptar o sistema para diferentes idiomas no futuro.
- **Consistência**: Ao utilizar códigos de erro numéricos e associar cada um a uma mensagem específica, garantimos que o tratamento de exceções será consistente ao longo de toda a aplicação.

7.5 Exemplo de Uso

Abaixo, é apresentado um exemplo de como as exceções de validação são utilizadas no código. Neste exemplo, caso o nome fornecido por um cliente seja inválido, uma exceção de validação é gerada e a mensagem correspondente é retornada:

```
1 try
2 {
3         ValidateClientName(client.Name);
4 }
5 catch (ValidationException ex)
6 {
7         Console.WriteLine($"Erro de validação: {ex.Message}");
8 }
```

Amostra de Código 8: Uso da Exceção de Validação

Neste exemplo, se o nome do cliente for inválido, a exceção `ValidationException` será lançada, e a mensagem de erro correspondente ao código de erro será mostrada ao utilizador.

8 Tratamento de Logs

8.1 Objetivo de Logs

8.2 Exemplos de Logging no Código

9 Persistência de Dados com Ficheiros

A persistência de dados é uma componente essencial para qualquer sistema de gestão de informações. No caso deste projeto, optei por utilizar ficheiros JSON para armazenar os dados do sistema, como clientes, acomodações e reservas. O formato JSON foi escolhido devido à sua popularidade em aplicações web, móveis e desktop, à facilidade de leitura e escrita e ao seu suporte em diversas linguagens de programação.

9.1 Escolha do Formato de Armazenamento

Optou-se pelo formato JSON para persistir os dados devido a vários fatores:

- **Simplicidade e Legibilidade**: O JSON é um formato de texto simples, que pode ser facilmente lido por humanos e máquinas. Isso facilita tanto a depuração durante o desenvolvimento quanto a análise de dados.
- **Compatibilidade Universal**: O JSON é amplamente suportado por diversas plataformas e frameworks, tornando-o ideal para integração com sistemas externos e APIs.
- **Estrutura Flexível**: O JSON oferece uma estrutura de dados hierárquica e facilmente extensível, o que é vantajoso para representar dados complexos como clientes, reservas e acomodações de forma clara e eficiente.
- **Desempenho**: As bibliotecas modernas de manipulação de JSON são muito eficientes, permitindo uma rápida serialização e desserialização dos dados.

9.2 Importação e Exportação de Dados

Para garantir que os dados possam ser facilmente armazenados e recuperados de forma persistente, implementaram-se métodos para exportar e importar os dados para e a partir de JSON. A seguir, são apresentados os métodos utilizados para a importação e exportação da lista de clientes, com a mesma abordagem sendo aplicada para acomodações e reservas.

Importação de Dados O método `Import` permite importar uma lista de clientes a partir de uma string JSON. Caso a string JSON seja inválida ou nula, são lançadas exceções específicas para garantir que os dados sejam válidos antes de serem inseridos na coleção. O código do método é o seguinte:

```
1 public void Import(string data)
2 {
3         if (string.IsNullOrEmpty(data))
4         {
5                 throw new ArgumentException("Data cannot be null or
                      empty", nameof(data));
6         }
7
```

```
8           var clients = JsonHelper.DeserializeFromJson<Client>(data) ??
9           throw new ArgumentException("Deserialized client data cannot
                be null", nameof(data));
10
11          foreach (var client in clients)
12          {
13                  _clientDictionary[client.Id] = client; // Inserção
                        direta para maior eficiência
14          }
15  }
```

Amostra de Código 9: Método Import para Importar Clientes

No método `Import`, são realizadas as seguintes validações:

- Se a string `data` for nula ou vazia, uma exceção `ArgumentException` é lançada.
- A deserialização do JSON é realizada utilizando um método auxiliar (`DeserializeFromJson`), e, caso falhe, outra exceção `ArgumentException` é gerada.
- Para cada cliente na lista importada, o cliente é adicionado ao dicionário `_clientDictionary`, substituindo qualquer cliente existente com o mesmo ID.

Exportação de Dados O método `Export` permite exportar a lista atual de clientes para uma string JSON, que pode ser armazenada num ficheiro ou enviada para outros sistemas. A implementação do método é simples:

```
1  public string Export()
2  {
3           return JsonHelper.SerializeToJson(_clientDictionary.Values ??
                Enumerable.Empty<Client>());
4  }
```

Amostra de Código 10: Método Export para Exportar Clientes

O método `Export` converte os valores armazenados no dicionário `_clientDictionary` numa string JSON utilizando o método auxiliar `SerializeToJson`. Se o dicionário estiver vazio, é retornada uma coleção vazia para garantir que o formato JSON seja válido.

9.3 Validação e Exceções na Persistência de Dados

É importante garantir que os dados sejam consistentes e válidos antes de serem persistidos no formato JSON. Para isso, o sistema inclui verificações e exceções que asseguram que a importação e exportação dos dados sejam feitas corretamente.

As exceções tratadas nas operações de importação e exportação incluem:

- **ArgumentException**: Lançada quando os dados fornecidos são nulos, vazios ou falham na desserialização.

– **InvalidOperationException**: Pode ser lançada em casos em que ocorre uma operação inesperada, como ao tentar aceder dados que não foram carregados corretamente.

Estas exceções permitem que o sistema lide com dados inválidos de forma controlada, sem causar falhas inesperadas. Além disso, a validação dos dados durante a importação e exportação impede que dados corrompidos ou malformados sejam persistidos, garantindo a integridade do sistema.

9.4 Considerações Finais sobre a Persistência de Dados

A escolha pelo formato JSON para persistência de dados oferece diversas vantagens em termos de simplicidade, flexibilidade e compatibilidade. A utilização de métodos de importação e exportação bem definidos, aliados ao tratamento de exceções para garantir a validade dos dados, proporciona um sistema robusto e fácil de manter. Além disso, a escolha de um formato de dados leve e amplamente suportado como o JSON garante que o sistema possa ser facilmente integrado a outras plataformas e serviços.

10 Qualidade do Código e Boas Práticas

10.1 Testes Unitários

10.2 CI/CD Pipeline

10.3 Padrões de Design Utilizados

10.4 Lambda Functions

11 Arquitetura em Camadas (Layered Architecture)

A adoção de uma arquitetura em camadas é uma prática muito reconhecida para organizar e estruturar o código de uma aplicação, visando separar responsabilidades e promover escalabilidade e manutenção. Dentro desta abordagem, o padrão de **Model-View-Controller (MVC)** [3] foi escolhido para gerir a interface de utilizador e a lógica de aplicação. Este padrão permite uma divisão clara entre as camadas que lidam com os dados, a lógica de negócio e a apresentação visual, facilitando o desenvolvimento modular e a evolução do sistema ao longo do tempo.

11.1 Escolha da Arquitetura

O padrão **MVC (Model-View-Controller)** [3] foi escolhido pela sua capacidade de simplificar o desenvolvimento de interfaces de utilizador complexas ao isolar as responsabilidades de cada componente. No contexto de um sistema de gestão de alojamentos, onde há interações contínuas entre utilizadores, dados e regras de negócio, o MVC destaca-se pela facilidade com que permite adicionar ou modificar funcionalidades na interface sem impactar diretamente a lógica de negócio ou a camada de dados.

As vantagens do MVC incluem:

- **Modularidade e Reutilização**: A estrutura do MVC facilita a reutilização de código, pois a lógica de negócio (Controller) e os dados (Model) são isolados da camada de apresentação (View), permitindo que o sistema suporte múltiplas interfaces ou plataformas sem duplicação de lógica.
- **Facilidade de Testes e Manutenção**: A divisão de responsabilidades permite testes mais eficazes em cada camada, especialmente na camada de lógica (Controller), que pode ser isolada para testes unitários sem necessidade de interações com a interface.
- **Escalabilidade**: O MVC permite que o sistema se expanda de forma coesa, com cada camada sendo ampliada ou substituída sem impacto nas demais, essencial em sistemas que requerem evolução contínua.

11.2 Desenvolvimento Backend Inicial e Adição de UI

O desenvolvimento deste sistema começou com a implementação das camadas fundamentais, como *Model*, *Repository*, *Service*, *Utility* e *Validation*. Estas camadas representam a lógica de negócios central e a persistência de dados, que são independentes da interface de utilizador. Iniciar com estas camadas permitiu que a lógica essencial fosse desenvolvida, testada e validada antes da introdução da interface visual.

Esta abordagem foi adotada com os seguintes benefícios:

- **Foco na Lógica de Negócio**: Começar com a implementação das camadas de Model, Repository, Service e Validation permitiu concentrar esforços na criação e validação da lógica de negócios antes de adicionar a complexidade da interface de utilizador.

- **Testabilidade**: A ausência de uma interface de utilizador no início permitiu que as camadas de lógica fossem testadas de forma independente e unitária, garantindo que a interação entre elas funcionasse corretamente antes de integrar a UI.
- **Facilidade de Evolução**: Ao desenvolver a lógica de negócios separadamente da UI, foi possível garantir que o sistema estivesse bem estruturado e funcional antes de se preocupar com a apresentação visual. Isso permitiu que a interface fosse adicionada posteriormente de forma coesa, sem impactar as camadas já implementadas.

Depois de garantir que as camadas essenciais estavam a funcionar corretamente, a camada de View (UI) foi então integrada ao sistema. O Controller e a View passaram a interagir com as camadas de *Model*, *Repository* e *Service*, proporcionando a interface de utilizador que permite aos utilizadores interagir com o sistema de maneira intuitiva.

Este processo progressivo de desenvolvimento inicial das camadas fundamentais, seguido pela implementação da interface de utilizador, proporcionou uma base sólida e escalável para o sistema.

12 Conclusão

Lorem Ipsum.

Referências

1. Microsoft: Microsoft C# Style Guide. `https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/identifier-names` (2024), visto: 2024-10-24
2. Wikipedia: Facade Pattern. `https://en.wikipedia.org/wiki/Facade_pattern` (2024), visto: 2024-11-13
3. CodeAcademy: MVC: Model, View, Controller. `https://www.codecademy.com/article/mvc` (2024), visto: 2024-11-14

Todas as ligações foram consultadas pela última vez no dia 15 de novembro de 2024.