

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 registo, 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 registo 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: registo, consultas, reservas, check-in, clientes, alojamentos

Índice

Gestão de Alojamentos Turísticos	1
<i>Enrique George Rodrigues N^o 28602</i>	
1 Introdução	1
1.1 Contexto e Motivação	1
1.2 Estrutura de Dados e Persistência	1
1.3 Boas Práticas de Programação	1
1.4 Objetivos e Organização do Documento	1
2 Padrões e Práticas de Programação	2
2.1 Convenções de Nomeação	2
2.2 Regras de Formatação	2
2.3 Práticas de Programação	2
2.4 Controlo de Versões	3
2.5 Documentação	3
2.6 Padrões de Design Utilizados	3
2.7 Funções Lambda	4
3 Diagrama de Classes	5
3.1 Introdução ao Sistema	5
3.2 Classes Principais	5
3.3 Processamento de Pagamentos	5
3.4 BookingManager e a Interface ManageableEntity	6
3.5 Padrão de Face (Facade Pattern)	6
4 Organização do Código	7
4.1 Nomeação de Ficheiros	7
4.2 Conformidade com a CLS	7
4.3 Models	7
4.4 Repositories	8
4.5 Services	8
4.6 Utilities	8
4.7 Bibliotecas	8
5 Arquitetura em Camadas (Layered Architecture)	10
5.1 Camadas da Arquitetura	10
5.2 Escolha do Padrão MVC	10
5.3 Desenvolvimento Progressivo	10
6 Estruturas de Dados Utilizadas	12
6.1 Utilização de Dictionary	12
6.2 Utilização de SortedSet	12
6.3 Vantagens das Estruturas de Dados Escolhidas	14
7 Uso de LINQ	15
7.1 Vantagens de LINQ	15
7.2 Utilização de LINQ no Código	15
8 Tratamento de Exceções	17

8.1	Estrutura das Exceções	17
8.2	Definição dos Códigos de Erro	17
8.3	Mensagens de Erro Dinâmicas	18
8.4	Benefícios da Abordagem	18
9	Tratamento de Logs	20
9.1	Injeção do Logger	20
9.2	Exemplo de Utilização	20
9.3	Benefícios do Uso de Logs	22
9.4	Estratégias de Log no Sistema	22
10	Persistência de Dados com Ficheiros	23
10.1	Importação e Exportação de Dados em JSON	23
	Exemplo de Importação de Dados em JSON	23
	Exemplo de Exportação de Dados em JSON	25
	Validação e Exceções na Persistência de Dados	25
10.2	Persistência de Dados em Formato Binário com Protobuf	25
	Vantagens do Protobuf	26
	Carregamento de Dados com Protobuf	26
	Gravação de Dados com Protobuf	27
	Decoração de Classes para Protobuf	28
11	Testes Unitários	29
11.1	Introdução aos Testes Unitários	29
11.2	Razões para Utilizar Testes Unitários	29
11.3	Implementação dos Testes com xUnit	29
11.4	Integração com CI/CD	30
11.5	Vantagens da Integração Contínua e Testes Automatizados	31
12	Desenvolvimento da Interface	32
12.1	Estado Atual da Aplicação	32
12.2	Interação com Controladores e HTTP Requests	32
13	Conclusão	36

Amostras de Código

1	Getter e Setter com Função Lambda	4
2	Uso de Dictionary	12
3	Verificação da Disponibilidade de um Alojamento	12
4	Encontra Reservas por ID de Cliente	15
5	Encontra Reservas por ID de Alojamento	16
6	Classe ValidationException	17
7	Enumeração ValidationErrorCode	17
8	Classe ValidationErrorMessages	18
9	Injeção do Logger no Construtor	20
10	Exemplo de Utilização do Logger	21
11	Método Import para Importar Clientes	23
12	Método Export para Exportar Clientes	25
13	Método Load para Carregar Clientes com Protobuf	26
14	Método Save para Guardar Clientes com Protobuf	27
15	Classe Client Decorada para Protobuf	28
16	Teste do Construtor da Classe Client	29
17	Configuração do Pipeline CI com GitHub Actions	30
18	Botão para iniciar sessão como cliente	32
19	Função que processa os dados e faz a chamada HTTP	33
20	Função que realiza a requisição HTTP para criar um cliente	34

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 registo 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:** Seguem o padrão `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 registrar 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.

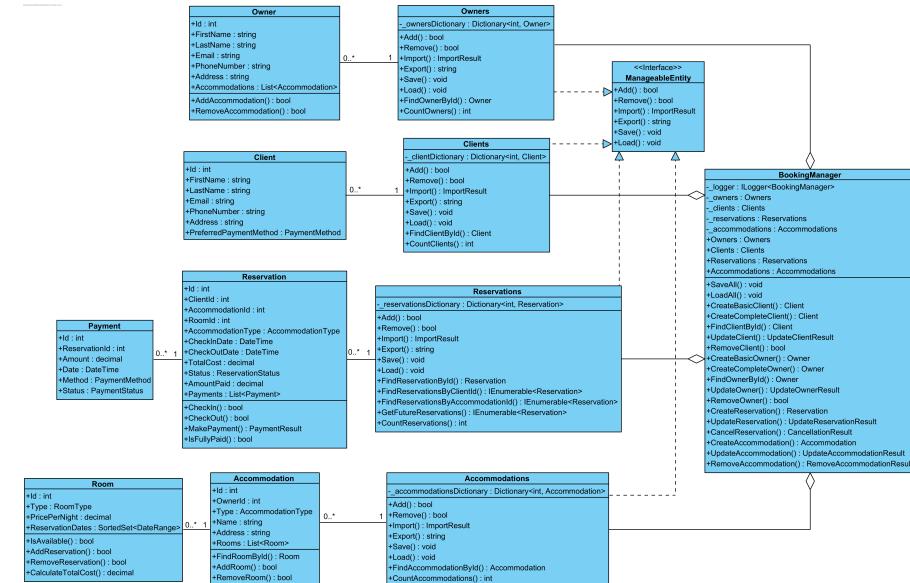


Fig. 1: Diagrama de Classes do Sistema de Gestão de Alojamentos Turísticos

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 naveabilidade 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 gera 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, $O(\log n)$. 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     var newReservation = new DateRange(startDate, endDate);
6
7     // Get potential conflicting reservations within the
8     // requested range
9     var potentialConflicts = _reservationDates.GetViewBetween(
10        new DateRange(DateTime.MinValue, startDate), // All
11        reservations that end before the start
12        new DateRange(DateTime.MaxValue, endDate)    // All
13        reservations that start after the end
14    );
15
16    // Check if there are any overlapping reservations
17    foreach (var existingReservation in potentialConflicts)
18    {
19        // Skip the existing reservation if it's the one
20        // we're trying to modify
21        if
22            (existingReservation.Equals(existingReservationRange))
23        {
24            continue; // Skip this reservation as it's
25            the one we're modifying
26        }
27
28        // An overlap occurs if the start date is before the
29        // end date, and the end date is after the start date
30        if ((newReservation.Start < existingReservation.End)
31            && (newReservation.End >
32            existingReservation.Start))
33        {
34            return false; // There's an overlap, so the
35            accommodation is not available
36        }
37    }
38
39    return true; // No overlap found, accommodation is available
40
41 }
```

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 $O(\log n)$, 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 e 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
2     clientId)
3 {
4     return _reservationDictionary.Values.Where(r => r.ClientId ==
5         clientId);
6 }
```

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.

```
1 public IEnumerable<Reservation> FindReservationsByAccommodationId(int
2     accommodationId)
3 {
4     return _reservationDictionary.Values.Where(r =>
5         r.AccommodationId == accommodationId);
6 }
```

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 `ValidationErrorMessage`. 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) :
6         base(ValidationErrorMessage.GetErrorMessage(errorCode))
7     {
8         ErrorCode = errorCode;
9     }

```

Amostra de Código 6: Classe ValidationException

Como pode ser visto, ao receber um `ValidationErrorCode`, a classe `ValidationException` utiliza o método `GetErrorMessage` da classe `ValidationErrorMessage` 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 `ValidationErrorMessage`s 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 `ValidationErrorMessage`s:

```

1 public static class ValidationErrorMessage
2 {
3     private static readonly ResourceManager _resourceManager =
4         new ResourceManager(
5             "SmartStay.Validation.Resources.ValidationMessages",
6             typeof(ValidationErrorMessage).Assembly);
7
8     public static string GetErrorMessage(ValidationErrorCode
9         errorCode)
10    {
11        return
12            _resourceManager.GetString(errorCode.ToString(),
13                CultureInfo.CurrentCulture) ??
14            "Unknown validation error.";
15    }
16 }

```

Amostra de Código 8: Classe ValidationErrorMessage

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(ILocator<BookingManager> logger)
2 {
3     _logger = logger ?? throw new
4         ArgumentNullException(nameof(logger));
5     _logger.LogInformation("BookingManager initialized.");

```

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 registrar informações importantes durante a execução do sistema, tais como:

- **Mensagens de informação** (*LogInformation*), para registrar 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,
2                                 string email)
3 {
4     try
5     {
6         // Log information before creating the client
7         _logger.LogInformation(
8             "Attempting to create a new client with Name:
9                 {FirstName} {LastName}, Email: {Email}.",
10                firstName,
11                lastName, email);
12
13         // Create a new client
14         Client client = new Client(firstName, lastName,
15                                     email); // May throw exception
16
17         // Add client to the system
18         _clients.Add(client);
19
20         // Log success information
21         _logger.LogInformation("Successfully created client:
22             {FirstName} {LastName}, Email: {Email}.",
23             firstName,
24             lastName, email);
25
26         // Return the created client
27         return client;
28     }
29     catch (ValidationException ex)
30     {
31         // Log the exception with details
32         _logger.LogError(ex, "Error while creating client
33             with Name: {FirstName} {LastName}, Email:
34                 {Email}.",
35             firstName, lastName, email);
36
37         // Throw a new exception with more context
38         throw new ClientCreationException("An error occurred
39             while creating the client due to invalid input.",
40             ex);
41     }
42 }

```

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 regtos 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 selecionado para a importação e exportação devido à sua popularidade e facilidade de utilização, especialmente em aplicações web, móveis e *desktop*. Além disso, a sua legibilidade elevada 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 de ambos os 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
6             empty", nameof(data));
7     }
8
9     // Deserialize the data into a List<Client> instead of a
10    // single Client
11    var clients = JsonHelper.DeserializeFromJson<Client>(data) ??
12    throw new ArgumentException("Deserialized client data cannot
13        be null", nameof(data));
14
15    int replacedCount = 0;
16    int importedCount = 0;
17
18    foreach (var client in clients)
19    {
20        if (_clientDictionary.ContainsKey(client.Id))
21        {
22            replacedCount++;
23        }
24        else
25        {
26            importedCount++;
27        }
28        _clientDictionary[client.Id] = client; // Direct
29        insertion for efficiency
30    }
31
32    return new ImportResult { ImportedCount = importedCount,
33        ReplacedCount = replacedCount };
34 }
```

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 ??
4         Enumerable.Empty<Client>());

```

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, ele 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
9             // file
10            var clients =
11                Serializer.Deserialize<Clients>(fileStream);
12
13            // Copy the data from the serialized object
14            // to the current instance
15            _clientDictionary.Clear();
16            foreach (var client in
17                clients._clientDictionary)
18            {
19                _clientDictionary[client.Key] =
20                    client.Value;
21            }
22        }
23    catch (IOException ioEx)
24    {
25        throw new IOException("An error occurred while
26            loading the clients data.", ioEx);
27    }
28}
```

```

23         catch (SerializationException serEx)
24     {
25         throw new SerializationException("An error occurred
26             during deserialization while loading the clients
27             data.", serEx);
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.
- Desserailiza 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
6         // dictionary contents to the temporary list
7         PrepareForSerialization();
8
9         // Open the file stream for saving the data to the
10        // specified file
11        using (var fileStream = File.Create(filePath))
12        {
13            // Serialize the clients object and write it
14            // to the file stream
15            Serializer.Serialize(fileStream, this);
16        }
17        catch (IOException ioEx)
18        {
19            throw new IOException("An error occurred while saving
20                the clients data.", ioEx);
21        }
22        catch (SerializationException serEx)
23        {

```

```

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

```

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 comportamento. 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,
17         client.PreferredPaymentMethod); // Default value
18     Assert.NotEqual(0, client.Id);
19         // ID should be
20         auto-assigned
21 }

```

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
8                     HomeRoute());
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 controadores 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:

```

1 Future<bool> _handleCreateClient() async {
2     // Obter valores dos controadores
3     final firstName = _firstNameController.text;
4     final lastName = _lastNameController.text;
5     final email = _emailController.text.trim();
6
7     // Validação básica dos dados de entrada
8     if (firstName.isEmpty || lastName.isEmpty || email.isEmpty) {
9         _showErrorMessage('Por favor, preencha todos os
10            campos.');
11        return false;
12    }
13
14    // Chamar a função para criar o cliente
15    final result = await createBasicClient(
16        firstName: firstName,
17        lastName: lastName,
18        email: email,
19    );
20
21    if (result != null) {
22        // Cliente criado com sucesso
23        if (kDebugMode) {
24            print('Cliente criado com sucesso: $result');
25        }
26        return true;
27    } else {
28        // Falha ao criar o cliente
29        return false;
30    }

```

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:

```

1 Future<Map<String, dynamic>?> createBasicClient({
2     required String firstName,
3     required String lastName,
4     required String email,
5 }) async {
6     try {
7         // Fazer uma requisição POST para o endpoint
8         // /client/basic
9         final response = await
10            _networkClient.post<Map<String, dynamic>>(
11                '/client/basic',
12                data: {
13                    'firstName': firstName,
14                    'lastName': lastName,
15                    'email': email,
16                },
17            );
18
19         // Verificar se a resposta indica sucesso
20         if (response.statusCode == 201) {
21             // Retornar os dados do cliente recém-criado
22             return response.data;
23         } else {
24             // Tratar códigos de status inesperados
25             _showErrorMessage('Código de status
26             inesperado: ${response.statusCode}');
27             return null;
28         }
29     } on DioException catch (e) {
30         if (e.response != null && e.response?.statusCode ==
31             400) {
32             // Tratar erro 400 - Bad Request
33             final errorMessage =
34                 e.response?.data['message'] ?? 'Erro
35                 desconhecido';
36             _showErrorMessage('Falha na criação do
37                 cliente: $errorMessage');
38             return null;
39         } else {
40             // Tratar outros erros Dio (ex: problemas de
41             rede)
42             _showErrorMessage('Erro de rede:
43                 ${e.message}');
44             return null;
45         }
46     } catch (e) {
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
287
288
289
289
290
291
292
293
294
295
296
297
297
298
299
299
300
301
302
303
304
305
306
307
308
309
309
310
311
312
313
314
315
316
317
318
319
319
320
321
322
323
324
325
326
327
328
329
329
330
331
332
333
334
335
336
337
338
339
339
340
341
342
343
344
345
346
347
348
349
349
350
351
352
353
354
355
356
357
358
359
359
360
361
362
363
364
365
366
367
368
369
369
370
371
372
373
374
375
376
377
378
379
379
380
381
382
383
384
385
386
387
387
388
389
389
390
391
392
393
394
395
396
397
398
399
399
400
401
402
403
404
405
406
407
408
409
409
410
411
412
413
414
415
416
417
418
419
419
420
421
422
423
424
425
426
427
428
429
429
430
431
432
433
434
435
436
437
438
439
439
440
441
442
443
444
445
446
447
448
449
449
450
451
452
453
454
455
456
457
458
459
459
460
461
462
463
464
465
466
467
468
469
469
470
471
472
473
474
475
476
477
478
479
479
480
481
482
483
484
485
486
487
487
488
489
489
490
491
492
493
494
495
496
497
497
498
499
499
500
501
502
503
504
505
506
507
508
509
509
510
511
512
513
514
515
516
517
518
519
519
520
521
522
523
524
525
526
527
528
529
529
530
531
532
533
534
535
536
537
538
539
539
540
541
542
543
544
545
546
547
548
549
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
687
688
689
689
690
691
692
693
694
695
696
697
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
787
788
789
789
790
791
792
793
794
795
796
797
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
878
879
880
881
882
883
884
885
886
887
887
888
889
889
890
891
892
893
894
895
896
897
897
898
899
899
900
901
902
903
904
905
906
907
908
909
909
910
911
912
913
914
915
916
917
917
918
919
920
921
922
923
924
925
926
927
928
929
929
930
931
932
933
934
935
936
937
938
939
939
940
941
942
943
944
945
946
947
948
948
949
950
951
952
953
954
955
956
957
958
958
959
960
961
962
963
964
965
966
967
968
969
969
970
971
972
973
974
975
976
977
978
978
979
980
981
982
983
984
985
986
987
987
988
989
989
990
991
992
993
994
995
996
997
998
999
999
1000
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1087
1088
1089
1089
1090
1091
1092
1093
1094
1095
1095
1096
1097
1097
1098
1099
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1177
1178
1179
1179
1180
1181
1182
1183
1184
1185
1186
1186
1187
1188
1188
1189
1190
1191
1192
1193
1194
1195
1195
1196
1197
1197
1198
1199
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1237
1238
1239
1239
1240
1241
1242
1243
1244
1245
1246
1247
1247
1248
1249
1249
1250
1251
1252
1253
1254
1255
1256
1257
1257
1258
1259
1259
1260
1261
1262
1263
1264
1265
1266
1266
1267
1268
1268
1269
1270
1271
1272
1273
1274
1275
1275
1276
1277
1277
1278
1279
1279
1280
1281
1282
1283
1284
1285
1285
1286
1287
1287
1288
1289
1289
1290
1291
1292
1293
1294
1295
1295
1296
1297
1297
1298
1299
1299
1300
1301
1302
1303
1304
1305
1306
1307
1307
1308
1309
1309
1310
1311
1312
1313
1314
1315
1316
1316
1317
1318
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1327
1328
1329
1329
1330
1331
1332
1333
1334
1335
1336
1337
1337
1338
1339
1339
1340
1341
1342
1343
1344
1345
1346
1346
1347
1348
1348
1349
1350
1351
1352
1353
1354
1355
1356
1356
1357
1358
1358
1359
1360
1361
1362
1363
1364
1365
1365
1366
1367
1367
1368
1369
1369
1370
1371
1372
1373
1374
1375
1375
1376
1377
1377
1378
1379
1379
1380
1381
1382
1383
1384
1385
1385
1386
1387
1387
1388
1389
1389
1390
1391
1392
1393
1394
1394
1395
1396
1396
1397
1398
1398
1399
1400
1401
1402
1403
1403
1404
1405
1405
1406
1407
1407
1408
1409
1409
1410
1411
1412
1413
1414
1415
1415
1416
1417
1417
1418
1419
1419
1420
1421
1422
1423
1424
1425
1425
1426
1427
1427
1428
1429
1429
1430
1431
1432
1433
1434
1435
1435
1436
1437
1437
1438
1439
1439
1440
1441
1442
1443
1444
1445
1445
1446
1447
1447
1448
1449
1449
1450
1451
1452
1453
1454
1455
1455
1456
1457
1457
1458
1459
1459
1460
1461
1462
1463
1464
1465
1465
1466
1467
1467
1468
1469
1469
1470
1471
1472
1473
1474
1474
1475
1476
1476
1477
1478
1478
1479
1480
1481
1482
1483
1484
1484
1485
1486
1486
1487
1488
1488
1489
1490
1491
1492
1493
1494
1494
1495
1496
1496
1497
1498
1498
1499
1500
1501
1502
1503
1503
1504
1505
1505
1506
1507
1507
1508
1509
1509
1510
1511
1512
1513
1514
1514
1515
1516
1516
1517
1518
1518
1519
1520
1521
1522
1523
1524
1524
1525
1526
1526
1527
1528
1528
1529
1530
1531
1532
1533
1534
1534
1535
1536
1536
1537
1538
1538
1539
1540
1541
1542
1543
1544
1544
1545
1546
1546
1547
1548
1548
1549
1550
1551
1552
1553
1554
1554
1555
1556
1556
1557
1558
1558
1559
1560
1561
1562
1563
1563
1564
1565
1565
1566
1567
1567
1568
1569
1569
1570
1571
1572
1573
1574
1574
1575
1576
1576
1577
1578
1578
1579
1580
1581
1582
1583
1584
1584
1585
1586
1586
1587
1588
1588
1589
1590
1591
1592
1593
1593
1594
1595
1595
1596
1597
1597
1598
1599
1599
1600
1601
1602
1603
1604
1604
1605
1606
1606
1607
1608
1608
1609
1610
1611
1612
1613
1613
1614
1615
1615
1616
1617
1617
1618
1619
1619
1620
1621
1622
1623
1624
1624
1625
1626
1626
1627
1628
1628
1629
1630
1631
1632
1633
1634
1634
1635
1636
1636
1637
1638
1638
1639
1640
1641
1642
1643
1643
1644
1645
1645
1646
1647
1647
1648
1649
1649
1650
1651
1652
1653
1654
1654
1655
1656
1656
1657
1658
1658
1659
1660
1661
1662
1663
1663
1664
1665
1665
1666
1667
1667
1668
1669
1669
1670
1671
1672
1673
1674
1674
1675
1676
1676
1677
1678
1678
1679
1680
1681
1682
1683
1683
1684
1685
1685
1686
1687
1687
1688
1689
1689
1690
1691
1692
1693
1694
1694
1695
1696
1696
1697
1698
1698
1699
1700
1701
1702
1703
1703
1704
1705
1705
1706
1707
1707
1708
1709
1709
1710
1711
1712
1713
1714
1714
1715
1716
1716
1717
1718
1718
1719
1720
1721
1722
1723
1723
1724
1725
1725
1726
1727
1727
1728
1729
1729
1730
1731
1732
1733
1734
1734
1735
1736
1736
1737
1738
1738
1739
1740
1741
1742
1743
1744
1744
1745
1746
1746
1747
1748
1748
1749
1750
1751
1752
1753
1754
1754
1755
1756
1756
1757
1758
1758
1759
1760
1761
1762
1763
1763
1764
1765
1765
1766
1767
1767
1768
1769
1769
1770
1771
1772
1773
1774
1774
1775
1776
1776
1777
1778
1778
1779
1780
1781
1782
1783
1783
1784
1785
1785
1786
1787
1787
1788
1789
1789
1790
1791
1792
1793
1794
1794
1795
1796
1796
1797
1798
1798
1799
1800
1801
1802
1803
1803
1804
1805
1805
1806
1807
1807
1808
1809
1809
1810
1811
1812
1813
1814
1814
1815
1816
1816
1817
1818
1818
1819
1820
1821
1822
1823
1823
1824
1825
1825
1826
1827
1827
1828
1829
1829
1830
1831
1832
1833
1834
1834
1835
1836
1836
1837
1838
1838
1839
1840
1841
1842
1843
1843
1844
1845
1845
1846
1847
1847
1848
1849
1849
1850
1851
1852
1853
1854
1854
1855
1856
1856
1857
1858
1858
1859
1860
1861
1862
1863
1863
1864
1865
1865
1866
1867
1867
1868
1869
1869
1870
1871
1872
1873
1874
1874
1875
1876
1876
1877
1878
1878
1879
1880
1881
1882
1883
1883
1884
1885
1885
1886
1887
1887
1888
1889
1889
1890
1891
1892
1893
1893
1894
1895
1895
1896
1897
1897
1898
1899
1899
1900
1901
1902
1903
1903
1904
1905
1905
1906
1907
1907
1908
1909
1909
1910
1911
1912
1913
1914
1914
1915
1916
1916
1917
1918
1918
1919
1920
1921
1922
1923
1923
1924
1925
1925
1926
1927
1927
1928
1929
1929
1930
1931
1932
1933
1934
1934
1935
1936
1936
1937
1938
1938
1939
1940
1941
1942
1943
1943
1944
1945
1945
1946
1947
1947
1948
1949
1949
1950
1951
1952
1953
1954
1954
1955
1956
1956
1957
1958
1958
1959
1960
1961
1962
1963
1963
1964
1965
1965
1966
1967
1967
1968
1969
1969
1970
1971
1972
1973
1974
1974
1975
1976
1976
1977
1978
1978
1979
1980
1981
1982
1983
1983
1984
1985
1985
1986
1987
1987
1988
1989
1989
1990
1991
1992
1993
1993
1994
1995
1995
1996
1997
1997
1998
1999
1999
2000
2001
2002
2003
2003
2004
2005
2005
2006
2007
2007
2008
2009
2009
2010
2011
2012
2013
2013
2014
2015
2015
2016
2017
2
```

```
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 solução implementada.

Os testes realizados demonstraram que todas as funcionalidades desenvolvidas funcionam conforme o esperado. O código foi estruturado de forma modular e respeitando boas práticas de desenvolvimento, como a utilização de controladores para interação com o backend através de *web requests*.

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 comprehensível e fundamentada.

A aplicação demonstrou alta qualidade, refletida no seguinte:

- **Estrutura Modular e Organizada:** A aplicação segue padrões como *MVC* e utiliza bibliotecas apropriadas para garantir uma arquitetura limpa e escalável.
- **Integração de Conceitos Avançados:** A solução implementa o *LINQ* e frameworks modernos para proporcionar uma implementação eficiente.
- **Controladores e Interações via web requests:** A aplicação faz uso de controladores que interagem com o backend através de *web requests*, promovendo uma arquitetura modular e de fácil manutenção.
- **Configuração de Pipelines CI/CD:** Implementação de pipelines para automação de testes e implantação contínua, utilizando ferramentas como *Github Actions* para garantir a qualidade e o fluxo contínuo de desenvolvimento.
- **Validação Rigorosa de Entradas:** A aplicação valida as entradas do utilizador de forma completa, garantindo a integridade dos dados e proporcionando um tratamento adequado de erros.
- **Integração entre Frontend e Backend:** Funcionalidades implementadas demonstram a integração eficiente e fluída entre o frontend e o backend, assegurando a comunicação e o processamento adequado das informações.

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 16 de dezembro de 2024.