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

17 de Dezembro 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 e binária, 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 optimizado, o sistema utiliza dicionários como estruturas de dados principais, baseados em tabelas de hash, que proporcionam um acesso rápido e eficiente às informações cruciais, com uma complexidade temporal de $O(1)$ para operações de consulta e inserção.

Adicionalmente, a persistência de dados é gerida em formato JSON, o que facilita a integração com outros sistemas e a manipulação dos dados de forma legível e estruturada. No entanto, para operações que exigem maior eficiência em termos de tempo de leitura/escrita, como guardar ou carregar rapidamente os dados sem necessidade de exportação ou importação, utilizei o **protobuf-net** para serializar e desserializar os dados em formato binário. Esta abordagem combina a flexibilidade do JSON com a velocidade do formato binário, garantindo um equilíbrio entre interoperabilidade e desempenho.

1.3 Boas Práticas de Programação

Embora a eficiência e a funcionalidade sejam os pilares deste projeto, a aplicação também foi criada 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`
- **Variáveis privadas**: `_variableName`
- **Constantes**: `CONSTANT_NAME`
- **Parâmetros**: `parameterName`
- **Enums**: `EnumName`
- **Métodos de Teste**: `MethodName_Condition_ExpectedOutcome`

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.

2.6 Padrões de Design Utilizados

O sistema implementado segue princípios sólidos de engenharia de software, recorrendo a padrões de design bem estabelecidos para garantir uma arquitectura modular, reutilizável e fácil de manter. Dois dos principais padrões utilizados foram:

- **MVC (Model-View-Controller)**: Este padrão foi utilizado para separar as preocupações do sistema. O backend implementa as regras de negócio e a manipulação de dados no modelo (Model), enquanto o frontend é responsável pela apresentação visual (View) e interage com o modelo através de controladores (Controllers). Este design melhora a organização do código e facilita a introdução de alterações numa camada sem impactar as outras.
- **Facade (Face)**: Uma classe face do sistema foi utilizada para simplificar e abstrair interações complexas entre diferentes componentes do sistema. Este padrão oferece uma interface simples e clara para funcionalidades comuns.

Estes padrões permitiram a divisão eficiente do código em duas camadas principais: backend e frontend, promovendo um design coeso e robusto.

2.7 Funções Lambda

O sistema também aproveita as funções lambda para simplificar a manipulação de coleções e propriedades. Estas expressões são frequentemente utilizadas para criar acessores e transformar coleções de forma concisa e eficiente. Seguem dois exemplos ilustrativos:

```
1 public static int LastAssignedId
2 {
3         get => _lastAccommodationId;
4         set => _lastAccommodationId = value;
5 }
```

Amostra de Código 1: Getter e Setter com Função Lambda

No exemplo acima, as expressões lambda proporcionam uma forma compacta e clara de definir os métodos de acesso para propriedades.

3 Diagrama de Classes

3.1 Introdução ao Sistema

O diagrama de classes 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, proprietários, 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) guarda informações sobre um cliente individual, como nome, e-mail, telefone e endereço. A classe *Clients* funciona como uma coleção que organiza várias instâncias de *Client* utilizando um dicionário, oferecendo acesso eficiente e métodos para manipulação como `Add()` e `Remove()`.

De forma semelhante, a classe *Owner* (proprietário) guarda dados sobre os proprietários, incluindo informações pessoais e as acomodações associadas. A classe *Owners* organiza múltiplos *Owner* utilizando um dicionário, facilitando operações como adição, remoção e pesquisa.

A classe *Accommodation* (alojamento) representa as unidades disponíveis para reserva. Cada alojamento inclui detalhes como o tipo de alojamento, o nome e a lista de quartos associados (*Rooms*). A classe *Room* fornece detalhes específicos sobre os quartos, como tipo, preço por noite e disponibilidade, além de funcionalidades para calcular custos e verificar disponibilidade. Para gerir vários alojamentos, a classe *Accommodations* organiza instâncias de *Accommodation* utilizando um dicionário, com métodos como `FindRoomById()` e `RemoveRoom()`.

A classe *Reservation* (reserva) liga clientes a acomodações, registrando detalhes como datas de check-in e check-out, tipo de alojamento, custo total e estado da reserva. Ela também mantém uma lista de pagamentos (*Payment*) associados, permitindo verificar se os pagamentos estão atualizados. A classe *Reservations* armazena várias reservas em um dicionário e oferece funcionalidades para adicionar, cancelar ou pesquisar reservas.

A classe *Payment* (pagamento) é responsável por registar informações detalhadas sobre transações, incluindo o montante pago, método de pagamento e data.

3.3 Processamento de Pagamentos

A classe *Payment* lida com detalhes individuais das transações financeiras associadas a reservas. A classe *Reservation* mantém uma lista de pagamentos e fornece métodos como `MakePayment()` para processar pagamentos e verificar o estado financeiro.

3.4 BookingManager e a Interface ManageableEntity

A classe *BookingManager* atua como o controlador central/face do sistema, sendo responsável por gerir os clientes, proprietários, reservas e alojamentos. Ela utiliza instâncias das classes *Clients*, *Owners*, *Reservations* e *Accommodations*, centralizando as operações.

Para uniformizar as operações de gestão, o sistema utiliza a interface *ManageableEntity*, implementada pelas classes de coleção (*Clients*, *Owners*, *Reservations* e *Accommodations*). Esta interface define métodos comuns, como `Add()`, `Remove()`, `Export()` e `Load()`.

O *BookingManager* centraliza operações comuns, como `CreateBasicClient()`, `FindClientById()` e `SaveAll()`, fornecendo uma interface unificada e simples para os utilizadores.

3.5 Padrão de Face (Facade Pattern)

O *BookingManager* implementa o *Padrão Facade* [2], funcionando como um ponto de acesso único que simplifica a interação com o sistema. Em vez de chamar métodos individuais nas classes de coleção, o *BookingManager* oferece métodos como `CreateCompleteOwner()`, `UpdateReservation()` e `CancelReservation()`, abstraindo a complexidade interna.

Este padrão promove uma maior modularidade e escalabilidade do sistema, permitindo a introdução de novas funcionalidades sem impactar a interface externa.

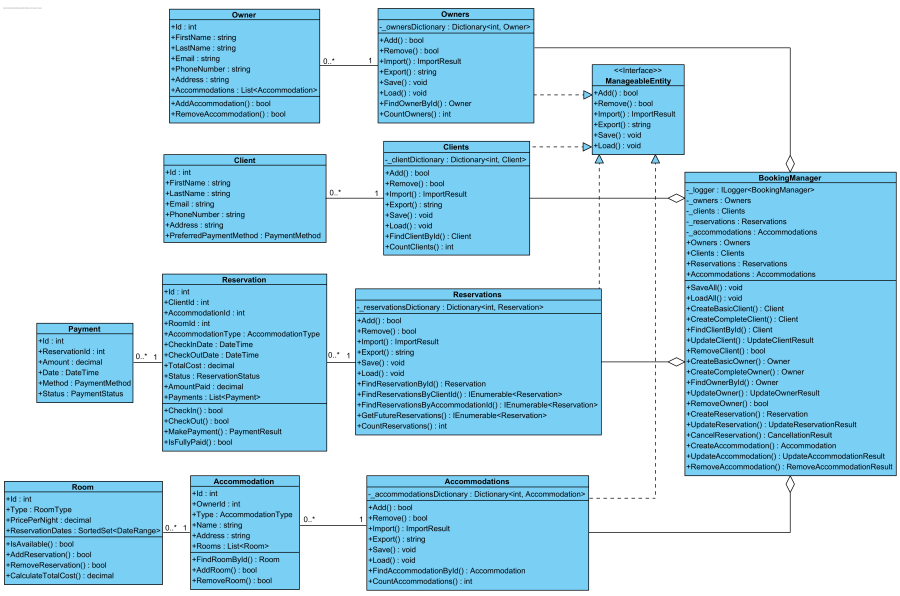

Fig. 1: Diagrama de Classes do Sistema de Gestão de Alojamentos Turísticos

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 (*unsigned*) 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 *Owner*, *Client*, *Accommodation* e

Reservation) são definidas com propriedades e comportamentos relacionados aos dados que representam. Estes modelos são utilizados frequentemente para mapear as entidades que serão manipuladas pela aplicação, sendo a base para as operações de armazenamento de dados.

4.4 Repositories

A camada de *Repositories* no nosso sistema é responsável por guardar e organizar listas de objetos de diferentes tipos, como Proprietários, 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 todos os processos complexos de dados.

4.6 Utilities

A camada de *Utilities* contém classes e métodos auxiliares que oferecem funcionalidades que podem ser usadas em diferentes partes do sistema. As *Utilities* ajudam a manter o código limpo e reutilizável, evitando a duplicação de lógica em várias partes da aplicação.

4.7 Bibliotecas

A organização do código também foi complementada com a utilização de bibliotecas específicas, guardadas na pasta `Libraries`, para promover a separação de responsabilidades e a modularidade do sistema. Estas bibliotecas foram estruturadas para encapsular diferentes funcionalidades do sistema, permitindo uma gestão mais clara e eficiente das preocupações. Abaixo descrevemos as principais bibliotecas criadas:

- **SmartStay.Common**: Esta biblioteca foi criada para conter elementos comuns que são utilizados por todas as partes do sistema. Ela inclui definições de *enums*, *exceptions* e outros componentes genéricos que podem ser reutilizados de maneira transversal. Ao centralizar esses elementos na biblioteca `SmartStay.Common`, é possível reduzir a redundância de código e melhorar a consistência em diferentes partes da aplicação.

- **SmartStay.Core**: A biblioteca central do sistema, `SmartStay.Core`, agrupa os modelos, repositórios, serviços e utilitários. Esta biblioteca encapsula a lógica de negócio principal e os elementos estruturais do sistema, assegurando uma clara separação entre os dados e as operações que os manipulam. Por exemplo, todas as classes de modelo, como `Client` e `Reservation`, bem como os repositórios que armazenam listas de objetos, estão organizados nesta biblioteca.
- **SmartStay.IO**: A biblioteca `SmartStay.IO` é responsável por lidar com operações de entrada e saída, particularmente o gerenciamento de dados em formato *JSON*. Ela encapsula toda a lógica relacionada à leitura e escrita de informações, permitindo que o sistema realize a persistência de dados de maneira modular. Esta abordagem melhora a manutenção, pois mudanças na forma como os dados são armazenados ou carregados podem ser feitas exclusivamente nesta biblioteca, sem impactar outras partes do sistema.
- **SmartStay.Validation**: A biblioteca `SmartStay.Validation` foi projetada para centralizar toda a lógica de validação de dados do sistema. Ela inclui regras específicas para garantir a integridade dos dados inseridos e manipulados pelo sistema, como validações de datas e formatos de entrada. Este design promove a reutilização de lógica de validação em diferentes componentes, aumentando a consistência e facilitando futuras alterações ou expansões nas regras de validação.

A utilização destas bibliotecas reflete um compromisso com a separação de preocupações (*Separation of Concerns*), promovendo uma arquitetura limpa, extensível e de fácil manutenção. Cada biblioteca tem um propósito bem definido e pode ser atualizada ou substituída de forma independente, reduzindo a complexidade e os riscos associados a alterações no sistema.

5 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.

5.1 Camadas da Arquitetura

A arquitetura do sistema é composta pelas seguintes camadas principais:

- **Model**: Representa os dados do sistema e encapsula a lógica de negócios principal.
- **View**: Responsável pela apresentação visual e interação com o utilizador.
- **Controller**: Atua como intermediário entre a interface de utilizador e a lógica de negócios, processando interações e executando as operações necessárias.
- **Repository e Service**: Fornecem suporte adicional à lógica do *Model*, gerindo persistência de dados e operações de negócios específicas.

Esta estrutura modular assegura que cada componente tem um papel claramente definido, facilitando a manutenção e evolução do sistema.

5.2 Escolha do Padrão MVC

O padrão MVC foi escolhido pelas suas características de modularidade e flexibilidade, ideais para sistemas com alta interação entre utilizadores, dados e regras de negócio. A separação entre Model, View e Controller garante que as alterações em uma camada não impactam diretamente as demais, promovendo robustez no desenvolvimento.

As vantagens do MVC incluem:

- **Facilidade de manutenção**: Cada camada pode ser atualizada ou substituída independentemente.
- **Escalabilidade**: O sistema pode ser expandido de forma simples.
- **Testabilidade**: As regras de negócio e os dados podem ser testados separadamente da interface de utilizador.

5.3 Desenvolvimento Progressivo

O desenvolvimento foi conduzido de forma progressiva, com as camadas das regras de negócios implementadas antes da camada de apresentação:

1. **Camadas Fundamentais**: O desenvolvimento inicial focou-se em *Model*, *Repository*, *Service*, *Utility* e *Validation*. Estas camadas foram projetadas, implementadas e testadas de forma independente, garantindo uma base sólida para o sistema.
2. **Camada de Interface (View)**: Após a conclusão das camadas fundamentais, a interface de utilizador foi integrada, utilizando a lógica já validada para proporcionar uma experiência intuitiva aos utilizadores.

Este fluxo de desenvolvimento garantiu que as regras de negócio estivessem bem estruturadas e funcionais antes da criação da interface, permitindo uma evolução coesa do sistema.

6 Estruturas de Dados Utilizadas

6.1 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.

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

Amostra de Código 2: 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.

6.2 Utilização de SortedSet

Para a classe *Room*, foi utilizada a estrutura *SortedSet<DateRange>* para guardar os intervalos de datas das reservas. O *SortedSet* é uma coleção que mantém os elementos em ordem, o que permite realizar buscas e consultas de maneira eficiente. A principal vantagem do *SortedSet* sobre outras estruturas, como listas ou dicionários, é que ele mantém a ordem dos elementos automaticamente e proporciona operações eficientes de inserção, remoção e busca.

O *SortedSet* é baseado numa árvore binária balanceada, tipicamente uma árvore vermelho-preta (*Red-Black Tree*), que garante um tempo de execução logarítmico para as operações básicas: inserção, remoção e busca, ou seja, $O(logn)$. Isto acontece porque, ao usar uma árvore binária balanceada, o *SortedSet* mantém o conjunto de dados ordenados, sem a necessidade de reordenar os elementos manualmente a cada operação.

Abaixo está um exemplo de como o sistema verifica a disponibilidade de um alojamento utilizando o *SortedSet<DateRange>*:

```
1 public bool IsAvailable(DateTime startDate, DateTime endDate,
      DateRange? existingReservationRange = null)
2 {
3         if (endDate <= startDate)
4         throw new ArgumentException("End date must be after the start
              date.");
```

```
5
6           var newReservation = new DateRange(startDate, endDate);
7
8           // Get potential conflicting reservations within the
                requested range
9           var potentialConflicts = _reservationDates.GetViewBetween(
10          new DateRange(DateTime.MinValue, startDate), // All
                reservations that end before the start
11          new DateRange(DateTime.MaxValue, endDate)    // All
                reservations that start after the end
12          );
13
14          // Check if there are any overlapping reservations
15          foreach (var existingReservation in potentialConflicts)
16          {
17                  // Skip the existing reservation if it's the one
                        we're trying to modify
18                  if
                        (existingReservation.Equals(existingReservationRange))
19                  {
20                          continue; // Skip this reservation as it's
                                the one we're modifying
21                  }
22
23                  // An overlap occurs if the start date is before the
                        end date, and the end date is after the start date
24                  if ((newReservation.Start < existingReservation.End)
                        && (newReservation.End >
                        existingReservation.Start))
25                  {
26                          return false; // There's an overlap, so the
                                accommodation is not available
27                  }
28          }
29
30          return true; // No overlap found, accommodation is available
31  }
```

Amostra de Código 3: Verificação da Disponibilidade de um Alojamento

O uso do *SortedSet* permite que o sistema verifique rapidamente se há conflitos de datas para uma nova reserva. Quando uma nova reserva é solicitada, o método `GetViewBetween` é utilizado para obter apenas as reservas que podem potencialmente se sobrepor com a nova reserva, o que torna o processo de verificação muito mais eficiente.

A principal vantagem do *SortedSet* é que ele mantém os dados ordenados automaticamente, e as operações de busca e inserção são realizadas em tempo $O(logn)$, o que é muito mais eficiente do que um algoritmo de busca linear que

teria um custo de $O(n)$. Em sistemas com muitas reservas, isto resulta num ganho de desempenho significativo.

Além disso, a estrutura *SortedSet* oferece uma forma eficiente de lidar com a inserção de novos intervalos de datas, pois a árvore binária balanceada garante que as operações de inserção e remoção também sejam realizadas de maneira logarítmica. Isso é fundamental para garantir que o sistema permaneça escalável à medida que o número de reservas aumenta.

6.3 Vantagens das Estruturas de Dados Escolhidas

As escolhas do *Dictionary* e *SortedSet* refletem uma análise cuidadosa das necessidades do sistema, como a eficiência de acesso, a escalabilidade e a organização dos dados. Abaixo, destacam-se as principais vantagens dessas escolhas:

- **Eficiência de Acesso**: O *Dictionary* permite acesso rápido às entidades, como clientes e reservas, através de chaves únicas. Esta característica é fundamental para garantir o bom desempenho do sistema, especialmente quando precisa de lidar com grandes volumes de dados.
- **Escalabilidade**: O uso do *Dictionary* para guardar entidades e do *SortedSet* para as datas das reservas garante um sistema escalável. À medida que o número de clientes ou reservas aumenta, o desempenho mantém-se estável devido à eficiência destas estruturas.
- **Verificação Rápida de Conflitos de Datas**: O *SortedSet* facilita a verificação de conflitos de datas de reservas, graças à sua capacidade de manter os dados ordenados automaticamente. A combinação com a operação `GetViewBetween` reduz significativamente o tempo de busca e garante uma execução eficiente mesmo com um número elevado de reservas.

Estas escolhas refletem uma análise cuidadosa dos requisitos de desempenho e eficiência do sistema, buscando sempre o equilíbrio entre a simplicidade na implementação o alto desempenho. Isto garante um sistema não apenas eficaz, mas também fácil de manter e escalar ao longo do tempo.

7 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.

7.1 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 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.

7.2 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 4: 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 5: 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.

8 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.

8.1 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 public class ValidationException : Exception
2 {
3         public ValidationErrorCode ErrorCode { get; }
4
5         public ValidationException(ValidationErrorCode errorCode) :
              base(ValidationErrorMessages.GetErrorMessage(errorCode))
6         {
7                 ErrorCode = errorCode;
8         }
9 }
```

Amostra de Código 6: Classe ValidationException

Como pode ser visto, 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.

8.2 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 está um exemplo simplificado desta enumeração:

```
1 namespace SmartStay.Validation
2 {
3         public enum ValidationErrorCode
4         {
5                 InvalidName = 1001,
```

```
6                   InvalidEmail = 1002,
7                   InvalidPhoneNumber = 1003
8           }
9   }
```

Amostra de Código 7: Enumeração ValidationErrorCode

8.3 Mensagens de Erro Dinâmicas

A classe `ValidationErrorMessages` é responsável por buscar as mensagens de erro localizadas a partir de ficheiros de recursos. Em vez de guardar as mensagens diretamente no código, as mensagens são mantidas em ficheiros de recursos (.resx), o que facilita a tradução e manutenção das mensagens de erro, especialmente em sistemas com muitos idiomas.

Veja abaixo um exemplo simplificado da classe *ValidationErrorMessages*:

```
1   public static class ValidationErrorMessages
2   {
3           private static readonly ResourceManager _resourceManager =
                new ResourceManager(
4           "SmartStay.Validation.Resources.ValidationMessages",
                typeof(ValidationErrorMessages).Assembly);
5
6           public static string GetErrorMessage(ValidationErrorCode
                errorCode)
7           {
8                   return
                        _resourceManager.GetString(errorCode.ToString(),
                        CultureInfo.CurrentCulture) ??
9                   "Unknown validation error.";
10          }
11  }
```

Amostra de Código 8: Classe ValidationErrorMessages

Desta forma, quando uma exceção de validação é gerada, a mensagem correspondente ao código de erro é obtida dinamicamente a partir do ficheiro de recursos com base na cultura atual. Caso o código de erro não esteja mapeado, uma mensagem padrão "Unknown validation error." *(Erro de validação desconhecido)* será devolvida, garantindo que todas as exceções são tratadas de forma consistente e traduzidas corretamente para o idioma apropriado.

8.4 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.

9 Tratamento de Logs

O tratamento de logs é uma componente essencial para monitorizar o funcionamento do sistema, detetar erros e analisar o comportamento das operações realizadas. Nesta aplicação, utilizou-se a biblioteca *Microsoft.Extensions.Logging*, que oferece uma solução integrada e extensível para registo de eventos.

A abordagem adotada inclui a injeção direta do *logger* através de dependência, garantindo que todas as classes que necessitam de registo de eventos podem utilizá-lo de forma centralizada e consistente.

9.1 Injeção do Logger

A injeção do logger é realizada diretamente no construtor da classe, utilizando a interface `ILogger<T>`, onde `T` é o tipo da classe onde o logger será utilizado. Este mecanismo permite associar o logger ao contexto específico da classe, proporcionando mensagens de log detalhadas e informativas.

O exemplo abaixo demonstra como o logger é injetado na classe `BookingManager`:

```
1 public BookingManager(ILogger<BookingManager> logger)
2 {
3         _logger = logger ?? throw new
              ArgumentNullException(nameof(logger));
4         _logger.LogInformation("BookingManager initialized.");
5 }
```

Amostra de Código 9: Injeção do Logger no Construtor

No exemplo acima, o logger é atribuído a uma variável privada e utilizado para registar uma mensagem de informação na inicialização da classe.

9.2 Exemplo de Utilização

O logger é usado para registar informações importantes durante a execução do sistema, tais como:

- **Mensagens de informação** (`LogInformation`), para registar o estado e progresso de operações.
- **Mensagens de erro** (`LogError`), para reportar exceções e falhas.
- **Mensagens de *debug*** (`LogDebug`), para análise detalhada durante o desenvolvimento.

Segue um exemplo do logger no código.

```
1  public Client CreateBasicClient(string firstName, string lastName,
       string email)
2  {
3          try
4          {
5                  // Log information before creating the client
6                  _logger.LogInformation(
7                  "Attempting to create a new client with Name:
                       {FirstName} {LastName}, Email: {Email}.",
                       firstName,
8                  lastName, email);
9
10                 // Create a new client
11                 Client client = new Client(firstName, lastName,
                       email); // May throw exception
12
13                 // Add client to the system
14                 _clients.Add(client);
15
16                 // Log success information
17                 _logger.LogInformation("Successfully created client:
                       {FirstName} {LastName}, Email: {Email}.",
                       firstName,
18                 lastName, email);
19
20                 // Return the created client
21                 return client;
22         }
23         catch (ValidationException ex)
24         {
25                 // Log the exception with details
26                 _logger.LogError(ex, "Error while creating client
                       with Name: {FirstName} {LastName}, Email:
                       {Email}.",
27                 firstName, lastName, email);
28
29                 // Throw a new exception with more context
30                 throw new ClientCreationException("An error occurred
                       while creating the client due to invalid input.",
                       ex);
31         }
32 }
```

Amostra de Código 10: Exemplo de Utilização do Logger

Neste exemplo:

- Antes de criar o cliente, é registada uma mensagem de informação com os detalhes fornecidos.
- Em caso de sucesso, uma mensagem de confirmação é registada.

– Se ocorrer uma exceção, é registada como erro, juntamente com os detalhes do contexto.

9.3 Benefícios do Uso de Logs

– **Monitorização**: Os logs permitem monitorizar o estado da aplicação em tempo real, registando operações críticas e sucessos.
– **Diagnóstico de erros**: As mensagens de erro detalhadas ajudam na identificação e resolução de problemas rapidamente.
– **Auditoria**: O histórico de operações registado pelos logs permite analisar as ações realizadas.
– **Facilidade de depuração**: Durante o desenvolvimento, as mensagens de *debug* podem fornecer informações detalhadas sobre o comportamento do sistema.

9.4 Estratégias de Log no Sistema

– **Níveis de log**: Configurar os níveis de log (`Information`, `Debug`, `Error`) para evitar excesso de registos e manter a relevância das mensagens.
– **Persistência**: Utilizar *providers* de log (ficheiros, bases de dados ou sistemas como o Serilog) para guardar logs persistentes, facilitando a análise a longo prazo.
– **Consistência**: Garantir que todas as classes seguem um padrão uniforme na utilização do logger, melhorando a legibilidade e a manutenção.

Com estas práticas, o sistema assegura um tratamento de logs robusto, contribuindo para a estabilidade, transparência e eficiência do desenvolvimento.

10 Persistência de Dados com Ficheiros

A persistência de dados desempenha um papel crucial em qualquer sistema de gestão. Neste projeto, foi adotada a utilização de ficheiros JSON para a importação e exportação de dados, complementada por ficheiros binários baseados em `Protocol Buffers` (Protobuf). O formato JSON foi escolhido para a importação e exportação devido à sua popularidade e facilidade de utilização, especialmente em aplicações web, móveis e *desktop*. Além disso, a sua fácil leitura torna-o ideal para a manipulação e *debugging* de dados de forma simples e eficiente. Por outro lado, o formato binário com Protobuf foi escolhido para guardar e carregar dados, garantindo eficiência e desempenho superiores em termos de armazenamento e velocidade de processamento.

Para a persistência binária, foi escolhido o `Protobuf`, em vez dos métodos tradicionais binários como o `BinaryFormatter`, que é obsoleto e apresenta vulnerabilidades de segurança. O `BinaryFormatter` tem riscos significativos na desserialização de dados, o que pode resultar em falhas de segurança, como execução remota de código. O `Protobuf`, por outro lado, oferece uma solução mais segura, eficiente e compacta para armazenar dados de forma binária. Além disso, é compatível com diversas linguagens e plataformas, o que facilita a integração com outros sistemas.

Portanto, enquanto o JSON é utilizado para a leitura e manipulação de dados de forma mais acessível, o `Protobuf` é usado para a gravação e carregamento eficiente de dados binários, garantindo melhor desempenho e menor tamanho de ficheiro. Este equilíbrio entre formatos textuais e binários permite que o sistema aproveite o melhor dos dois mundos: a simplicidade e flexibilidade do JSON para operações de importação/exportação e a eficiência do `Protobuf` para persistência binária.

10.1 Importação e Exportação de Dados em JSON

A importação e exportação de dados são funcionalidades essenciais para assegurar que a informação possa ser manipulada de forma eficaz em diferentes contextos. Neste projeto, optou-se pelo uso do formato JSON para estas operações, devido à sua legibilidade e ampla adoção em diversas plataformas. O formato JSON permite que os dados sejam facilmente lidos, editados e transferidos, sendo particularmente útil em ambientes que exigem interação com outras aplicações ou serviços.

Exemplo de Importação de Dados em JSON O método `Import` permite importar uma lista de objetos 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 seguinte apresenta um exemplo do método *Import* dos clientes:

```
1 public ImportResult 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           // Deserialize the data into a List<Client> instead of a
                single Client
9           var clients = JsonHelper.DeserializeFromJson<Client>(data) ??
10          throw new ArgumentException("Deserialized client data cannot
                be null", nameof(data));
11
12          int replacedCount = 0;
13          int importedCount = 0;
14
15          foreach (var client in clients)
16          {
17                  if (_clientDictionary.ContainsKey(client.Id))
18                  {
19                          replacedCount++;
20                  }
21                  else
22                  {
23                          importedCount++;
24                  }
25                  _clientDictionary[client.Id] = client; // Direct
                        insertion for efficiency
26          }
27
28          return new ImportResult { ImportedCount = importedCount,
                ReplacedCount = replacedCount };
29  }
```

Amostra de Código 11: 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.
- O método `Import` retorna um objeto `ImportResult`, que contém a quantidade de clientes importados e substituídos, proporcionando informações úteis sobre o processo de importação.

Exemplo de Exportação de Dados em JSON 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. O código do método é o seguinte:

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

Amostra de Código 12: Método Export para Exportar Clientes

O método `Export` converte os valores guardados 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.

Neste método, os valores do dicionário `_clientDictionary` são serializados diretamente para uma string JSON. O uso de `Enumerable.Empty<Client>()` garante que, caso não existam clientes no dicionário, uma lista vazia seja serializada, evitando problemas com a estrutura JSON.

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.

10.2 Persistência de Dados em Formato Binário com Protobuf

Embora o formato JSON seja simples e eficaz, em alguns casos, como sistemas com grandes volumes de dados ou requisitos de desempenho mais exigentes, pode ser vantajoso utilizar um formato binário mais compacto e eficiente. Nesse contexto, o `Protocol Buffers` (Protobuf) apresenta-se como uma alternativa poderosa.

Protobuf é um formato binário de serialização de dados desenvolvido pela Google, que permite uma codificação e decodificação mais rápida e compacta do que o JSON. Ao contrário do JSON, que é um formato de texto, o Protobuf

armazena os dados de forma binária, o que reduz o tamanho do ficheiro e melhora o desempenho de operações de leitura e escrita, especialmente quando lidamos com muitos dados.

Vantagens do Protobuf O uso do Protobuf para persistência de dados oferece várias vantagens:

- **Eficiência de Espaço**: Como o Protobuf é um formato binário, é muito mais compacto em comparação ao JSON, o que resulta em menos espaço de armazenamento.
- **Desempenho Superior**: O Protobuf oferece tempos de leitura e escrita mais rápidos devido à sua natureza binária, tornando-o adequado para sistemas com alto tráfego de dados.

Carregamento de Dados com Protobuf O método `Load` permite carregar uma coleção de clientes a partir de um ficheiro binário utilizando o formato `Protocol Buffers` (Protobuf). Caso ocorra algum erro durante o processo de leitura ou desserialização, são lançadas exceções específicas para diagnosticar o problema.

```
1  public void Load(string filePath)
2  {
3          try
4          {
5                  // Open the file stream for reading
6                  using (var fileStream = File.OpenRead(filePath))
7                  {
8                          // Deserialize the clients object from the
                               file
9                          var clients =
                               Serializer.Deserialize<Clients>(fileStream);
10
11                         // Copy the data from the deserialized object
                               to the current instance
12                         _clientDictionary.Clear();
13                         foreach (var client in
                               clients._clientDictionary)
14                         {
15                                 _clientDictionary[client.Key] =
                                       client.Value;
16                         }
17                 }
18         }
19         catch (IOException ioEx)
20         {
21                 throw new IOException("An error occurred while
                       loading the clients data.", ioEx);
22         }
```

```
23          catch (SerializationException serEx)
24          {
25                  throw new SerializationException("An error occurred
                        during deserialization while loading the clients
                        data.",
26                  serEx);
27          }
28  }
```

Amostra de Código 13: Método Load para Carregar Clientes com Protobuf

O método `Load` realiza as seguintes operações:

- Abre o ficheiro especificado para leitura, utilizando uma *stream* do ficheiro.
- Desserializa os dados utilizando `Protocol Buffers`, recriando o objeto de clientes.
- Substitui os dados atuais na instância com os dados carregados do ficheiro.
- Lança exceções detalhadas em caso de falha na leitura ou na desserialização dos dados.

Gravação de Dados com Protobuf O método `Save` permite guardar o estado atual da coleção de clientes num ficheiro binário utilizando o formato `Protocol Buffers` (Protobuf). Em caso de erro durante o processo de escrita ou serialização, são lançadas exceções adequadas.

```
1   public void Save(string filePath)
2   {
3           try
4           {
5                   // Prepare for serialization by copying the
                        dictionary contents to the temporary list
6                   PrepareForSerialization();
7
8                   // Open the file stream for saving the data to the
                        specified file
9                   using (var fileStream = File.Create(filePath))
10                  {
11                          // Serialize the clients object and write it
                                to the file stream
12                          Serializer.Serialize(fileStream, this);
13                  }
14          }
15          catch (IOException ioEx)
16          {
17                  throw new IOException("An error occurred while saving
                        the clients data.", ioEx);
18          }
19          catch (SerializationException serEx)
20          {
```

```
21                 throw new SerializationException("An error occurred
                       during serialization while saving the clients
                       data.",
22                 serEx);
23         }
24 }
```

Amostra de Código 14: Método Save para Guardar Clientes com Protobuf

O método `Save` realiza as seguintes operações:

- Prepara os dados para serialização, copiando os conteúdos do dicionário para uma estrutura temporária, caso necessário.
- Abre uma *stream* do ficheiro no caminho especificado para guardar os dados.
- Serializa os dados no formato `Protocol Buffers` e escreve-os no ficheiro.
- Lança exceções detalhadas em caso de falha na escrita ou na serialização dos dados.

Decoração de Classes para Protobuf Para utilizar o `Protocol Buffers` (Protobuf) como mecanismo de serialização, as classes a serem serializadas precisam de ser decoradas com atributos específicos fornecidos pela biblioteca. Estes atributos definem como os dados serão organizados e manipulados durante os processos de serialização e desserialização. No caso deste projeto, as classes como `Client` foram configuradas utilizando os atributos `[ProtoContract]` e `[ProtoMember]`.

```
1  [ProtoContract]
2  public class Client
3  {
4          [ProtoMember(1)]
5          readonly int _id; // ID of the client
6
7          [ProtoMember(2)]
8          string _firstName; // First name of the client
9
10         [ProtoMember(3)]
11         string _lastName; // Last name of the client
12
13         (...)
```

Amostra de Código 15: Classe Client Decorada para Protobuf

Atributos Principais

- `[ProtoContract]`: Este atributo marca a classe como serializável com Protobuf. Sem ele, a classe não será processada pelo mecanismo de Protobuf.
- `[ProtoMember(n)]`: Atributo utilizado para especificar a ordem dos campos durante a serialização. Cada membro deve ser associado a um número único que define sua posição na estrutura de dados serializada.

11 Testes Unitários

11.1 Introdução aos Testes Unitários

Os testes unitários são uma parte essencial do desenvolvimento de software porque garantem que as funções e os métodos tenham o correto comporamento. O principal objetivo é validar o comportamento de cada parte do sistema de forma isolada, o que permite identificar e corrigir erros de maneira eficaz. Para implementar estes testes, usei o framework `xUnit`, que é uma escolha popular no ecossistema .NET devido à sua simplicidade e integração fácil com outras ferramentas.

11.2 Razões para Utilizar Testes Unitários

Os testes unitários ajudam a assegurar a qualidade do código, fornecendo uma camada extra de segurança ao desenvolvimento. No meu projeto, a utilização de testes permitiu:

- **Detecção de erros**: Ao testar cada unidade de código de forma isolada, é possível identificar erros, evitando que se propaguem para outras partes do sistema.
- **Facilidade de refatoração**: Com testes automatizados, é possível refatorar o código com maior confiança, pois sabemos que os testes garantirão que as funcionalidades existentes não sejam comprometidas.
- **Documentação do comportamento esperado**: Os testes servem como uma forma de documentação do comportamento das funcionalidades, tornando o código mais compreensível.

11.3 Implementação dos Testes com xUnit

A escolha do `xUnit` foi motivada pela sua popularidade na comunidade .NET, bem como pela sua sintaxe simples e pela facilidade de integração com outros frameworks e ferramentas, como o `GitHub Actions` para CI/CD. Abaixo segue um exemplo de um teste unitário implementado com `xUnit`.

```
1  [Fact]
2  public void Constructor_ValidParameters_InitializesClient()
3  {
4          // Arrange
5          var firstName = "Enrique";
6          var lastName = "Rodrigues";
7          var email = "Enrique.Rodrigues@example.com";
8
9          // Act
10         var client = new Client(firstName, lastName, email);
11
12         // Assert
13         Assert.Equal(firstName, client.FirstName);
```

```
14          Assert.Equal(lastName, client.LastName);
15          Assert.Equal(email, client.Email);
16          Assert.Equal(PaymentMethod.None,
                client.PreferredPaymentMethod); // Default value
17          Assert.NotEqual(0, client.Id);
                                                // ID should be
                auto-assigned
18  }
```

Amostra de Código 16: Teste do Construtor da Classe Client

Este teste verifica o comportamento do construtor da classe `Client`, garantindo que os parâmetros fornecidos sejam corretamente atribuídos às propriedades da classe e que o valor do `ID` seja atribuído automaticamente.

11.4 Integração com CI/CD

A integração de testes unitários num pipeline CI/CD é uma prática recomendada para garantir que o código seja validado automaticamente em cada mudança ou atualização. No meu projeto, configurei um pipeline de integração contínua utilizando `GitHub Actions`, permitindo que os testes fossem executados automaticamente sempre que alterações fossem feitas no repositório.

O ficheiro `dotnet.yml`, que configura o pipeline de CI, é o seguinte:

```
1  name: .NET CI
2
3  on:
4  push:
5  branches:
6  - main
7  pull_request:
8  branches:
9  - main
10
11 jobs:
12 build-and-test:
13 runs-on: ubuntu-latest
14
15 steps:
16 - name: Checkout code
17 uses: actions/checkout@v3
18
19 - name: Setup .NET
20 uses: actions/setup-dotnet@v3
21 with:
22 dotnet-version: 8.0.x
23
24 - name: Restore dependencies
25 working-directory: ./SmartStay
26 run: dotnet restore SmartStay.sln
```

```
27
28 - name: Build solution
29 working-directory: ./SmartStay
30 run: dotnet build SmartStay.sln --no-restore --configuration Release
31
32 - name: Run tests
33 working-directory: ./SmartStay
34 run: dotnet test --no-build --verbosity normal --configuration Release
```

Amostra de Código 17: Configuração do Pipeline CI com GitHub Actions

Este ficheiro configura o pipeline para que, sempre que houver um `push` ou um `pull request` na branch `main`, o código seja compilado e os testes sejam executados de forma automática.

11.5 Vantagens da Integração Contínua e Testes Automatizados

Integrar os testes unitários com um pipeline CI/CD oferece várias vantagens:

- **Automação**: A execução dos testes é automática sempre que há uma alteração no código, garantindo que nada seja esquecido.
- **Detecção rápida de falhas**: Caso algum teste falhe, o pipeline falha imediatamente, permitindo que os desenvolvedores corrijam os problemas de forma rápida.
- **Consistência**: Os testes são executados nas mesmas condições em cada execução do pipeline, assegurando que o comportamento do sistema seja consistente ao longo do tempo.

A utilização de testes unitários combinada com um pipeline de CI/CD robusto oferece uma solução eficiente e confiável para garantir a qualidade e a estabilidade do código durante todo o ciclo de desenvolvimento.

12 Desenvolvimento da Interface

O desenvolvimento da interface da aplicação foi realizado utilizando a linguagem de programação *Dart* e a *framework Flutter*, uma combinação que oferece uma abordagem eficiente para a construção de interfaces de utilizador modernas e reativas. O *Flutter* é uma framework de *UI* de código aberto que permite o desenvolvimento de aplicações nativas para várias plataformas com uma única base de código.

12.1 Estado Atual da Aplicação

A aplicação desenvolvida encontra-se num estado preliminar, servindo como um exemplo de implementação prática do uso de controladores (*controllers*) e *web requests*. Não é uma aplicação completa, mas sim um modelo demonstrativo, cujo objetivo principal é ilustrar a interação básica entre uma interface de utilizador e um backend.

Por enquanto, a interface implementada permite executar operações básicas, como simular o login de um cliente ou realizar ações limitadas a partir de botões que acionam funções. A lógica é simplificada, focando-se em práticas fundamentais que podem ser expandidas para projetos mais robustos no futuro.

12.2 Interação com Controladores e HTTP Requests

Neste subcapítulo, detalha-se o funcionamento básico da interação entre a interface de utilizador e os controladores através de chamadas HTTP. A aplicação utiliza botões que chamam funções específicas, as quais recolhem dados a partir de controladores, realizam validações e chamam métodos para comunicação com o backend.

A seguir, apresenta-se um exemplo de implementação de um botão que simula o login de um cliente:

```
1  // Botão de exemplo - Iniciar Sessão como Cliente
2  Expanded(
3          child: ElevatedButton(
4          onPressed: () async {
5                  final clientCreated = await _handleCreateClient();
6                  if (clientCreated && context.mounted) {
7                          await context.router.replace(const
                               HomeRoute());
8                          }
9                  },
10         ),
11 );
```

Amostra de Código 18: Botão para iniciar sessão como cliente

O botão acima está configurado para chamar a função `_handleCreateClient()` ao ser pressionado. Esta função realiza as seguintes etapas:

- **Recolha de Dados**: Obtém os valores inseridos pelo utilizador nos controladores de texto.
- **Validação**: Verifica se os dados recolhidos são válidos (por exemplo, se nenhum campo obrigatório está vazio).
- **Pedido HTTP**: Envia os dados para o backend, utilizando a função `createBasicClient()`, que realiza a interação com o controlador.

Segue-se a implementação da função `_handleCreateClient()`, que realiza estas operações:

```
Future<bool> _handleCreateClient() async {
        // Obter valores dos controladores
        final firstName = _firstNameController.text;
        final lastName = _lastNameController.text;
        final email = _emailController.text.trim();

        // Validação básica dos dados de entrada
        if (firstName.isEmpty || lastName.isEmpty || email.isEmpty) {
                _showErrorMessage('Por favor, preencha todos os
                    campos.');
                return false;
        }

        // Chamar a função para criar o cliente
        final result = await createBasicClient(
                firstName: firstName,
                lastName: lastName,
                email: email,
        );

        if (result != null) {
                // Cliente criado com sucesso
                if (kDebugMode) {
                        print('Cliente criado com sucesso: $result');
                }
                return true;
        } else {
                // Falha ao criar o cliente
                return false;
        }
}
```

Amostra de Código 19: Função que processa os dados e faz a chamada HTTP

Nesta função, a chamada HTTP é realizada por `createBasicClient()`, que comunica diretamente com o controlador do backend. Esta abordagem modular permite que a lógica de validação e comunicação seja isolada em funções separadas, facilitando a manutenção e a escalabilidade do projeto.

Segue-se agora a implementação da função `createBasicClient()`, que faz o pedido HTTP para o backend. Ela é responsável por enviar os dados do cliente para a API e retornar a resposta do servidor:

```
Future<Map<String, dynamic>?> createBasicClient({
        required String firstName,
        required String lastName,
        required String email,
}) async {
        try {
                // Fazer uma requisição POST para o endpoint /client/basic
                final response = await _networkClient.post<Map<String, dynamic>>(
                '/client/basic',
                data: {
                        'firstName': firstName,
                        'lastName': lastName,
                        'email': email,
                },
                );

                // Verificar se a resposta indica sucesso
                if (response.statusCode == 201) {
                        // Retornar os dados do cliente recém-criado
                        return response.data;
                } else {
                        // Tratar códigos de status inesperados
                        _showErrorMessage('Código de status inesperado: ${response.statusCode}');
                        return null;
                }
        } on DioException catch (e) {
                if (e.response != null && e.response?.statusCode == 400) {
                        // Tratar erro 400 - Bad Request
                        final errorMessage =
                        e.response?.data['message'] ?? 'Erro desconhecido';
                        _showErrorMessage('Falha na criação do cliente: $errorMessage');
                        return null;
                } else {
                        // Tratar outros erros Dio (ex: problemas de rede)
                        _showErrorMessage('Erro de rede: ${e.message}');
                        return null;
                }
        } catch (e) {
```

```
39                  // Tratar outras exceções
40                  _showErrorMessage('Erro inesperado: $e');
41                  return null;
42          }
43  }
```

Amostra de Código 20: Função que realiza a requisição HTTP para criar um cliente

A função `createBasicClient()` faz um pedido `POST` para o endpoint `/client/basic`, enviando os dados do cliente (nome, sobrenome e e-mail) para o backend. Caso o pedido seja bem-sucedida (código de estado 201), ela retorna os dados do cliente criado. Se houver um erro (código de estado diferente de 201 ou problemas com a rede), a função trata o erro e mostra uma mensagem de erro.

Por enquanto, esta estrutura serve apenas como exemplo prático de como organizar a lógica em torno de controladores e chamadas HTTP, com o objetivo de criar uma base sólida para o desenvolvimento de aplicações mais completas e funcionais no futuro.

13 Conclusão

O desenvolvimento deste projeto foi concluído com sucesso, atendendo a todos os critérios e requisitos estabelecidos. A execução do trabalho foi feita com foco na qualidade, na documentação e na qualidade da solução desenvolvida.

Os testes realizados demonstraram que todas as funcionalidades funcionam conforme o esperado. O código foi estruturado de forma modular e respeitando boas práticas de desenvolvimento.

O relatório produzido seguiu um padrão claro e bem organizado, apresentando detalhadamente a solução desenvolvida, os processos implementados e os resultados obtidos. Os conceitos e decisões técnicas foram documentados de forma compreensível e completa.

A aplicação demonstrou alta qualidade, refletida no seguinte:

- **Qualidade do Código Produzido**: O código foi estruturado de forma clara e modular, com nomenclatura apropriada para ficheiros e respeito às boas práticas de desenvolvimento, incluindo o uso de bibliotecas (DLL) e conformidade com a norma CLS.
- **Organização e Implementação de Classes**: A implementação fez uso de interfaces, garantindo uma arquitetura flexível, modular e reutilizável.
- **Qualidade dos Algoritmos**: Os algoritmos desenvolvidos foram analisados quanto à sua eficiência teórica, considerando o seu comportamento em termos de complexidade *Big O*, e testados com rigor por meio de testes unitários para assegurar o seu correto funcionamento.
- **Estruturas de Dados e Tratamento de Exceções**: As estruturas de dados foram exploradas de forma eficiente, e o tratamento de exceções foi implementado para garantir a robustez do sistema.
- **Tratamento de Logs**: A aplicação implementa mecanismos adequados para registro de logs, facilitando a monitorização e resolução de problemas.
- **Persistência de Dados**: Foi implementada a persistência de dados através do uso de ficheiros JSON e binários, garantindo a integridade e acessibilidade das informações.
- **Programação por Camadas**: A aplicação segue uma arquitetura por camadas, como *MVC*, promovendo a escalabilidade e a separação de responsabilidades.
- **Exploração de Outras Valências**: Foram integrados conceitos como LINQ e a utilização do *framework* Flutter para a interface do utilizador, demonstrando a aplicação de tecnologias modernas e eficientes.
- **Configuração de Pipelines CI/CD**: A automação de testes e implantação contínua foi implementada com sucesso, utilizando *Github Actions*, garantindo qualidade e eficiência no fluxo de desenvolvimento.

Com este trabalho, consegui atender a todos os critérios de avaliação definidos. A solução final demonstra não apenas a aplicação prática de conceitos teóricos, mas também uma atenção elevada aos detalhes.

Em resumo, os objetivos definidos foram todos alcançados e foi desenvolvido um sistema eficiente para a gestão de alojamentos turísticos utilizando o paradigma de programação orientada a objetos.

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 17 de dezembro de 2024.