# Universidade da Beira Interior
Departamento de Informática


## Nº Trabalho Prático 2: *Computação Desconectada*

Elaborado por:

**M13939 João Martins**
**M13943 Rui Simões**
**E11307 Fábio Dias**

Orientador:

**Professor Doutor João Manuel da Silva Fernandes Muranho**

22 de novembro de 2023

**UC** Unidade Curricular

**UBI** Universidade da Beira Interior

**UI** *User Interface*

**SSMS** *SQL Server Management Studio*

# Conteúdo

# Lista de Figuras

# Lista de Excertos de Código

# Capítulo 1

# Introdução

## 1.1 Enquadramento

Como componente da Unidade Curricular (UC) de Sistemas de Gestão de Bases de Dados do Mestrado em Engenharia Informática da Universidade da Beira Interior, o Professor Doutor João Muranho solicitou o desenvolvimento do conceito "Computação Desconectada" no âmbito do seu trabalho prático número dois.

## 1.2 Objetivos

O objetivo principal deste trabalho é estudar e desenvolver o tema proposto, tanto de uma perspetiva teórica quanto de uma perspetiva prática. Numa perspetiva teórica, foi feita uma revisão bibliográfica sobre o tema.

Enquanto que, numa perspetiva prática foi desenvolvida uma aplicação que permite simular e superar os problemas deste tema para desenvolver a parte prática.

## 1.3 Organização do Documento

Este documento foi estruturado da seguinte forma:

- O primeiro capítulo - **Introdução** - descreve o projeto, como ele se enquadra na unidade curricular, os seus objetivos e como o documento está organizado.

- O segundo capítulo - **Computação Desconectada** - introduz conceitos-chave e desafio por detrás deste tema, aborda um caso de estudo e possível abordagem aos problemas do tema.
- O terceiro capítulo - **Aplicação Desenvolvida** - descreve como foi desenvolvida cada funcionalidade da aplicação, como esta trabalha em modo desconectado e o critério que utiliza aquando existe conflito de dados
- O quarto capítulo - **Exploração da Aplicação** - apresenta o teste utilizado para gerar conflito na base de dados, como a aplicação lidou com isso e outras possíveis formas de resolução
- O quinto capítulo - **Conclusões** - apresenta uma reflexão critica sobre o trabalho, bem como o que podia ter sido implementado a mais e ideias para trabalhos futuros nesta temática.

# Capítulo 2

# Computação desconectada: Estado da Arte

## 2.1 Introdução

A computação desconectada, também conhecida como *Disconnected Model* ou *Mobile Computing* refere-se a uma abordagem em que sistemas e aplicações são projetadas para operar sem depender de uma conexão constante à rede. Isso implica a capacidade de realizar operações, manipular dados e executar processos mesmo quando o dispositivo ou sistema não está conectado à internet, ou a uma rede centralizada.

O principal desafio na computação desconectada reside na gestão de dados. Em ambientes *offline*, a sincronização e a reconciliação de dados tornam-se cruciais. Quando um dispositivo está desconectado, as alterações nos dados locais precisam ser geridos e posteriormente reconciliadas quando a conexão for reestabelecida. Isso envolve questões complexas, como garantir a consistência entre os dados armazenados localmente e aqueles em servidores remotos.

## 2.2 Caso de Estudo: *Mobile Computing*

Um aspeto chave destes sistemas de gestão de base de dados é a sua capacidade de lidar com a desconexão. A desconexão refere-se à condição em que um sistema móvel não consegue comunicar com alguns ou todos os seus pares. Nessa situação, o dispositivo móvel deixa de ter acesso aos dados partilhados.

Abordagens de replicação otimista têm sido propostas para lidar com o problema da desconexão. Nestas abordagens, é permitido ao dispositivo mó-

vel replicar localmente os dados partilhados e operar com esses dados enquanto está desconectado. As atualizações locais podem ser propagadas para o restante do sistema quando a reconexão ocorre. No entanto, uma vez que as atualizações locais podem entrar em conflito com outras atualizações no sistema, é necessária alguma forma de deteção e resolução de conflitos.[1]

## 2.3 Computação Desconectada: Possível Abordagem

Uma abordagem comum para lidar com a computação desconectada envolve o uso de tabelas temporárias ou *caches* locais. Os dados necessários são armazenados temporariamente no dispositivo em tabelas específicas, e as operações ocorrem localmente. Quando a conexão é reestabelecida, ocorre a sincronização e reconciliação de dados, garantindo a consistência entre o estado local e remoto.

A implementação bem-sucedida dessa abordagem requer o estabelecimento de políticas robustas de reconciliação de dados. Isso inclui a definição de regras claras sobre como lidar com conflitos de dados, garantindo que as versões mais recentes e precisas dos dados sejam preservadas e atualizadas em ambientes locais e remotos. O processo de sincronização e reconciliação é essencial para evitar inconsistências e manter a integridade dos dados em sistemas que operam em ambientes desconectados.

# Capítulo 3

# Apresentação da Aplicação Desenvolvida

## 3.1 Introdução

Neste capítulo, será apresentada a aplicação desenvolvida. Esta foi concebida para permitir aos utilizadores realizar operações de Inserir, Alterar, Apagar e Consultar encomendas, mesmo quando desconectados do servidor de bases de dados.

A capacidade de trabalhar em modo desconectado e realizar a reconciliação dos dados é fundamental para otimizar a carga do servidor e aumentar a sua a escalabilidade.

### Funcionalidades Principais da Aplicação

A aplicação desenvolvida abrange as seguintes funcionalidades-chave:

1. **Inserir, Alterar, Apagar e Consultar Encomendas** - oferece uma interface intuitiva que permite aos utilizadores realizar operações básicas de gestão de encomendas. Isso inclui a capacidade de adicionar novas encomendas, alterar informações existentes, apagar encomendas e consultar dados de encomendas previamente inseridos.

2. **Trabalho em Modo Desconectado** - permite aos utilizadores obter uma cópia local dos dados, armazenada numa tabela em memória. Isso facilita a realização de modificações nos dados mesmo quando desconectados do servidor, preservando a integridade das informações.

3. **Reconciliação dos Dados** - envolve a submissão de alterações locais para o servidor e a resolução de conflitos que possam surgir quando os dados locais e do servidor estão em desacordo.

### Objetivos da Aplicação Desenvolvida

A aplicação foi concebida com os seguintes objetivos principais:

1. **Redução da Carga do Servidor**: Permitir que os utilizadores realizem operações básicas mesmo quando desconectados, reduzindo assim a carga no servidor de bases de dados.

2. **Aumento da Escalabilidade**: Ao possibilitar a operação em modo desconectado, a aplicação visa aumentar a escalabilidade do sistema, permitindo que um maior número de utilizadores seja servido com os mesmos recursos.

3. **Flexibilidade na Gestão de Dados**: Proporcionar uma experiência de utilizador mais flexível, permitindo a gestão de dados mesmo em situações de conectividade intermitente.

Nos próximos capítulos, detalharemos o processo de desenvolvimento da aplicação, abrangendo desde a criação das tabelas e da base de dados até à implementação das funcionalidades de trabalho em modo desconectado e a reconciliação dos dados.

## 3.2 Inserir/Alterar/Apagar/Consultar Encomendas

Para realizar as operações que o utilizador deseja realizar, foi desenvolvido um *User Interface* (UI) que permite ao utilizador interagir com a base de dados. A consulta pode ser realizada através de uma tabela preenchida com todos os dados da tabela Encomenda da base de dados. Deste modo o utilizador pode realizar as operações que desejar e manter o acesso aos dados. As outras operações podem ser realizadas com os elementos de UI designados para cada operação. Existem 4 campos para o utilizador inserir os dados de uma nova entrada para a operação de inserir. Dois campos que pedem o id de uma encomenda e o novo valor do total que é pretendido para a operação de alterar. E por fim um campo que pede o id de uma encomenda para que esta possa ser removida.

## 3.3 Trabalhar em Modo Desconectado

Se o utilizador quiser trabalhar em modo desconectado, poderá clicar no botão *disconnect*, que irá criar uma nova janela semelhante á original, com a diferença de que agora o utilizador pode se voltar a ligar á base de dados, ou atualizar a base de dados com os seus novos dados e receber uma nova versão com as suas alterações bem como as realizadas por outros utilizadores. Enquanto o utilizador se encontra no modo Desconectado, os dados serão armazenados na *DataGridView* que está a mostrar os dados ao utilizador, qualquer alteração a esta tabela não afetará a base de dados. Quando o utilizador decidir atualizar a base de dados, todas as alterações detetadas na *datagridview* serão efetuadas na base de dados. Se uma linha inserida existe na base de dados, esta é atualizada com os valores que o utilizador inseriu, senão é inserida de forma normal. Para as alterações, é comparada a hora a que a última alteração desta linha foi realizada. Se o timestamp é diferente, então o utilizador é questionado se pretende substituir os dados na tabela com os seus. Apagar um valor é a operação mais simples, basta verificar se a encomenda existe, e realizar a operação se existir ou ignorar caso não exista.

## 3.4 Reconciliação dos Dados

Nesta secção, será abordada detalhadamente a lógica e implementação da reconciliação de dados, essencial para manter a consistência entre a aplicação e a base de dados, especialmente em cenários onde conflitos podem surgir.

A função responsável por essa tarefa é `SaveChangesToDatabase()`, exibida em 1, acionada através da interação do utilizador com a aplicação, especificamente pelo clique no botão *"Refresh and Update"*.

Esta opera num modelo desconectado, o que significa que as alterações feitas na interface do utilizador não são refletidas imediatamente na base de dados. Em vez disso, as alterações são armazenadas localmente e posteriormente aplicadas na base de dados quando a função é chamada. Isso permite que a aplicação continue a funcionar mesmo quando a conexão com o banco de dados é interrompida.

### Análise da Função SaveChangesToDatabase()

1. A função começa por estabelecer uma conexão com a base de dados. Posteriormente, percorre cada linha do *DataGridView*. Para cada linha, verifica se esta já existe na base de dados. Se existir, a função tenta efe-

tuar a sua atualização na base de dados. Caso não exista, a função procura inserir uma nova linha.

2. A reconciliação de dados torna-se relevante quando a função procura atualizar uma linha que já foi modificada por outra pessoa (no momento em que aplicação corre em modo desconectado), o que gera uma situação de **conflito** de dados. Nesta situação, a função dá a escolher ao utilizador entre as seguintes opções:

    - `Sobrescrever` *`(Overwrite)`*: Se o utilizador optar por sobrescrever, a linha do DataGridView será atualizada com os dados existentes na base de dados, refletindo as alterações feitas por outra entidade.

    - Manter Atual *(Keep Current)*: Se o utilizador optar por não sobrescrever, os dados da aplicação da linha do DataGridView serão mantidos, e a base de dados será atualizada com esses dados, preservando as alterações locais.

3. Esta abordagem garante a sincronização entre os dados na interface do utilizador e na base de dados, de acordo com a escolha do utilizador, sendo eficaz na gestão de eventuais conflitos resultantes de atualizações simultâneas.

4. Finalmente, a função encerra a conexão.

Ilustrado na figura 3.1 está representado o diagrama de fluxo para esta função.

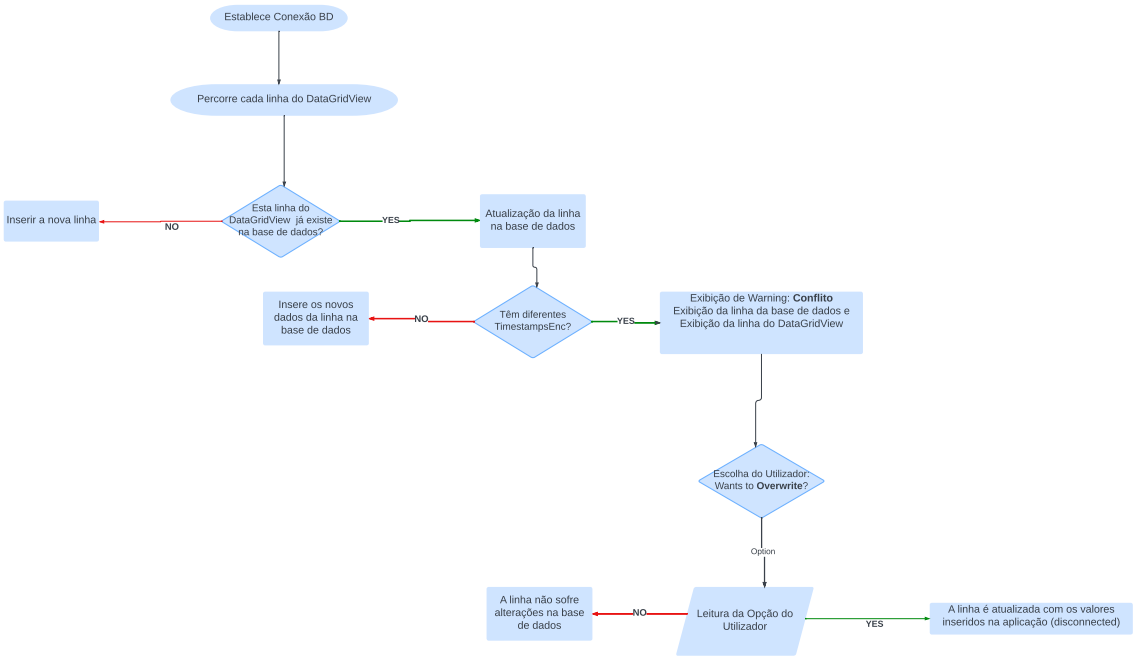


Figura 3.1: Diagrama de Fluxo da Função

## 3.5 Resolução de Conflitos

O cerne da função reside na resolução de conflitos, principalmente baseada no campo `TimestampEnc`, cuja representação está ilustrada na figura 4.2 . Este campo é utilizado como uma marca temporal para verificar se os dados locais são mais recentes que os dados no servidor. Caso ocorram conflitos, o utilizador é notificado, e a função apresenta opções para resolver o conflito.

### Representação do TimeStamp

Indo um pouco mais afundo, a representação `timestampEnc` é um tipo de dados de `timestamp`, que é uma sequência de bytes. No SQL Server, este tipo de dados é representado como uma sequência de 8 bytes que é única dentro

de uma base de dados, não relacionada com o tempo real (não armazena uma marca de tempo que indica um ponto específico no tempo real ou na data e hora). Em vez disso, é uma sequência de bytes que é única dentro de uma base de dados e é atualizada cada vez que a linha é modificada.

Portanto, apesar do nome, o tipo de dados 'timestamp' no SQL Server não é usado para registar o tempo real em que um evento ocorreu. Em vez disso, é usado principalmente para implementar a concorrência otimista, permitindo que se detete se uma linha foi modificada por outro processo desde a última vez que foi lida.

Os valores ilustrados na figura 4.2, como 0x00000000000008D6, 0x00000000000008D7, etc., são representações hexadecimais desses valores de `timestamp`. Cada valor hexadecimal é único e aumenta cada vez que uma modificação é feita.

Results | Messages

| | EncId | ClienteId | Data | Total | TimestampEnc |
|---|---|---|---|---|---|
| 1 | 2 | 2 | 2023-11-17 | 25.99 | 0x00000000000008D6 |
| 2 | 3 | 3 | 2023-11-17 | 90.99 | 0x00000000000008D7 |
| 3 | 5 | 5 | 2023-11-17 | 53.99 | 0x00000000000008D8 |
| 4 | 6 | 6 | 2023-11-17 | 62.99 | 0x00000000000008D9 |
| 5 | 7 | 7 | 2023-11-17 | 76.99 | 0x00000000000008DA |
| 6 | 8 | 4 | 2023-11-17 | 90.99 | 0x00000000000008DB |
| 7 | 9 | 5 | 2023-11-19 | 40.01 | 0x00000000000008D5 |

Figura 3.2: Representação TimestampEnc

O trecho de código ilustrado na figura 3.1 permite ler o valor do *timestampEnc* de determinada encomenda.

```
byte[] timestampEnc = (byte[])Form2.dataTable.Rows[rowIndex]["
    TimestampEnc"];
```

Excerto de Código 3.1: Armazenar valor do timestamp

Aqui, `Form2.dataTable.Rows[rowIndex]["TimestampEnc"]` está a aceder ao valor da coluna "TimestampEnc"na linha especificada (rowIndex) da tabela de dados. O (byte[]) antes disso está a converter esse valor em um array de bytes para que possa ser usado.

Este valor de timestamp pode ser usado para verificar se uma linha foi modificada desde a última vez que foi lida. Se o valor de timestamp na base de dados for diferente do valor de timestamp guardado localmente (aquando o processo de desconexão), significa que a linha foi modificada.

## 3.6 Conclusões

Nesta secção é então explicado como funciona a base da aplicação (com as operações primárias de: Inserir/Alterar/Eliminar/Consultar), como esta trabalha em modo desconectado, e por fim, o que faz para lidar e resolver possíveis conflitos nos dados aquando o processo de reconexão à base de dados.

O próximo capítulo detalha o teste para gerar conflito e mostra como a aplicação lidou com isso (resultado) e as lições aprendidas durante a exploração da aplicação.

# Capítulo 4

# Exploração da aplicação

## 4.1 Introdução

Este capitulo pretende apresentar o teste feito à aplicação e exibir os resultados (modo como operou) da mesma. Sendo que este teste se divide em duas partes, na geração de um conflito (deve ser detetado) e abordagem ao conflito (deve ser resolvido). Na secção de conclusão, reflete-se sobre outras estratégias de resolução de conflitos que não foram implementadas nesta aplicação.

## 4.2 Elaboração do Teste

Este teste é dividido em duas partes principais: a deteção de conflitos e o manejo de conflitos.

### Deteção de Conflitos

O objetivo da primeira parte do teste é avaliar como a aplicação **deteta** conflitos de dados. Para isso, um conflito de dados é intencionalmente gerado seguindo os passos abaixo:

1. Iniciar a aplicação e entrar em modo desconectado - é inicialmente lançada e colocada em modo desconectado. Este modo permite que as alterações sejam feitas na aplicação sem que sejam imediatamente refletidas na base de dados.

2. Alterar a quantidade da encomenda 6 no SSMS - enquanto a aplicação está em modo desconectado, a quantidade da encomenda 6 é alterada

diretamente na base de dados usando o SSMS, através do trecho de código ilustrado em 4.1. Esta alteração cria uma discrepância entre os dados na aplicação e os dados na base de dados.

3. Alterar a quantidade da encomenda 6 na aplicação - a aplicação é então usada para alterar a quantidade da mesma encomenda 6. Como a aplicação está em modo desconectado, ela não está ciente da alteração feita na base de dados.

4. Observar o aviso de conflito - quando a aplicação tenta sincronizar as suas alterações com a base de dados, ela deteta o conflito de dados.

```
UPDATE Encomenda
SET Total = 201.99
WHERE EncId = 6;
```

Excerto de Código 4.1: Função `SaveChangesToDatabase()`

| EncId | ClienteId | Data | Total | TimestampEnc |
|---|---|---|---|---|
| 2 | 2 | 2023-11-17 | 25.99 | 0x00000000000008D6 |
| 3 | 3 | 2023-11-17 | 90.99 | 0x00000000000008D7 |
| 5 | 5 | 2023-11-17 | 53.99 | 0x00000000000008D8 |
| 6 | 6 | 2023-11-17 | 25.99 | 0x00000000000008DC |

Figura 4.1: Antes do update no SSMS

| EncId | ClienteId | Data | Total | TimestampEnc |
|---|---|---|---|---|
| 2 | 2 | 2023-11-17 | 25.99 | 0x00000000000008D6 |
| 3 | 3 | 2023-11-17 | 90.99 | 0x00000000000008D7 |
| 5 | 5 | 2023-11-17 | 53.99 | 0x00000000000008D8 |
| 6 | 6 | 2023-11-17 | 201.99 | 0x00000000000008DD |

Figura 4.2: Depois do update no SSMS

## Reconciliação dos Dados

A segunda parte do teste foca-se em como a aplicação lida com o conflito detetado, ilustrado na figura 4.3. Quando um conflito é detetado, a aplicação oferece ao utilizador as duas opções já referenciadas em 2. Em que:

1. *Overwrite* - é opção "Sim"na caixa de diálogo

2. *Keep Current* - opção "Não"na caixa de diálogo

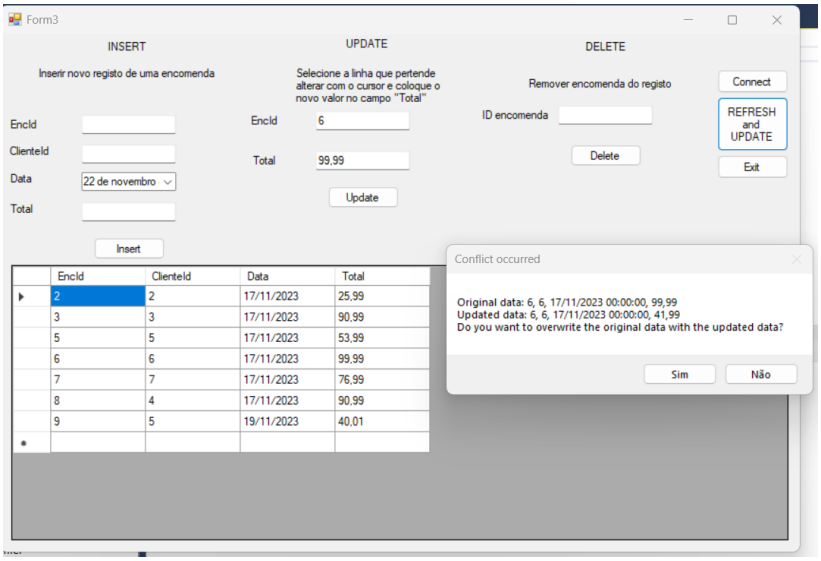


Figura 4.3: Warning de conflito e momento de decisão

Caso o utilizador prossiga com a opção:

- Sim - será mantido o registo da base de dados. Sendo assim, a encomenda com o ID seis ficará com um total de 41,99, como ilustrado na figura 4.5. Preservam-se as modificações locais.

- Não - será dado *'update'* ao total da encomenda com o respetivo ID, para um total de 99,99, como ilustrado na Figura 4.4. Atualização desta linha na base de dados. Preservam-se as modificações externas.

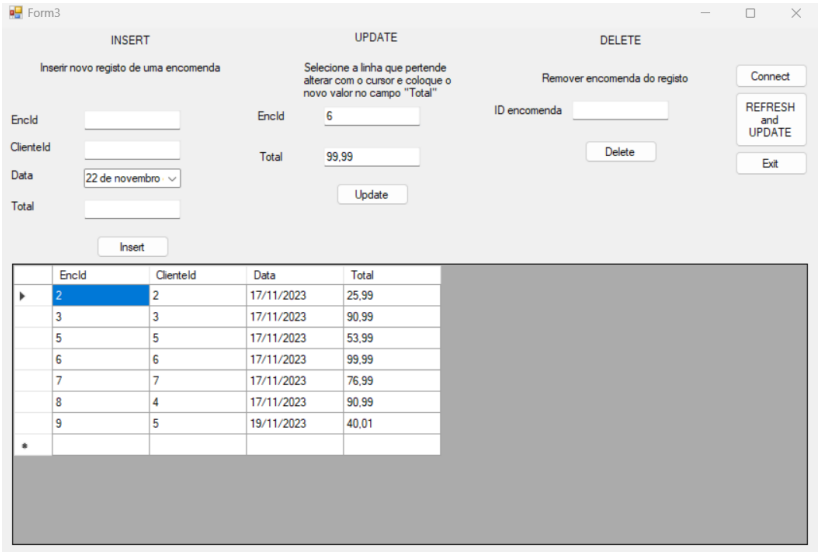


Figura 4.4: Escolha da opção Não

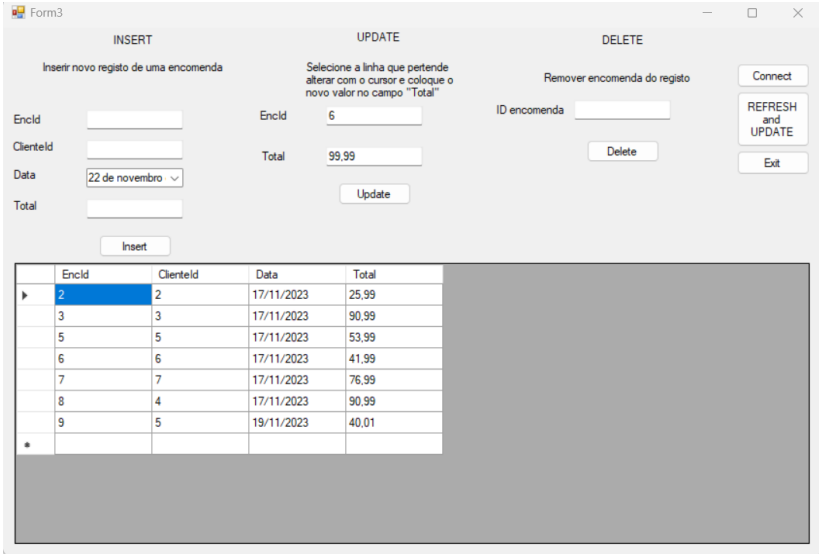


Figura 4.5: Escolha da opção Sim

## 4.3 Resultados e Lições

Como foi observado, a aplicação foi capaz de detetar conflito e resolver este com sucesso.

## 4.4 Conclusões

Concluímos que a nossa aplicação deteta conflitos e que seguindo o critério de resolução manual pelo utilizador, em que se fornece uma notificação ao utilizador, apresentando as alterações em é capaz de fazer a reconciliação dos dados.

É de notar que existem outros critérios:

- **Timestamps:** Utilização de timestamps para determinar a temporalidade das alterações, favorecendo as alterações mais recentes.
- **Prioridade do Utilizador:** Se existir um conflito, dar prioridade às alterações realizadas pelo utilizador, assumindo que as modificações locais têm maior relevância.
- **Modo de Acesso:** Avaliar se as alterações foram realizadas em modo desconectado ou conectado, priorizando aquelas efetuadas em modo desconectado para incentivar a flexibilidade de utilização.

# Capítulo 5

# Conclusão e Sugestões de Melhoria

## 5.1 Conclusão

Neste trabalho explicamos o conceito de "computação desconectada", falamos nas vantagens que este oferece, bem como os problemas que este levanta, entre estes problemas, o foco principal incidiu na questão da reconciliação de dados.

Foi desenvolvida uma aplicação para implementar um sistema de gestão de encomendas que pudesse tirar partido da computação desconectada para modificar os dados das encomendas quando não existe ligação à base de dados.

Podemos concluir que, na prática, a computação desconectada, consiste em guardar localmente, na máquina em que os dados são alterados, e a pedido do utilizador, a reconciliação entre os dados da base de dados e a aplicação é feita. Foi abordado também as implicações desta conciliação.

## 5.2 Sugestões de Melhoria

Embora o objetivo deste trabalho seja o estudo da computação desconectada, para fins da criação de uma aplicação que pudesse representar uma aplicação real deste conceito, este objetivo foi cumprido, no entanto, os métodos pelos quais o mesmo foi alcançado poderiam ser alvos de melhoria, como por exemplo, os dados não conciliados, na aplicação foram guardados em memória, e com este trabalho podemos concluir que esta não é uma boa abordagem, pois é limitada pela memória do sistema (memória RAM) disponível à aplicação.

Idealmente deveríamos utilizar a memória ROM do dispositivo, pois esta tende em ser mais generosa. Outra sugestão de melhoria, seria criar algum

tipo de procedimento que verificasse quais as linhas alteradas, e só modificar estas, pois em grandes datasets a abordagem atual de utilizar um loop para inserir todos os registos locais na base de dados, mesmo os que não foram alterados, seria muito pouco eficaz de um ponto de vista de performance.

# Bibliografia

[1] Shirish Hemant Phatak and B. R. Badrinath. Conflict resolution and reconciliation in disconnected databases, 2002. [Online] `https://ieeexplore.ieee.org/abstract/document/795148`. Último acesso a 15 de Novembro de 2023.

# Capítulo 6

# Apêndice

## .1 Função `SaveChangesToDatabase()`

```
private void SaveChangesToDatabase()
{
    try
    {
        connection.Open();

        foreach (DataGridViewRow row in dataGridView1.Rows)
        {
            if (!row.IsNewRow)
            {
                string tableName = "Encomenda"; // Change this to other
                    table names as needed

                string idColumnName = "EncId"; // Change this to other
                    primary key column names as needed

                // Get the TimestampEnc value from the original
                    DataTable in Form2
                int rowIndex = row.Index; // Get the index of the
                    current row
                byte[] timestampEnc = (byte[])Form2.dataTable.Rows[
                    rowIndex]["TimestampEnc"];

                // Check if the row already exists in the database
                string checkQuery = $"SELECT COUNT(*) FROM {tableName}
                    WHERE {idColumnName} = @{idColumnName}";
                SqlCommand checkCommand = new SqlCommand(checkQuery,
                    connection);
```

```
checkCommand.Parameters.AddWithValue($"@{idColumnName}"
    , row.Cells[idColumnName].Value);
checkCommand.Parameters.AddWithValue("@TimestampEnc",
    timestampEnc);


int count = (int)checkCommand.ExecuteScalar();

if (count > 0)
{

    // Row already exists, perform an UPDATE
    string updateQuery = $"UPDATE {tableName} SET ";

    foreach (DataGridViewCell cell in row.Cells)
    {
        string columnName = cell.OwningColumn.Name;
        if (columnName != idColumnName) // Skip the
            primary key column in the update query
        {
            updateQuery += $"{columnName} = @{
                columnName}, ";
        }
    }

    // Remove the trailing comma and space
    updateQuery = updateQuery.TrimEnd(',', ' ');

    updateQuery += $" WHERE {idColumnName} = @{
        idColumnName} AND TimestampEnc = @TimestampEnc"
        ;

    SqlCommand updateCommand = new SqlCommand(
        updateQuery, connection);

    foreach (DataGridViewCell cell in row.Cells)
    {
        string columnName = cell.OwningColumn.Name;
        if (columnName != idColumnName)
        {
            updateCommand.Parameters.AddWithValue($"@{
                columnName}", cell.Value);
        }
    }

    updateCommand.Parameters.AddWithValue($"@{
        idColumnName}", row.Cells[idColumnName].Value);
    updateCommand.Parameters.AddWithValue("
        @TimestampEnc", timestampEnc);
```

```
int rowsUpdated = updateCommand.ExecuteNonQuery();
if (rowsUpdated == 0)
{
    Debug.Write("Conflito");
    // Conflict occurred
    MessageBox.Show("Conflict occurred. The row was
        updated by someone else.");

    // Fetch the updated data
    string selectQuery = $"SELECT * FROM {tableName
        } WHERE {idColumnName} = @{idColumnName}";
    SqlCommand selectCommand = new SqlCommand(
        selectQuery, connection);
    selectCommand.Parameters.AddWithValue($"@{
        idColumnName}", row.Cells[idColumnName].
        Value);
    SqlDataReader reader = selectCommand.
        ExecuteReader();

    if (reader.Read())
    {
        int timestampIndex = reader.GetOrdinal("
            timestampEnc");
        List<object> updatedData = new List<object
            >();
        for (int i = 0; i < reader.FieldCount; i++)
        {
            if (i != timestampIndex)
            {
                updatedData.Add(reader.GetValue(i))
                    ;
            }
        }

        // Show both versions of the data to the
            user
        string originalDataString = string.Join(",
            ", row.Cells.Cast<DataGridViewCell>().
            Select(cell => cell.Value));
        string updatedDataString = string.Join(", "
            , updatedData);
        DialogResult result = MessageBox.Show($"
            Original data: {originalDataString}\
            nUpdated data: {updatedDataString}\nDo
            you want to overwrite the original data
             with the updated data?", "Conflict
            occurred", MessageBoxButtons.YesNo);
```

```
                    reader.Close();
                    if (result == DialogResult.Yes)
                    {
                        // Update the DataGridView row with the
                            updated data
                        for (int i = 0; i < updatedData.Count;
                            i++)
                        {
                            row.Cells[i].Value = updatedData[i
                                ];
                        }

                    }
                    else
                    {
                        // User chose not to overwrite, update
                            the database with the original data
                        string updateSql = "UPDATE Encomenda
                            SET Total = @OriginalValue WHERE
                            EncId = @EncId";

                        using (SqlCommand originalUpdateCommand
                            = new SqlCommand(updateSql,
                            connection))
                        {
                            // Get the original value and EncId
                                from the row
                            object originalValue = row.Cells["
                                Total"].Value;
                            object encId = row.Cells["EncId"].
                                Value;

                            originalUpdateCommand.Parameters.
                                AddWithValue("@OriginalValue",
                                originalValue);
                            originalUpdateCommand.Parameters.
                                AddWithValue("@EncId", encId);

                            originalUpdateCommand.
                                ExecuteNonQuery();
                        }
                    }
                }

                reader.Close();
            }
            else
            {
```

```
                        // If no conflict occurred, execute the update
                            command
                        updateCommand.ExecuteNonQuery();
                    }
                }
                else
                {

                    // Row doesn't exist, perform an INSERT
                    string insertQuery = $"INSERT INTO {tableName} (";

                    foreach (DataGridViewColumn column in dataGridView1
                        .Columns)
                    {
                        insertQuery += $"{column.Name}, ";
                    }

                    // Remove the trailing comma and space
                    insertQuery = insertQuery.TrimEnd(',', ' ');

                    insertQuery += ") VALUES (";

                    foreach (DataGridViewCell cell in row.Cells)
                    {
                        insertQuery += $"@{cell.OwningColumn.Name}, ";
                    }

                    // Remove the trailing comma and space
                    insertQuery = insertQuery.TrimEnd(',', ' ');

                    insertQuery += ")";

                    SqlCommand insertCommand = new SqlCommand(
                        insertQuery, connection);

                    foreach (DataGridViewCell cell in row.Cells)
                    {
                        insertCommand.Parameters.AddWithValue($"@{cell.
                            OwningColumn.Name}", cell.Value);
                    }

                    insertCommand.ExecuteNonQuery();
                }
            }
        }

        connection.Close();
    }
    catch (Exception ex)
```

```
        {
            MessageBox.Show("Error: " + ex.Message);
            Debug.Write("Error: " + ex.Message);
        }
    }
```

Excerto de Código 1: Função `SaveChangesToDatabase()`