Funções da aplicação

Form – “allDataBaseAcess”

Variáveis e instâncias:

dbHelper db = new dbHelper();

- Instância a classe dbHelper através do objeto “db";

private string atualTableName;

- Cria uma string privada "atualTableName";

private string searchVar1;

- Cria uma string privada "searchVar1";

private string searchVar2;

- Cria uma string privada "searchVar2";

private string searchVar3;

- Cria uma string privada "searchVar3";

public string columnName;

- Cria uma string "columnName";

Funções:

```
private void allDataBaseAcess_Load(object sender, EventArgs e)
{
   putTableNames();
   chooseBd_comboBox.DropDownStyle = ComboBoxStyle.DropDownList;
   if (chooseBd_comboBox.SelectedItem == null)
        {
           chooseBd_comboBox.SelectedIndex = 0;
        }
}
```

- Quando o formulário é aberto, as funções definidas são executadas. Se não houver um item selecionado na combobox, o primeiro item será selecionado automaticamente;

```
private void putTableNames()
{
   string[] tableNames = db.getJsonTableNames();

   foreach (string tableName in tableNames)
   {
      chooseBd_comboBox.Items.Add(tableName);
   }
}
```

- Utilizando a função da classe dbHelper, procura-se, em formato de array, todos os nomes das tabelas que estão no arquivo JSON e adiciona esses nomes aos itens da comboBox;

```
private void chooseBd_SelectedIndexChanged(object sender, EventArgs e)
    {
      if (chooseBd_comboBox.SelectedItem != null)
        search_TextBox1.Text = "";
        search_TextBox2.Text = "";
        search_TextBox3.Text = "";
      {
        atualTableName = chooseBd_comboBox.SelectedItem.ToString();

        tabelaAtual_label.Text = "Tabela: " + atualTableName;

        updateDataGrid();

        string searchOption = chooseBd_comboBox.SelectedItem.ToString();

        if (searchOption == "dbo.LeituraTag_FX96000")
        {
          labelSearch1.Text = "ID da Tag:";
          labelSearch1.Location = new Point(18, labelSearch1.Location.Y);
          labelSearch2.Text = "EPC da Tag:";
          labelSearch2.Location = new Point(8, labelSearch2.Location.Y);
          labelSearch3.Text = "Antena:";
          labelSearch3.Location = new Point(32, labelSearch3.Location.Y);
        }
        else if (searchOption == "dbo.tableReadInfo")
        {
          labelSearch1.Text = "Serie ID:";
          labelSearch1.Location = new Point(28, labelSearch1.Location.Y);
          labelSearch2.Text = "ID da Tag:";
          labelSearch2.Location = new Point(18, labelSearch2.Location.Y);
          labelSearch3.Text = "Antena:";
          labelSearch3.Location = new Point(32, labelSearch3.Location.Y);
        }
```

```
        else if (searchOption == "dbo.tableModelInfo")
        {
            labelSearch1.Text = "Serie ID:";
            labelSearch1.Location = new Point(28, labelSearch1.Location.Y);
            labelSearch2.Text = "ID Orde Prod:";
            labelSearch2.Location = new Point(4, labelSearch2.Location.Y);
            labelSearch3.Text = "Artigo:";
            labelSearch3.Location = new Point(39, labelSearch3.Location.Y);
        }
    }
}
```

- Quando o item selecionado na combobox chooseBd_comboBox é alterado, se o item selecionado não for nulo, os campos de texto search_TextBox1, search_TextBox2 e search_TextBox3 são limpos. Em seguida, o nome da tabela atual (atualTableName) é atualizado com o item selecionado na combobox e a label tabelaAtual_label é atualizada com o novo nome da tabela. A função updateDataGrid() é chamada para atualizar o grid de dados. Dependendo da tabela selecionada, os textos e as posições das labels de pesquisa (labelSearch1, labelSearch2 e labelSearch3) são configurados conforme necessário;

```
private void updateDataGrid()
{
  DataTable dadosDaTabela = db.allTablesShowAllBd(atualTableName);

  dataGrid_DB.DataSource = dadosDaTabela;
}
```

- Atualiza os dados do dataGrid_DB com os valores da tabela dadosDaTabela, que são obtidos pela função allTablesShowAllBd da classe dbHelper, e depois exibe esses valores na dataGrid.

```
private void getVarsSearch()
{
   searchVar1 = search_TextBox1.Text;
   searchVar2 = search_TextBox2.Text;
   searchVar3 = search_TextBox3.Text;
}
```

- A função getVarsSearch obtém os valores inseridos nas caixas de texto search_TextBox1, search_TextBox2 e search_TextBox3, e atribui esses valores às variáveis searchVar1, searchVar2 e searchVar3, respetivamente

```
private void searchButton_Click(object sender, EventArgs e)
    {
       GetVarsSearch();

       string searchOption = chooseBd_comboBox.SelectedItem.ToString();

       DataTable tableData = null;

       if (chooseBd_comboBox.SelectedItem != null)
       {
          if (searchOption == "dbo.LeituraTag_FX96000")
          {
             tableData = SearchInDatabase(searchOption, "Tag_id", "EPC_id", "Antenna_id");
          }
          else if (searchOption == "dbo.tableReadInfo")
          {
             tableData = SearchInDatabase(searchOption, "serie_ID", "tagId", "idAntena");
          }
          else if (searchOption == "dbo.tableModelInfo")
          {
             tableData = SearchInDatabase(searchOption, "serie_ID", "idProdOrder", "article");
          }
       }
```

```
        else
        {
            MessageBox.Show("Selecione uma tabela.");
        }

        if (tableData != null && tableData.Rows.Count > 0)
        {
            dataGrid_DB.DataSource = tableData;
        }
        else
        {
            MessageBox.Show("Nenhum resultado encontrado.");
        }
    }
```

- A função searchButton_Click é chamada quando o botão de pesquisa é clicado. Primeiro, chama a função getVarsSearch para obter os valores das caixas de texto de pesquisa e atribuí-los às variáveis correspondentes. Depois, verifica qual a tabela selecionada na combobox chooseBd_comboBox e, dependendo da tabela selecionada, chama a função SearchInDatabase com os parâmetros adequados para obter os dados da pesquisa. Se nenhum item estiver selecionado na combobox, exibe uma mensagem solicitando a seleção de uma tabela. Se a pesquisa retornar resultados, os dados são exibidos no dataGrid_DB; caso contrário, exibe uma mensagem informando que nenhum resultado foi encontrado;

```
private DataTable SearchInDatabase(string tableName, string columnName1, string columnName2,
string columnName3)
    {
        DataTable result = null;

        if (!string.IsNullOrEmpty(searchVar1) && !string.IsNullOrEmpty(searchVar2) &&
!string.IsNullOrEmpty(searchVar3))
        {
            result = db.ModelSearch(tableName, columnName1, searchVar1, columnName2, searchVar2,
columnName3, searchVar3);
        }
        else if (!string.IsNullOrEmpty(searchVar1) && !string.IsNullOrEmpty(searchVar2))
```

```
        {
            result = db.ModelSearch(tableName, columnName1, searchVar1, columnName2, searchVar2);
        }
        else if (!string.IsNullOrEmpty(searchVar1) && !string.IsNullOrEmpty(searchVar3))
        {
            result = db.ModelSearch(tableName, columnName1, searchVar1, columnName3, searchVar3);
        }
        else if (!string.IsNullOrEmpty(searchVar2) && !string.IsNullOrEmpty(searchVar3))
        {
            result = db.ModelSearch(tableName, columnName2, searchVar2, columnName3, searchVar3);
        }
        else if (!string.IsNullOrEmpty(searchVar1))
        {
            result = db.ModelSearch(tableName, columnName1, searchVar1);
        }
        else if (!string.IsNullOrEmpty(searchVar2))
        {
            result = db.ModelSearch(tableName, columnName2, searchVar2);
        }
        else if (!string.IsNullOrEmpty(searchVar3))
        {
            result = db.ModelSearch(tableName, columnName3, searchVar3);
        }

        return result;
    }
```

- A função SearchInDatabase realiza uma pesquisa na base de dados com base nos valores das variáveis de pesquisa (searchVar1, searchVar2, searchVar3) e nos nomes das colunas fornecidos. Dependendo das combinações de variáveis de pesquisa que não estão vazias, chama a função ModelSearch da classe db com os parâmetros apropriados. Se apenas uma das variáveis de pesquisa estiver preenchida, a pesquisa é realizada com um único parâmetro. Se duas ou mais variáveis de pesquisa estiverem preenchidas, a pesquisa é realizada com os parâmetros correspondentes. A função retorna os resultados da pesquisa como um DataTable;

```
private void checkBox1_CheckedChanged(object sender, EventArgs e)
    {
      if (checkBox1.Checked)
      {
        UpdateDataGrid();
        checkBox1.Checked = false;
      }
    }
```

- A função checkBox1_CheckedChanged é chamada quando o estado do checkBox1 é alterado. Se a checkbox estiver marcada (Checked for true), a função UpdateDataGrid é chamada para atualizar os dados do dataGrid_DB com os valores da tabela dadosDaTabela obtidos através da função allTablesShowAllBd da classe db. Após a atualização, a checkbox é desmarcada (checkBox1.Checked = false);

```
private void UpdateDataGrid()
{
 DataTable dadosDaTabela = db.allTablesShowAllBd(atualTableName);
 dataGrid_DB.DataSource = dadosDaTabela;
}
```

- A função UpdateDataGrid obtém os dados da tabela atual (atualTableName) através da função allTablesShowAllBd da classe db, e define esses dados como a fonte de dados (DataSource) do dataGrid_DB, atualizando assim a visualização da grelha de dados;

Form – “initialPage”

Variáveis e instâncias:

dbHelper db = new dbHelper();

- Instância a classe dbHelper através do objeto “db";

private string iniPath = "config.ini";

- Indica o caminho do ficheiro .ini;

public static initialPage instance;

- Permite este form ser instanciado em outro form;

public string idProdOrdem;

- Cria uma string "idProdOrdem";

public string artigo;

- Cria uma string "artigo";

public int quantidade;

- Cria uma int "quantidade";

public int quantidadeMin;

- Cria uma string "quantidadeMin";

public string idSerie;

- Cria uma string "idSerie";

public bool canOpenRfid = true;

- Cria um booleano "canOpenRfid" e inicaliza-lo a true;

public bool canOpenDb = true;

- Cria um booleano "canOpenDb" e inicaliza-lo a true;

Funções:

```
private void submitModelBTN_Click(object sender, EventArgs e)
{
   bool isRfidFormOpen = Application.OpenForms["rfidReader"] as Form != null;

   getVars();

   if (!isRfidFormOpen && canGoBd())
   {
         showRfidForm();
   }
}
```

- Na função submitModelBTN_Click, primeiro verifica-se se o formulário chamado "rfidReader" está aberto, atribuindo um valor booleano à variável isRfidFormOpen. Em seguida, chama-se a função getVars() para obter os valores dos campos de entrada. Posteriormente, verifica-se se o formulário "rfidReader" não está aberto e se é possível prosseguir com o BD (banco de dados), usando a função canGoBd(). Se essas condições forem cumpridas, o formulário "rfidReader" é exibido através da função showRfidForm();

```
private void getVars()
{
   idProdOrdem = idOrdProdText.Text;
   artigo = artigoText.Text;
   quantidade = Convert.ToInt32(quantidadeText.Value);
   quantidadeMin = Convert.ToInt32(quantMinDiaText.Value);
   idSerie = idSerieText.Text;
}
```

- Na função getVars(), os valores dos campos de entrada (idOrdProdText, artigoText, quantidadeText, quantMinDiaText e idSerieText) são obtidos e atribuídos às variáveis correspondentes (idProdOrdem, artigo, quantidade, quantidadeMin e idSerie, respetivamente);

```
private void varsClear()
{
   idProdOrdem = "";
   artigo = "";
   quantidade = 0;
   quantidadeMin = 0;
   idSerie = "";

   idOrdProdText.Text = "";
   artigoText.Text = "";
   quantidadeText.Value = 0;
   quantMinDiaText.Value = 0;
   idSerieText.Text = "";
}
```

- Na função varsClear(), os valores das variáveis (idProdOrdem, artigo, quantidade, quantidadeMin e idSerie) são limpos, ou seja, são atribuídos valores vazios ou zero. Além disso, os campos de entrada correspondentes (idOrdProdText, artigoText, quantidadeText, quantMinDiaText e idSerieText) também são limpos, definindo seus valores como vazios ou zero, conforme necessário;

```
private bool canGoBd()
    {
      if (string.IsNullOrEmpty(idProdOrdem) || string.IsNullOrEmpty(artigo) || quantidade == 0)
      {
        MessageBox.Show("Por favor, preencha todos os campos obrigatórios.");
        return false;
      }
      else
      {
        db.modelAddToBd(idProdOrdem, idSerie, artigo, quantidade, quantidadeMin);
        return true;
      }
    }
```

- Esta função verifica se os campos obrigatórios (idProdOrdem, artigo e quantidade) estão preenchidos. Se algum deles estiver vazio ou se a quantidade for igual a zero, exibe uma mensagem de aviso e retorna falso, indicando que não é possível prosseguir com a operação no banco de dados. Caso contrário, se todos os campos obrigatórios estiverem preenchidos, chama a função modelAddToBd da classe db para adicionar o modelo à base de dados e retorna verdadeiro, indicando que é possível prosseguir com a operação na base de dados;

```
private void showAviso()
    {
      avisoForm aviso = new avisoForm();
      aviso.Show();
    }
```

- Esta função cria uma instância de um formulário chamado avisoForm e abre o rewa;

```
private void clearVars_Click(object sender, EventArgs e)
{
  varsClear();
}
```

- Quando o botão é clicado, esta função chama a função varsClear(), que é responsável por limpar as variáveis do formulário;

```
private void showDbButton_Click(object sender, EventArgs e)
{
   bool isBdFormOpen = Application.OpenForms["allDataBaseAcess"] as Form != null;

   if (!isBdFormOpen)
   {
      allDataBaseAcess db = new allDataBaseAcess();
           db.Show();
   }
}
```

- Quando o botão é clicado, esta função verifica se o formulário "allDataBaseAcess" já está aberto. Se não estiver aberto, cria uma nova instância desse formulário e mostra-o na tela;

```
private void exitToolStripMenuItem_Click(object sender, EventArgs e)
{
 rfidReader.ActiveForm.Close();
}
```

- Realiza a ação de fechar os forms quando o botão de sair no menu é carregado;

```
private void aboutToolStripMenuItem_Click(object sender, EventArgs e)
{
 ConfigManager configManager = new ConfigManager();
 configManager.PrintConfig(iniPath);
}
```

- Quando o botão de "sobre" é pressionado vai mostrar as informações do rfid que estão no ficheiro .ini;

Form – “rfidReader”

Variáveis e instâncias:

dbHelper db = new dbHelper();

- Instância a classe dbHelper através do objeto “db";

private string jsonPath = @".\config_DB_JSON.json";

- Cria uma string "jsonPath" que vai indicar o caminho para o ficheiro de configuração de ligação à base de dados JSON;

private string iniPath = "config.ini";

- Cria uma string "iniPath" que vai indicar o caminho para o ficheiro de configuração de ligação ao RFID .ini;

private bool _IsConnecting = false;

- Cria um boolean "_IsConnecting" e inicializa-lo a false;

private bool _ISConnected = false;

- Cria um boolean "_ISConnected" e inicializa-lo a false;

public RFIDReader _RFIDReaderAPI;

- Refere a API do RFID com o nome "_RFIDReaderAPI"

private bool _IsReading = false;

- Cria um boolean "_IsReading" e inicializa-lo a false;

private int readCount = 0;

- Cria um int "readCount" com o valor 0;

private int tagCount = 0;

- Cria um int "tagCount" com o valor 0;

private STATUS_EVENT_TYPE _RFIDEventStatus;

- Refere ao evento do status do RFID com o nome "_RFIDEventStatus"

private delegate void _UpdateStatus(Events.StatusEventData e);

- Cria um delegate "_UpdateStatus";

private delegate void _UpdateReader(Events.ReadEventData e);

- Cria um delegate "_UpdateReader";

private _UpdateReader _UpdateReaderHandler;

- Refere ao evento do status do RFID com o nome "_UpdateReaderHandler";

private Hashtable _TagTable;

- Refere a uma HASTABLE com o nome "_TagTable";

private TriggerInfo _TriggerInfo;

- Refere ao evento do status do RFID com o nome "_TriggerInfo";

private TagStorageSettings _TagStorageSettings;

- Refere à TagStorageSettings do RFID com o nome "_TagStorageSettings";

private bool rfidIsReading = true;

- Cria um boolean "rfidIsReading" e inicializa-lo a true;

private bool firstClick = false;

- Cria um boolean "firstClick" e inicializa-lo a false;

private string postoAntena;

- Cria uma string "postoAntena";

public bool isOpen;

- Cria um boolean "isOpen";

private bool canInsertRead = false;

- Cria um boolean "canInsertRead" e inicializa-lo a false;

public bool IsReading { get; private set; }

- Cria um boolean "IsReading" com os respetivos getters e setters;

public int RSSIValueSet { get; set; }

- Cria um boolean "RSSIValueSet" com os respetivos getters e setters;

public bool RSSIEnable { get; set; }

- Cria um boolean "RSSIEnable" com os respetivos getters e setters;

Classes:

```
internal class TagModel
{
 public string EPCCode { get; set; }
 public string AntennaID { get; set; }
 public string TIDCode { get; set; }
 public sbyte RSSIValue { get; set; }
 public DateTime FirstSeenTimeStamp { get; set; }
}
```

- Esta classe é chamada TagModel e é utilizada para representar um modelo de tag RFID. Ela possui cinco atributos:

EPCCode: Uma string que representa o código EPC da tag RFID.

AntennaID: Uma string que representa o identificador da antena onde a tag RFID foi detectada.

TIDCode: Uma string que representa o código TID (Tag ID) da tag RFID.

RSSIValue: Um valor sbyte que representa a potência do sinal recebido (RSSI) da tag RFID.

FirstSeenTimeStamp: Um objeto DateTime que representa o horário em que a tag RFID foi inicialmente detectada.

Estes atributos permitem armazenar informações importantes sobre as tags RFID, como identificação, localização e potência do sinal, facilitando o processamento e análise dos dados coletados pelo sistema RFID;

```
public class ConfigSettings
    {
        public string IP { get; set; }
        public int Port { get; set; }
        public string DataSource { get; set; }
        public bool RSSIEnable { get; set; }
        public int RSSIValueSet { get; set; }
    }
```

- Esta classe ConfigSettings é usada para armazenar as configurações do sistema. Ela contém os seguintes atributos:

IP: Uma string que representa o endereço IP utilizado pelo sistema.

Port: Um número inteiro que representa a porta utilizada para comunicação.

DataSource: Uma string que indica a fonte de dados utilizada pelo sistema.

RSSIEnable: Um booleano que indica se a funcionalidade RSSI está habilitada ou não.

RSSIValueSet: Um número inteiro que representa o valor configurado para o RSSI;

```
public class ConfigManager
    {
        public ConfigSettings LoadConfig(string filePath)
        {
            var parser = new FileIniDataParser();
            IniData data = parser.ReadFile(filePath);

            return new ConfigSettings
            {
                IP = data["Connection"]["IP"],
                Port = int.Parse(data["Connection"]["Port"]),
                DataSource = data["Database"]["DataSource"],
                RSSIEnable = bool.Parse(data["ReadSettings"]["RSSIEnable"]),
                RSSIValueSet = int.Parse(data["ReadSettings"]["RSSIValueSet"])
            };
        }
        public void PrintConfig(string filePath)
        {
            var parser = new FileIniDataParser();
            IniData data = parser.ReadFile(filePath);

            string configText = $"[Connection]\nIP: {data["Connection"]["IP"]}\nPort:
{data["Connection"]["Port"]}\n\n" +
                                $"[ReadSettings]\nRSSIEnable:
{data["ReadSettings"]["RSSIEnable"]}\nRSSIValueSet: {data["ReadSettings"]["RSSIValueSet"]}";

            MessageBox.Show(configText, "Configurações do Sistema");
        }
    }
```

- Esta classe ConfigManager é responsável por carregar e imprimir as configurações do sistema a partir de um arquivo INI. Ela possui os seguintes métodos:

LoadConfig(string filePath): Este método carrega as configurações do sistema a partir de um arquivo INI especificado pelo caminho filePath. Utiliza a biblioteca FileIniDataParser para analisar o arquivo INI e extrair as configurações relevantes. Em seguida, instância um objeto ConfigSettings e atribui os valores lidos do arquivo INI aos atributos correspondentes desse objeto. Finalmente, retorna o objeto ConfigSettings preenchido com as configurações.

PrintConfig(string filePath): Este método imprime as configurações do sistema em uma mensagem de caixa de diálogo. Assim como o método LoadConfig, ele analisa o arquivo INI usando a biblioteca FileIniDataParser. Em seguida, constrói uma string formatada que contém as configurações importantes, como IP, porta, habilitação do RSSI e valor do RSSI. Por fim, exibe essa string em uma mensagem de caixa de diálogo para que o usuário possa visualizar as configurações do sistema;

Funções:

```
private void rfidReader_Load(object sender, EventArgs e)
    {
       LerConfiguracoes(jsonPath);
       Connect();
       getVarsModelo();
       timer1.Start();
       _UpdateReaderHandler = new _UpdateReader(ActionRead);
       _TagTable = new Hashtable();
       StartReadding();
       pictureBox1.BackColor = System.Drawing.Color.Green;
       optionsEnabled();
    }
```

- Lê as configurações do arquivo JSON, conecta-se ao dispositivo RFID, obtém as variáveis do modelo, inicia um temporizador, inicializa os objetos necessários para a leitura do RFID, define a cor de fundo da pictureBox1 como verde, habilita as opções;

```
private void rfidReader_FormClosing(object sender, FormClosingEventArgs e)
    {
        Disconnect();
    }
```

- Quando o formulário rfidReader está prestes a ser fechado, esta função é acionada para desconectar o dispositivo RFID;

```
private void timer1_Tick(object sender, EventArgs e)
    {
        hours_label.Text = DateTime.Now.ToString("HH:mm:ss");
        date_label.Text = DateTime.Now.ToString("dd/MM/yyyy");
    }
```

- A cada intervalo de tempo definido pelo temporizador (timer1), esta função atualiza os labels hours_label e date_label com a hora atual e a data atual;

```
public Rootobject LerConfiguracoes(string filePath)
    {
        string fileName = File.ReadAllText(filePath);
        return JsonConvert.DeserializeObject<Rootobject>(fileName);
    }
```

- Esta função lê o conteúdo do arquivo JSON especificado por filePath e, em seguida, o deserializa em um objeto da classe Rootobject usando o método JsonConvert.DeserializeObject. Finalmente, retorna esse objeto deserializado;

```
public bool Connect()
    {
      if (!_IsConnecting)
      {
        try
        {
          _IsConnecting = true;
          read_button.Enabled = true;
          insertDB_button.Enabled = true;
          ConfigManager configManager = new ConfigManager();
          ConfigSettings configSettings = configManager.LoadConfig(iniPath);
          _RFIDReaderAPI = new RFIDReader(configSettings.IP, (uint)configSettings.Port, 0);

          if (_RFIDReaderAPI != null)
          {
            _RFIDReaderAPI.Connect();
            if (!_RFIDReaderAPI.IsConnected)
            {
              IsReading = false;
              return false;
            }
            _RFIDReaderAPI.Events.ReadNotify += new
Events.ReadNotifyHandler(Events_ReadNotify);
            _RFIDReaderAPI.Events.AttachTagDataWithReadEvent = false;
            _RFIDReaderAPI.Events.StatusNotify += new
Events.StatusNotifyHandler(Events_StatusNotify);
            _RFIDReaderAPI.Events.ReadNotify += Events_ReadNotify;
            _RFIDReaderAPI.Events.AttachTagDataWithReadEvent = false;
            _RFIDReaderAPI.Events.NotifyReaderDisconnectEvent = true;
            _RFIDReaderAPI.Events.NotifyGPIEvent = true;
            _RFIDReaderAPI.Events.NotifyBufferFullEvent = true;
            _RFIDReaderAPI.Events.NotifyBufferFullWarningEvent = true;
            _RFIDReaderAPI.Events.NotifyReaderDisconnectEvent = true;
            _RFIDReaderAPI.Events.NotifyReaderExceptionEvent = true;
```

```
                _RFIDReaderAPI.Events.NotifyAccessStartEvent = true;
                _RFIDReaderAPI.Events.NotifyAccessStopEvent = true;
                _RFIDReaderAPI.Events.NotifyInventoryStartEvent = true;
                _RFIDReaderAPI.Events.NotifyInventoryStopEvent = true;
                _RFIDReaderAPI.Events.NotifyAntennaEvent = true;

                _ISConnected = true;

                optionsEnabled();

                return _RFIDReaderAPI.IsConnected;

            }
            return false;
        }
        catch (Exception ex)
        {
            MessageBox.Show("CONNECTION ERROR: " + ex.Message);
            read_button.Enabled = false;
            insertDB_button.Enabled = false;
            return false;
        }
        finally
        {
            _IsConnecting = false;
        }
    }
    return false;
}
```

- Esta função tenta estabelecer uma conexão com o dispositivo RFID usando as configurações fornecidas. Se a conexão for bem-sucedida, habilita os botões de leitura e inserção no banco de dados e configura os eventos e notificações do RFIDReader. Se ocorrer um erro durante o processo de conexão, uma mensagem de erro será exibida e os botões serão desabilitados;

```
public bool Disconnect()
    {
        if (_RFIDReaderAPI == null) return true;
        try
        {
            if (_RFIDReaderAPI.Actions.TagAccess.OperationSequence.Length > 0)
            {
                _RFIDReaderAPI.Actions.TagAccess.OperationSequence.StopSequence();
            }
            else
            {
                _RFIDReaderAPI.Actions.Inventory.Stop();
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine("DISCONNECTION ERRORS: " + ex.Message);
        }
        try
        {
            _RFIDReaderAPI.Disconnect();
            bool temp = _RFIDReaderAPI.IsConnected;
            _RFIDReaderAPI = null;
            read_button.Enabled = false;
            read_button.Text = "Start Reading";
            //clearDB_button.Enabled = true;
            insertDB_button.Enabled = false;
            return true;
        }
        catch (Exception ex)
        {
            Console.WriteLine("DISCONNECTION ERRORS: " + ex.Message);
            return false;
```

```
        }
        finally
        {
            _ISConnected = false;
            optionsEnabled();
        }
    }
```

- Esta função primeiro verifica se o objeto _RFIDReaderAPI não é nulo. Em seguida, para o inventário ou a sequência de operações, se houver alguma em andamento. Depois, desconecta do dispositivo RFID e atualiza as configurações do botão de leitura. Se ocorrer algum erro durante a desconexão, a função registra o erro e retorna false. Por fim, define que não está mais conectado e habilita as opções;

```
private void Events_ReadNotify(object sender, Events.ReadEventArgs e)
    {
        _UpdateReaderHandler.Invoke(e.ReadEventData);
    }
```

- Quando uma leitura de tag ocorre, este manipulador de evento invoca o método _UpdateReaderHandler e passa os dados de leitura como argumento. Isso permite que os dados de leitura sejam processados ou exibidos de acordo com a lógica de negócios do aplicativo;

```
public void Events_StatusNotify(object sender, Events.StatusEventArgs statusEventArgs)
    {
        try
        {
            _RFIDEventStatus = statusEventArgs.StatusEventData.StatusEventType;
            switch (_RFIDEventStatus)
            {
                case STATUS_EVENT_TYPE.INVENTORY_START_EVENT:
                    UpdateRichTextBox("Inventory Start Event");
                    break;
                case STATUS_EVENT_TYPE.INVENTORY_STOP_EVENT:
                    UpdateRichTextBox("Inventory Stop Event");
```

```
                break;
            case STATUS_EVENT_TYPE.ACCESS_START_EVENT:
                UpdateRichTextBox("Access Start Event");
                break;
            case STATUS_EVENT_TYPE.ACCESS_STOP_EVENT:
                UpdateRichTextBox("Access Stop Event");
                break;
            case STATUS_EVENT_TYPE.BUFFER_FULL_WARNING_EVENT:
                UpdateRichTextBox("Buffer Full Warning Event");
                break;
            case STATUS_EVENT_TYPE.BUFFER_FULL_EVENT:
                UpdateRichTextBox("Buffer Full Event");
                break;
            case STATUS_EVENT_TYPE.DISCONNECTION_EVENT:
                UpdateRichTextBox("Disconnection Event");
                break;
            case STATUS_EVENT_TYPE.READER_EXCEPTION_EVENT:
                UpdateRichTextBox("Reader Exception Event");
                break;
            default:
                break;
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine("STATUS NOTIFY ERRORS: " + ex.Message);
    }
}
```

- Este manipulador de evento recebe notificações de status do dispositivo RFID e, com base no tipo de evento recebido, atualiza o conteúdo de uma caixa de texto (presumivelmente uma RichTextBox) com uma mensagem correspondente. Essas mensagens geralmente informam o usuário sobre eventos significativos, como início ou término de inventário, eventos de conexão ou desconexão, etc… ;

```
private void UpdateRichTextBox(string message)
    {
      if (textBox1.InvokeRequired)
      {
        textBox1.BeginInvoke(new Action(() => { textBox1.Text = message; }));
      }
      else
      {
        textBox1.Text = message;
      }
    }
```

- Esta função verifica se a chamada é feita a partir de uma thread diferente da thread que criou o controle textBox1. Se for o caso, a atualização do texto da caixa de texto é invocada na thread que a criou usando BeginInvoke. Se não, a atualização do texto é feita diretamente;

```
public bool StartReadding()
    {
      _IsReading = false;
      try
      {
        if (_RFIDReaderAPI != null && _RFIDReaderAPI.IsConnected)
        {
          //Setting Memorybank
          if (_RFIDReaderAPI.Actions.TagAccess.OperationSequence.Length <= 0)
          {
            _RFIDReaderAPI.Actions.TagAccess.OperationSequence.DeleteAll();
            var operation = new TagAccess.Sequence.Operation
            {
              AccessOperationCode = ACCESS_OPERATION_CODE.ACCESS_OPERATION_READ
            };
            operation.ReadAccessParams.MemoryBank = MEMORY_BANK.MEMORY_BANK_TID;
            _RFIDReaderAPI.Actions.TagAccess.OperationSequence.Add(operation);
          }
          //
          //Start operation
          if (_RFIDReaderAPI.Actions.TagAccess.OperationSequence.Length > 0)
          {
            _RFIDReaderAPI.Actions.PurgeTags();
            _RFIDReaderAPI.Actions.TagAccess.OperationSequence.PerformSequence(new
AccessFilter(), _TriggerInfo, new AntennaInfo());
            IsReading = true;
            //_IsTrigger = false;//MinhChau.Nguyen: Reset trigger params
            Console.WriteLine("Start Read: " + IsReading);

            return true;
          }
        }
        return false;
```

```
        }
        catch (Exception ex)
        {
            Console.WriteLine("Start Readding Fail: " + ex.Message);
            return false;
        }
    }
```

- Esta função primeiro verifica se o objeto _RFIDReaderAPI existe e está conectado. Em seguida, configura a operação de leitura da memória bank TID, se ainda não estiver definida. Finalmente, inicia a operação de leitura no dispositivo RFID e define que está atualmente em processo de leitura;

```
private void ActionRead(Events.ReadEventData e)
    {
      //_IsDetectedTag = true;
      var tagData = _RFIDReaderAPI.Actions.GetReadTags(500);
      if (tagData != null)
      {
        foreach (var tag in tagData)
        {
          if (tag.OpCode == ACCESS_OPERATION_CODE.ACCESS_OPERATION_NONE ||
            (tag.OpCode == ACCESS_OPERATION_CODE.ACCESS_OPERATION_READ &&
             tag.OpStatus == ACCESS_OPERATION_STATUS.ACCESS_SUCCESS))
          {
            var tagID = tag.TagID;
            var keyTag = tagID;
            var isFound = false;

            Console.WriteLine("===> " + tag.MemoryBankData.ToString() + "+_TagTable: " +
_TagTable.Count);
            lock (_TagTable.SyncRoot) // lock sync access to hashtable
            {
              keyTag = tag.TagID + tag.MemoryBankData.ToString();
              isFound = _TagTable.ContainsKey(keyTag);
            }

            if (isFound)
            {
              var item = new TagModel
              {
                EPCCode = tagID,
                AntennaID = tag.AntennaID.ToString(),
                TIDCode = tag.MemoryBankData.ToString(),
                RSSIValue = tag.PeakRSSI,
                FirstSeenTimeStamp = DateTime.Now,
```

```
    };
    if (RSSIEnable && tag.PeakRSSI >= RSSIValueSet)
    {
        lock (_TagTable.SyncRoot)
        {
            _TagTable.Remove(keyTag);
            _TagTable.Add(keyTag, item);
        }
        AddItemToDataGridView(item);
    }
    else if (!RSSIEnable)
    {
        lock (_TagTable.SyncRoot)
        {
            _TagTable.Remove(keyTag);
            _TagTable.Add(keyTag, item);
        }
        AddItemToDataGridView(item);
    }
}
else
{
    var item = new TagModel
    {
        EPCCode = tagID,
        AntennaID = tag.AntennaID.ToString(),
        TIDCode = tag.MemoryBankData.ToString(),
        RSSIValue = tag.PeakRSSI,
        FirstSeenTimeStamp = DateTime.Now,
    };
    if (RSSIEnable && tag.PeakRSSI >= RSSIValueSet)
    {
        lock (_TagTable.SyncRoot)
```

```
                {
                    _TagTable.Add(keyTag, item);
                }
                AddItemToDataGridView(item);
            }
            else if (!RSSIEnable)
            {
                lock (_TagTable.SyncRoot)
                {
                    _TagTable.Add(keyTag, item);
                }
                AddItemToDataGridView(item);
            }
        }
    }
  }
}
```

- Esta função obtém os dados das tags lidos pelo dispositivo RFID. Em seguida, itera sobre as tags lidas e verifica se a operação foi bem-sucedida. Se for, cria um objeto TagModel com informações da tag e a adiciona à tabela, desde que atenda aos critérios de intensidade de sinal (RSSI) se essa verificação estiver ativada;

```
private void AddItemToDataGridView(TagModel item)
    {
        ConfigManager configManager = new ConfigManager();
        ConfigSettings configSettings = configManager.LoadConfig(iniPath);
        BeginInvoke(new MethodInvoker(() =>
        {
            if (clear_checkBox.Checked && item.RSSIValue < configSettings.RSSIValueSet)
            {
                // Encontra e remove a linha correspondente na tabela
                foreach (DataGridViewRow row in tagDataView.Rows)
                {
                    if (row.Cells[0].Value != null && row.Cells[0].Value.ToString() == item.EPCCode)
                    {
                        tagDataView.Rows.Remove(row);
                        return;
                    }
                }
            }
            else if (item.RSSIValue > configSettings.RSSIValueSet)
            {
                //se a tag já existe na tabela, apenas atualiza o RSSI a antennaID e o tempo de leitura data
e hora
                foreach (DataGridViewRow row in tagDataView.Rows)
                {
                    if (row.Cells[0].Value != null && row.Cells[0].Value.ToString() == item.EPCCode)
                    {
                        row.Cells[1].Value = item.AntennaID;
                        row.Cells[3].Value = item.RSSIValue;
                        row.Cells[4].Value = item.FirstSeenTimeStamp.ToString("yyyy-MM-dd");
                        row.Cells[5].Value = item.FirstSeenTimeStamp.ToString("HH:mm:ss");
                        readCount++;
                        updateTagCountLabel();
                        return;
```

```
            }
        }
        // Adiciona uma nova linha à DataGridView
        int rowIndex = tagDataView.Rows.Add();

        // Define os valores nas células correspondentes
        tagDataView.Rows[rowIndex].Cells["Column1"].Value = item.EPCCode;
        tagDataView.Rows[rowIndex].Cells["Column2"].Value = item.AntennaID;
        tagDataView.Rows[rowIndex].Cells["Column3"].Value = item.TIDCode;
        tagDataView.Rows[rowIndex].Cells["Column4"].Value = item.RSSIValue;
        tagDataView.Rows[rowIndex].Cells["Column5"].Value =
item.FirstSeenTimeStamp.ToString("yyyy-MM-dd");
        tagDataView.Rows[rowIndex].Cells["Column6"].Value =
item.FirstSeenTimeStamp.ToString("HH:mm:ss");

        //variaveis para enviar para a base de dados
        int antenaId = Convert.ToInt32(item.AntennaID);
        string epcCode = item.EPCCode;
        string tidCode = item.TIDCode;
        int rssiValue = item.RSSIValue;
        string dataLeitura = item.FirstSeenTimeStamp.ToString("dd/MM/yyyy");
        string horaLeitura = item.FirstSeenTimeStamp.ToString("HH:mm:ss");

        string idSerie = textIdSerie.Text;
        string idProdOrd = textIdOrdProd.Text;
        int quantity = int.Parse(textQuantidade.Text);

        if (idProdOrd != "" && db.rfidCanInsert(antenaId, tidCode))
        {
            db.rfidAddToBd(antenaId, tidCode, epcCode, rssiValue, dataLeitura, horaLeitura);
            postoAntena = postByAntenna(tidCode);
            //db.readAddToBd(idSerie, tidCode, postoAntena, idProdOrd, antenaId, quantity,
dataLeitura, horaLeitura);
        }
```

```
            else
            {
              // MessageBox.Show("Não tentou adicionar na BD");
            }

            tagCount++;

            if (db.readCanInsert(tidCode, antenaId))
            {
               db.readAddToBd(idSerie, tidCode, postoAntena, idProdOrd, antenaId, quantity,
dataLeitura, horaLeitura);
            }
            updateTagCountLabel();
            compareQuantWithTagRead();
          }
        }));
    }
```

- Esta função adiciona uma nova linha à DataGridView, exibindo informações sobre a tag lida. Ela verifica se a opção de limpeza está ativada e se o RSSI da tag é maior que o valor definido. Se sim, adiciona a tag à tabela e também a insere na base de dados, se possível. Se a tag já existe na tabela, apenas atualiza as informações;

```
public bool StopReadding()
    {
      _IsReading = true;
      try
      {
        if (_RFIDReaderAPI != null && _RFIDReaderAPI.IsConnected)
        {
          if (_RFIDReaderAPI.Actions.TagAccess.OperationSequence.Length > 0)
          {
            _RFIDReaderAPI.Actions.TagAccess.OperationSequence.StopSequence();

            Console.WriteLine("Stop Read: " + !IsReading);

            return true;
          }
        }
        return false;
      }
      catch (Exception ex)
      {

        Console.WriteLine("Stop Readding Fail: " + ex.Message);

        return false;
      }
    }
```

- Esta função para a leitura de tags RFID. Ela verifica se o leitor RFID está conectado e se há uma sequência de operação em andamento. Se sim, ela interrompe essa sequência de operação;

```
private void updateTagCountLabel()
    {
      readCountLabel.Invoke((MethodInvoker)delegate
      {
        //readCountLabel.Text = $"Nº of readings: {tagCount} ({readCount})";
        readCountLabel.Text = $"Nº de Leituras: {tagCount}";
      });
    }
```

- Esta função utiliza um método Invoke para garantir que a atualização do rótulo seja feita no thread da interface gráfica, evitando assim problemas de concorrência;

```
private void limparDB()
    {
        var dbconfig = LerConfiguracoes(jsonPath);
        try
        {
            using (SqlConnection conn = new SqlConnection($"Data Source={dbconfig.dataSource};Initial
Catalog={dbconfig.DBname};User Id={dbconfig.userID};Password={dbconfig.password}"))
            {
                conn.Open();
                using (SqlCommand cmd = new SqlCommand($"DELETE FROM
{dbconfig.tableNames.table1}", conn))
                {
                    cmd.ExecuteNonQuery();
                }
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
    }
```

- Esta função utiliza as configurações do banco de dados fornecidas no arquivo JSON para estabelecer uma conexão com a base de dados. Em seguida, executa um comando SQL para excluir todos os dados da tabela especificada;

```
private void read_button_Click(object sender, EventArgs e)
    {
      if (!firstClick)
      {
        firstClick = true;

        rfidIsReading = false;
        _UpdateReaderHandler = new _UpdateReader(ActionRead);
        _TagTable = new Hashtable();
        StartReadding();
        read_button.Text = "Parar leitura";
        //---color---//
        pictureBox1.BackColor = System.Drawing.Color.Green;
      }

      if (rfidIsReading)
      {
        rfidIsReading = false;
        _UpdateReaderHandler = new _UpdateReader(ActionRead);
        _TagTable = new Hashtable();
        StartReadding();
        read_button.Text = "Parar leitura";
        //---color---//
        pictureBox1.BackColor = System.Drawing.Color.Green;
      }
      else if (!rfidIsReading)
      {
        rfidIsReading = true;
        StopReadding();
        read_button.Text = "Retomar leitura";
        //---color---//
        pictureBox1.BackColor = System.Drawing.Color.Red;
      }
```

```
        else if (!_RFIDReaderAPI.IsConnected)
        {
            Console.WriteLine("Please connect to the reader first!");
        }
    }
```

- Esta função controla o comportamento do botão de leitura, iniciando, parando ou retomando a leitura das tags RFID, dependendo do estado atual da leitura e da conexão com o leitor RFID;

```
private void connectionToolStripMenuItem_Click(object sender, EventArgs e)
    {
        Connect();
    }
```

- Esta função simplesmente chama a função Connect() para estabelecer a conexão com o leitor RFID quando o item de menu "Conexão" é clicado;

```
private void exitToolStripMenuItem_Click(object sender, EventArgs e)
    {
        Disconnect();
        //limparDB();
        rfidReader.ActiveForm.Close();
    }
```

- Esta função primeiro chama a função Disconnect() para desconectar do leitor RFID e, em seguida, fecha o formulário ativo, que é o formulário do leitor RFID;

```
private void clearDB_button_Click(object sender, EventArgs e)
    {
       tagDataView.Rows.Clear();
       limparDB();
       //UpdateDataGridView();
       readCount = 0;
       tagCount = 0;
       updateTagCountLabel();

    }
```

- Esta função realiza várias ações quando o botão "Limpar BD" é clicado: limpa as linhas da DataGridView que exibe os dados da base de dados, limpa a própria base de dados, reinicia os contadores de leituras e de tags e atualiza a etiqueta que exibe o número de leituras;

```
private void disconnectToolStripMenuItem_Click(object sender, EventArgs e)
    {
       Disconnect();
    }
```

- Esta função simplesmente chama a função Disconnect() para desconectar do leitor RFID quando o item de menu "Desconectar" é clicado;

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
    {
       Disconnect();
       //limparDB();
       tagDataView.Rows.Clear();
    }
```

- Esta função primeiro chama a função Disconnect() para desconectar do leitor RFID e, em seguida, limpa as linhas da DataGridView que exibe os dados da base de dados;

```
private void insertDB_button_Click(object sender, EventArgs e)
    {
        bool isBdFormOpen = Application.OpenForms["editorDeTabela"] as Form != null;

        if (!isBdFormOpen)
        {
            editorDeTabela allDb = new editorDeTabela();
            allDb.Show();
        }
    }
```

- Esta função verifica se o formulário do editor de tabela está aberto. Se não estiver aberto, cria uma nova instância do formulário editorDeTabela e exibe-o;

```
private void aboutToolStripMenuItem_Click(object sender, EventArgs e)
    {
        ConfigManager configManager = new ConfigManager();
        configManager.PrintConfig(iniPath);
    }
```

- Esta função cria uma instância do ConfigManager e chama a função PrintConfig, que provavelmente imprime as configurações de algum arquivo específico;

```
private void clear_checkBox_CheckedChanged(object sender, EventArgs e)
    {
      if (clear_checkBox.Checked)
      {
        tagDataView.Rows.Clear();
        updateTagCountLabel();
      }
    }
```

- Esta função verifica se o CheckBox está marcado. Se estiver marcado, limpa as linhas da DataGridView que exibe os dados da base de dados e atualiza a etiqueta que mostra o número de leituras;

```
private void viewBd_btn_Click(object sender, EventArgs e)
    {
      bool isBdFormOpen = Application.OpenForms["editorDeTabela"] as Form != null;

      if (!isBdFormOpen)
      {
        allDataBaseAcess allDb = new allDataBaseAcess();
        allDb.Show();
      }
    }
```

- Esta função verifica se o formulário "editorDeTabela" já está aberto. Se não estiver aberto, cria uma nova instância do formulário "allDataBaseAcess" e o mostra;

```
private void getVarsModelo()
    {
        textIdOrdProd.Text = initialPage.instance.idProdOrdem;
        textArtigo.Text = initialPage.instance.artigo;
        textQuantidade.Text = Convert.ToString(initialPage.instance.quantidade);
        textQuantMin.Text = Convert.ToString(initialPage.instance.quantidadeMin);
        textIdSerie.Text = initialPage.instance.idSerie;
    }
```

- Esta função obtém os valores das variáveis da instância de initialPage e preenche os campos correspondentes no formulário;

```
private void compareQuantWithTagRead()
    {
        int quant = Convert.ToInt32(textQuantidade.Text);
        if (tagCount >= quant && quant != 0)
        {
            Disconnect();
            MessageBox.Show("A quantidade pretendida para este produto foi completa. A retornar para selecionar novos dados");
            this.Close();
        }
    }
```

- Esta função verifica se a quantidade de tags lidas é igual ou superior à quantidade definida para um produto. Se essa condição for verdadeira e a quantidade definida não for zero, então a função desconecta do leitor RFID, exibe uma mensagem informativa ao utilizador e fecha o formulário atual;

```
private string postByAntenna(string tag)
    {
        string posto = "";

        int antena = db.rfidGetAntenaId(tag);

        switch (antena)
        {
            case 1:
                posto = "LINHA 1 EXPEDIÇÃO";
                break;
            case 2:
                posto = "LINHA 2 EXPEDIÇÃO";
                break;
            case 3:
                posto = "LINHA 3 EXPEDIÇÃO";
                break;
            case 4:
                posto = "LINHA 4 EXPEDIÇÃO";
                break;
            default:
                break;
        }

        return posto;
    }
```

- Esta função recebe como argumento o código da etiqueta RFID e utiliza esse código para determinar em que antena a etiqueta foi detetada. Com base nessa informação, o posto correspondente é atribuído à etiqueta e devolvido como resultado. Se a antena não estiver mapeada para um posto específico, é devolvida uma string vazia;

Classe – “dbHelper”

Variáveis e instâncias:

private string jsonPath = @".\config_DB_JSON.json";

- String para indicar o caminho para o ficheiro JSON;

Funções:

```
public Rootobject readConfig(string filePath)
{
   string fileName = File.ReadAllText(filePath);
   return JsonConvert.DeserializeObject<Rootobject>(fileName);
}
```

- Esta função recebe o caminho do ficheiro JSON como argumento. Em seguida, lê o conteúdo desse ficheiro e utiliza o método DeserializeObject da classe JsonConvert para desserializar o conteúdo JSON em um objeto da classe Rootobject. Finalmente, retorna o objeto desserializado;

```
public string[] getJsonTableNames()
{
   string jsonText = File.ReadAllText(jsonPath);

   dynamic config = JsonConvert.DeserializeObject(jsonText);

   List<string> tableNamesList = new List<string>();

   foreach (var tableName in config.tableNames)
   {
      tableNamesList.Add(tableName.Value.ToString());
   }
   return tableNamesList.ToArray();
}
```

- Esta função lê o conteúdo do ficheiro JSON, desserializa-o em um objeto dinâmico e itera sobre as propriedades desse objeto para extrair os nomes das tabelas. Esses nomes são então armazenados numa lista e convertidos num array de strings antes de serem retornados;

```
private SqlConnection ConnectBd()
{
   //SqlConnection conn = new SqlConnection("Data Source=localhost;Initial Catalog=dbFase2;User Id=sa;Password=sa");
   var dbconfig = readConfig(jsonPath);

   string sqlCommandStr = $"Data Source={dbconfig.dataSource};Initial Catalog={dbconfig.DBname};User Id={dbconfig.userID};Password={dbconfig.password}";

   SqlConnection conn = new SqlConnection(sqlCommandStr);

   try
   {
      conn.Open();
      return conn;
   }
   catch (SqlException e)
   {
      MessageBox.Show(e.Message);
      return null;
   }
}
```

- Esta função lê as configurações de conexão do ficheiro JSON, constrói uma string de conexão com base nessas configurações e, em seguida, tenta abrir uma conexão com a base de dados usando essa string de conexão. Se a conexão for bem-sucedida, retorna a conexão aberta; caso contrário, exibe uma mensagem de erro e retorna null;

```
public DataTable allTablesShowAllBd(string tableName)
{
   DataTable dt = new DataTable();

   var dbconfig = readConfig(jsonPath);

   using (SqlConnection conn = ConnectBd())
   {
      string sqlCommandStr = $"SELECT * FROM {tableName}";
      SqlCommand cmd = new SqlCommand(sqlCommandStr, conn);
      //SqlCommand cmd = new SqlCommand("SELECT * FROM dbo.tableReadIInfo", conn);
      SqlDataAdapter da = new SqlDataAdapter(cmd);
      da.Fill(dt);
   }

   return dt;
}
```

- Esta função começa por criar uma nova DataTable para armazenar os dados da tabela. Em seguida, lê as configurações de conexão do ficheiro JSON e abre uma conexão com a base de dados usando essas configurações. Depois, constrói uma string de comando SQL para selecionar todos os dados da tabela especificada e executa essa consulta. Os resultados da consulta são preenchidos na DataTable usando um SqlDataAdapter e, por fim, a DataTable preenchida é retornada;

```
public bool rfidCanInsert(int db_antenaId, string db_tagId)
{
   var dbconfig = readConfig(jsonPath);

   int antenaId = db_antenaId;
   string tagId = db_tagId;

   bool tagExists = false;

   using (SqlConnection conn = ConnectBd())
   {
      using (SqlCommand checkCmd = new SqlCommand($"SELECT COUNT(*) FROM
{dbconfig.tableNames.table1} WHERE Antenna_id = @antenaId AND Tag_id = @tagId", conn))
      {
         checkCmd.Parameters.AddWithValue("@antenaId", antenaId);
         checkCmd.Parameters.AddWithValue("@tagId", tagId);
         tagExists = (int)checkCmd.ExecuteScalar() > 0;
      }
   }

   return !tagExists;
}
```

- Esta função começa por ler as configurações de conexão do ficheiro JSON. Em seguida, recebe o ID da antena e o ID da etiqueta como argumentos. A função verifica se o registo já existe na tabela da base de dados RFID com base no ID da antena e no ID da etiqueta. Se a etiqueta não existir na base de dados, a função retorna verdadeiro, indicando que o registo pode ser inserido. Se a etiqueta já existir, a função retorna falso, indicando que o registo não pode ser inserido;

```
public void rfidAddToBd(int db_antenaId, string db_tagId, string db_epcId, int db_rssiValueH, string
db_dataLeitura, string db_horaLeitura)

{

   var dbconfig = readConfig(jsonPath);



   int antenaId = db_antenaId;

   string tagId = db_tagId;

   string epcId = db_epcId;

   int rssi = db_rssiValueH;


   string date = db_dataLeitura;

   string hour = db_horaLeitura;


   DateTime Tdate = DateTime.ParseExact(date, "dd/MM/yyyy", null);

   DateTime Thours = DateTime.ParseExact(hour, "HH:mm:ss", null);



   using (SqlConnection conn = ConnectBd())

   {

      if (rfidCanInsert(db_antenaId, db_tagId))

      {

         string sqlCommandStr = $"INSERT INTO {dbconfig.tableNames.table1} (Antenna_id, Tag_id,
EPC_id, RSSI, Data_Leitura, Hora_Leitura)" +

         $" VALUES (@antenaId, @tagId, @epcId, @rssi, @date, @hour)";

         SqlCommand cmd = new SqlCommand(sqlCommandStr, conn);

         //SqlCommand cmd = new SqlCommand("INSERT INTO dbo.tableReadlInfo (idProdOrder, tagId,
readDate, readHour, post) VALUES (@id, @atagId, @aDate, @aHour, @aPost)", conn);

         cmd.Parameters.AddWithValue("@antenaId", antenaId);

         cmd.Parameters.AddWithValue("@tagId", tagId);

         cmd.Parameters.AddWithValue("@epcId", epcId);

         cmd.Parameters.AddWithValue("@rssi", rssi);

         cmd.Parameters.AddWithValue("@date", Tdate);

         cmd.Parameters.AddWithValue("@hour", Thours);
```

```
        try
        {
           cmd.ExecuteNonQuery();
        }
        catch (SqlException e)
        {
           MessageBox.Show(e.Message);
        }
     }
  }
}
```

- Esta função começa por ler as configurações de conexão do ficheiro JSON. Em seguida, recebe os parâmetros necessários para inserir um novo registo na tabela da base de dados RFID. Antes de inserir o registo, verifica se ele pode ser inserido utilizando a função rfidCanInsert. Se o registo puder ser inserido, define o comando SQL para inserção e executa-o na base de dados. Se ocorrer algum erro durante a execução do comando SQL, uma mensagem de erro é mostrada numa caixa de diálogo;

```
public int rfidGetAntenaId(string tagId)
{
   var dbconfig = readConfig(jsonPath);

   string tag = tagId;

   int antenaId;

   using (SqlConnection conn = ConnectBd())
   {
      using (SqlCommand checkCmd = new SqlCommand($"SELECT Antenna_id FROM
{dbconfig.tableNames.table1} WHERE Tag_id = @tagId", conn))
      {
         checkCmd.Parameters.AddWithValue("@tagId", tag);
         antenaId = (int)checkCmd.ExecuteScalar();
      }
   }

   return antenaId;
}
```

- Esta função começa por ler as configurações de conexão do ficheiro JSON. Em seguida, recebe a etiqueta RFID como parâmetro e utiliza-a para pesquisar na base de dados o ID da antena associado a essa etiqueta. Utiliza um comando SQL SELECT para realizar a pesquisa e executa-o na base de dados. O resultado é armazenado na variável antenaId e posteriormente retornado pela função;

```
public bool modelCanInsert(string db_idProdOrder)
{
   var dbconfig = readConfig(jsonPath);

   string idProdOrder = db_idProdOrder;

   bool idExists = false;

   using (SqlConnection conn = ConnectBd())
   {
      using (SqlCommand checkCmd = new SqlCommand($"SELECT COUNT(*) FROM
{dbconfig.tableNames.table2} WHERE idProdOrder = @idProdOrder", conn))
      {
         checkCmd.Parameters.AddWithValue("@idProdOrder", idProdOrder);
         idExists = (int)checkCmd.ExecuteScalar() > 0;
      }
   }

   return !idExists;
}
```

- Esta função começa por ler as configurações de conexão do ficheiro JSON. Em seguida, recebe o ID da ordem de produto como parâmetro e utiliza-o para verificar se já existe na tabela correspondente da base de dados. Utiliza um comando SQL SELECT COUNT(*) para contar o número de ocorrências do ID na tabela e retorna verdadeiro se o resultado for zero (indicando que o ID ainda não existe);

```
public void modelAddToBd(string db_idProdOrder, string db_serieId, string db_article, int db_quantity,
int db_capacDia)

{

   var dbconfig = readConfig(jsonPath);



   string idProdOrder = db_idProdOrder;

   string serieId = db_serieId;

   string article = db_article;

   int quantity = db_quantity;

   int capacidadeDia = db_capacDia;



   using (SqlConnection conn = ConnectBd())

   {

      if(modelCanInsert(db_idProdOrder))

      {

         string sqlCommandStr = $"INSERT INTO {dbconfig.tableNames.table3} (idProdOrder, serie_ID,
article, quantity, capacDia)" +

         $" VALUES (@idProdOrder, @serieId, @article, @quantity, @capacidadeDia)";

         SqlCommand cmd = new SqlCommand(sqlCommandStr, conn);

         //SqlCommand cmd = new SqlCommand("INSERT INTO dbo.tableReadlInfo (idProdOrder, tagId,
readDate, readHour, post) VALUES (@id, @atagId, @aDate, @aHour, @aPost)", conn);

         cmd.Parameters.AddWithValue("@idProdOrder", idProdOrder);

         cmd.Parameters.AddWithValue("@serieId", serieId);

         cmd.Parameters.AddWithValue("@article", article);

         cmd.Parameters.AddWithValue("@quantity", quantity);

         cmd.Parameters.AddWithValue("@capacidadeDia", capacidadeDia);

         try

         {

            cmd.ExecuteNonQuery();

         }

         catch (SqlException e)

         {
```

```
            MessageBox.Show(e.Message);
        }
    }
  }
}
```

- Esta função começa por ler as configurações de conexão do ficheiro JSON. Em seguida, recebe os detalhes do modelo de produto como parâmetros e utiliza-os para construir e executar um comando SQL INSERT na tabela correspondente da base de dados. Antes de inserir os dados, verifica se o ID da ordem de produto já existe na tabela usando a função modelCanInsert. Se o ID ainda não existir, os dados são inseridos na tabela;

```
public bool readCanInsert(string tagId, int antenaId)
    {
        var dbconfig = readConfig(jsonPath);

        string tag = tagId;
        int antena = antenaId;

        bool tagExists = false;

        using (SqlConnection conn = ConnectBd())
        {
            using (SqlCommand checkCmd = new SqlCommand($"SELECT COUNT(*) FROM
{dbconfig.tableNames.table2} WHERE tagId = @tagId AND idAntena = @antenaId", conn))
            {
                checkCmd.Parameters.AddWithValue("@tagId", tag);
                checkCmd.Parameters.AddWithValue("@antenaId", antena);
                tagExists = (int)checkCmd.ExecuteScalar() > 0;
            }
        }

        return !tagExists;
    }
```

- Esta função começa por ler as configurações de conexão do ficheiro JSON. Em seguida, recebe os identificadores da etiqueta RFID e da antena como parâmetros e utiliza-os para construir e executar uma consulta SQL SELECT na tabela correspondente da base de dados. Se a combinação de etiqueta e antena não existir na tabela, a função retorna verdadeiro, indicando que a leitura pode ser inserida na base de dados;

```
public void readAddToBd(string db_serie_ID, string db_tagId, string db_postInf, string db_idProdOrder,
int db_idAntena, int db_quantity, string db_readDate, string db_readHour)

{

   var dbconfig = readConfig(jsonPath);


   string serieId = db_serie_ID;

   string tagId = db_tagId;

   string postInfo = db_postInf;

   string idProdOrder = db_idProdOrder;

   int idAntena = db_idAntena;

   int quantity = db_quantity;

   string readDate = db_readDate;

   string readHour = db_readHour;


   DateTime Tdate = DateTime.ParseExact(readDate, "dd/MM/yyyy", null);

   DateTime Thours = DateTime.ParseExact(readHour, "HH:mm:ss", null);


   using (SqlConnection conn = ConnectBd())

   {

      string sqlCommandStr = $"INSERT INTO {dbconfig.tableNames.table2} (serie_ID, tagId, postInf,
idProdOrder, idAntena, quantity, readDate, readHour)" +

      $" VALUES (@serieId, @tagId, @postInfo, @idProdOrder, @idAntena, @quantity, @readDate,
@readHour)";

      SqlCommand cmd = new SqlCommand(sqlCommandStr, conn);

      cmd.Parameters.AddWithValue("@serieId", serieId);

      cmd.Parameters.AddWithValue("@tagId", tagId);

      cmd.Parameters.AddWithValue("@postInfo", postInfo);

      cmd.Parameters.AddWithValue("@idProdOrder", idProdOrder);

      cmd.Parameters.AddWithValue("@idAntena", idAntena);

      cmd.Parameters.AddWithValue("@quantity", quantity);

      cmd.Parameters.AddWithValue("@readDate", Tdate);

      cmd.Parameters.AddWithValue("@readHour", Thours);

      try

      {
```

```
            cmd.ExecuteNonQuery();
        }
        catch (SqlException e)
        {
            MessageBox.Show(e.Message);
        }
    }
}
```

- Esta função recebe vários parâmetros representando os dados da leitura, como o ID da série, a etiqueta RFID, a informação de localização, o ID do pedido de produção, o ID da antena, a quantidade, a data e a hora da leitura. Em seguida, constrói e executa uma instrução SQL INSERT para adicionar os dados à tabela de leituras na base de dados. Se ocorrer algum erro durante a execução da instrução SQL, a função mostra uma mensagem de erro;

```
public DataTable ModelSearch(string tableName, string columnName1, string columnValue1)
    {
        return ModelSearchInternal(tableName, columnName1, columnValue1);
    }
```

- Esta função recebe o nome da tabela, o nome da coluna e o valor a ser pesquisado. Em seguida, chama uma função interna chamada ModelSearchInternal para executar a pesquisa real. Esta abordagem permite que a lógica de pesquisa seja encapsulada em uma função interna, tornando o código mais modular e fácil de manter;

```
public DataTable ModelSearch(string tableName, string columnName1, string columnValue1, string columnName2, string columnValue2)
    {
        return ModelSearchInternal(tableName, columnName1, columnValue1, columnName2, columnValue2);
    }
```

- Esta função recebe o nome da tabela e pares de nome/valor de coluna para duas condições de pesquisa. Em seguida, chama uma função interna chamada ModelSearchInternal passando esses parâmetros para realizar a pesquisa real. Esta versão da função oferece mais flexibilidade ao permitir a especificação de duas condições de pesquisa em vez de apenas uma;

```
public DataTable ModelSearch(string tableName, string columnName1, string columnValue1, string columnName2, string columnValue2, string columnName3, string columnValue3)
    {
        return ModelSearchInternal(tableName, columnName1, columnValue1, columnName2, columnValue2, columnName3, columnValue3);
    }
```

- Esta função recebe o nome da tabela e pares de nome/valor de coluna para três condições de pesquisa. Em seguida, chama uma função interna chamada ModelSearchInternal passando esses parâmetros para realizar a pesquisa real. Esta versão da função oferece ainda mais flexibilidade, permitindo especificar três condições de pesquisa em vez de apenas uma ou duas;

```
private DataTable ModelSearchInternal(string tableName, params object[] parameters)
    {
        DataTable dt = new DataTable();
        var dbconfig = readConfig(jsonPath);

        using (SqlConnection conn = ConnectBd())
        {
            string query = $"SELECT * FROM {tableName} WHERE ";

            for (int i = 0; i < parameters.Length; i += 2)
            {
                query += $"{parameters[i]} = @{parameters[i]}";

                if (i + 2 < parameters.Length)
                {
                    query += " AND ";
                }
            }

            using (SqlCommand checkCmd = new SqlCommand(query, conn))
            {
                for (int i = 0; i < parameters.Length; i += 2)
                {
                    checkCmd.Parameters.AddWithValue($"@{parameters[i]}", parameters[i + 1]);
                }

                SqlDataAdapter adapter = new SqlDataAdapter(checkCmd);
                adapter.Fill(dt);
            }
        }
        return dt;
    }
```

- Esta função realiza uma pesquisa na tabela especificada com base nos parâmetros fornecidos. Os parâmetros são passados como argumentos de comprimento variável, permitindo que você especifique o número desejado de condições de pesquisa. Ele constrói uma consulta SQL dinâmica com base nessas condições e preenche um DataTable com os resultados da consulta;

```
public DataTable ModelDelete(string tableName, string columnName1, string columnValue1)
    {
        return ModelDeleteInternal(tableName, columnName1, columnValue1);
    }
```

- Esta função ModelDelete serve como uma interface para a função interna ModelDeleteInternal, que executa a exclusão de registros na tabela especificada com base nos critérios fornecidos;

```
public DataTable ModelDelete(string tableName, string columnName1, string columnValue1, string columnName2, string columnValue2)
    {
        return ModelDeleteInternal(tableName, columnName1, columnValue1, columnName2, columnValue2);
    }
```

- Esta função ModelDelete funciona como uma interface para a função interna ModelDeleteInternal, que realiza a exclusão de registos na tabela especificada, com base nos critérios fornecidos;

```
public DataTable ModelDelete(string tableName, string columnName1, string columnValue1, string columnName2, string columnValue2, string columnName3, string columnValue3)
    {
        return ModelDeleteInternal(tableName, columnName1, columnValue1, columnName2, columnValue2, columnName3, columnValue3);
     }
```

- Esta função ModelDelete funciona como uma interface para a função interna ModelDeleteInternal, que realiza a exclusão de registos na tabela especificada, com base nos critérios fornecidos;

```
private DataTable ModelDeleteInternal(string tableName, params object[] parameters)
    {
        DataTable dt = new DataTable();
        var dbconfig = readConfig(jsonPath);

        using (SqlConnection conn = ConnectBd())
        {
            string query = $"DELETE FROM {tableName} WHERE ";

            for (int i = 0; i < parameters.Length; i += 2)
            {
                query += $"{parameters[i]} = @{parameters[i]}";

                if (i + 2 < parameters.Length)
                {
                    query += " AND ";
                }
            }

            using (SqlCommand checkCmd = new SqlCommand(query, conn))
            {
                for (int i = 0; i < parameters.Length; i += 2)
                {
                    checkCmd.Parameters.AddWithValue($"@{parameters[i]}", parameters[i + 1]);
                }

                SqlDataAdapter adapter = new SqlDataAdapter(checkCmd);
                adapter.Fill(dt);
            }
        }
        return dt;
    }
```

- Esta função ModelDeleteInternal é responsável por executar a eliminação de registos numa tabela específica com base nos critérios fornecidos

.

Esta função fornece uma maneira flexível de eliminar registos de uma tabela, permitindo especificar múltiplos parâmetros para eliminar;

Form – “editorDeTabela”

Variáveis e instâncias:

dbHelper db = new dbHelper();

- Instância a classe dbHelper como um objeto "db";

private string atualTableName;

- Cria uma string "atualTableName";

private string searchVar1;

- Cria uma string "searchVar1";

private string searchVar2;

- Cria uma string "searchVar2";

private string searchVar3;

- Cria uma string "searchVar3";

private string addVar1;

- Cria uma string "addVar1";

private string addVar2;

- Cria uma string "addVar2";

private string addVar3;

- Cria uma string "addVar3";

private string deleteVar1;

- Cria uma string "deleteVar1";

private string deleteVar2;

- Cria uma string "deleteVar2";

private string deleteVar3;

- Cria uma string "deleteVar3";

private string infoIdOrdProd;

- Cria uma string "infoIdOrdProd";

private int infoQuantidade;

- Cria um int "infoQuantidade";

private string infoIdSerie;

- Cria uma string "infoIdSerie";

private int rssi = 0;

- Cria um int "rssi" com o valor de 0;

private int antenaId;

- Cria um int "antenaId";

private string tagId;

- Cria uma string "tagId";

private string epcId;

- Cria uma string "epcId";

private string data;

- Cria uma string "data";

private string hora;

- Cria uma string "hora";

Funções:

```
private void editorDeTabela_Load(object sender, EventArgs e)
    {
      chooseBd_comboBox.DropDownStyle = ComboBoxStyle.DropDownList;
      putTableNames();
      timer1.Start();
      if (chooseBd_comboBox.SelectedItem == null)
      {
        chooseBd_comboBox.SelectedIndex = 0;
        chooseBd_comboBox.Enabled = false;
      }
    }
```

- Este método configura o formulário quando é carregado, garantindo que o ComboBox é preenchido com os nomes das tabelas, o estilo de dropdown é definido e um item é selecionado por padrão. Além disso, inicia o temporizador;

```
private void getVarsModelo()
    {
      infoIdOrdProd = initialPage.instance.idProdOrdem;
      infoQuantidade = initialPage.instance.quantidade;
      infoIdSerie = initialPage.instance.idSerie;
    }
```

- Este método basicamente extrai informações relevantes da instância da classe initialPage e armazena-as em variáveis locais para uso posterior no código;

```
private void putTableNames()
    {
       string[] tableNames = db.getJsonTableNames();

       foreach (string tableName in tableNames)
       {
          chooseBd_comboBox.Items.Add(tableName);
       }
    }
```

- Este método permite que o utilizador selecione uma tabela da base de dados a partir de um menu suspenso, facilitando a interação com a aplicação;

```
private void chooseBd_SelectedIndexChanged(object sender, EventArgs e)
    {
      if (chooseBd_comboBox.SelectedItem != null)
      {
        atualTableName = chooseBd_comboBox.SelectedItem.ToString();

        tabelaAtual_label.Text = "Tabela: " + atualTableName;

        UpdateDataGrid();

        string searchOption = chooseBd_comboBox.SelectedItem.ToString();

        if (searchOption == "dbo.LeituraTag_FX96000")
        {
          labelSearch1.Text = "ID da Tag:";
          labelSearch1.Location = new Point(18, labelSearch1.Location.Y);
          labelSearch2.Text = "EPC da Tag:";
          labelSearch2.Location = new Point(8, labelSearch2.Location.Y);
          labelSearch3.Text = "Antena:";
          labelSearch3.Location = new Point(32, labelSearch3.Location.Y);
        }
      }
    }
```

- Obtém o nome da tabela selecionada atualmente a partir do ComboBox e atribui-o à variável atualTableName. Em seguida, atualiza o texto do rótulo tabelaAtual_label para exibir o nome da tabela atual.Verifica se a opção selecionada corresponde a uma tabela específica. Se a tabela selecionada for "dbo.LeituraTag_FX96000", altera os rótulos e a posição dos rótulos na interface do utilizador para corresponder aos campos de pesquisa relevantes para essa tabela. Este evento garante que a interface do utilizador é atualizada dinamicamente com base na tabela selecionada, facilitando a interação do utilizador com a aplicação;

```
private void UpdateDataGrid()
    {
       DataTable dadosDaTabela = db.allTablesShowAllBd(atualTableName);
       dataGrid_DB.DataSource = dadosDaTabela;
    }
```

- A função UpdateDataGrid() atualiza o conteúdo do DataGridView com os dados da tabela atualmente selecionada.;

```
private void GetVarsSearch()
    {
       searchVar1 = search_TextBox1.Text;
       searchVar2 = search_TextBox2.Text;
       searchVar3 = search_TextBox3.Text;

    }
```

- A função GetVarsSearch() obtém os valores inseridos pelo utilizador em três caixas de texto de pesquisa e armazena esses valores em três variáveis;

```
private void GetVarsAdd()
    {
       addVar1 = add_textBox1.Text;
       addVar2 = add_textBox2.Text;
       addVar3 = add_textBox3.Text;
    }
```

- Esta função, denominada GetVarsAdd(), obtém os valores inseridos pelo utilizador em três caixas de texto de adição e armazena esses valores em três variáveis;

```
private void GetVarsDelete()
    {
       deleteVar1 = delete_textBox1.Text;
       deleteVar2 = delete_textBox2.Text;
       deleteVar3 = delete_textBox3.Text;
    }
```

- Recolhe os valores inseridos em três caixas de texto de eliminação e guarda esses valores em três variáveis;

```
private DataTable DeleteInDb(string tableName, string columnName1, string columnName2, string
columnName3)
    {
        DataTable result = null;


        if (!string.IsNullOrEmpty(deleteVar1) && !string.IsNullOrEmpty(deleteVar2) &&
!string.IsNullOrEmpty(deleteVar3))
        {
            result = db.ModelDelete(tableName, columnName1, deleteVar1, columnName2, deleteVar2,
columnName3, deleteVar3);
        }
        else if (!string.IsNullOrEmpty(deleteVar1) && !string.IsNullOrEmpty(deleteVar2))
        {
            result = db.ModelDelete(tableName, columnName1, deleteVar1, columnName2, deleteVar2);
        }
        else if (!string.IsNullOrEmpty(deleteVar1) && !string.IsNullOrEmpty(deleteVar3))
        {
            result = db.ModelDelete(tableName, columnName1, deleteVar1, columnName3, deleteVar3);
        }
        else if (!string.IsNullOrEmpty(deleteVar2) && !string.IsNullOrEmpty(deleteVar3))
        {
            result = db.ModelDelete(tableName, columnName2, deleteVar2, columnName3, deleteVar3);
        }
        else if (!string.IsNullOrEmpty(deleteVar1))
        {
            result = db.ModelDelete(tableName, columnName1, deleteVar1);
        }
        else if (!string.IsNullOrEmpty(deleteVar2))
        {
            result = db.ModelDelete(tableName, columnName2, deleteVar2);
        }
        else if (!string.IsNullOrEmpty(deleteVar3))
        {
            result = db.ModelDelete(tableName, columnName3, deleteVar3);
```

```
        }


        return result;

    }
```

- Esta função, DeleteInDb, recebe o nome da tabela e os nomes das colunas a serem usados como critérios de exclusão. Ela verifica se os valores nas variáveis deleteVar1, deleteVar2 e deleteVar3 não estão vazios e, em seguida, executa a exclusão de acordo com os valores disponíveis. Se um valor estiver presente em uma variável, será utilizado como critério de exclusão para a coluna correspondente. O resultado da operação de exclusão é armazenado numa tabela de dados e retornado. Se nenhuma operação de exclusão for possível devido à falta de valores, a função retorna null.;

```
private DataTable SearchInDatabase(string tableName, string columnName1, string columnName2,
string columnName3)
    {
        DataTable result = null;

        if (!string.IsNullOrEmpty(searchVar1) && !string.IsNullOrEmpty(searchVar2) &&
!string.IsNullOrEmpty(searchVar3))
        {
            result = db.ModelSearch(tableName, columnName1, searchVar1, columnName2, searchVar2,
columnName3, searchVar3);
        }
        else if (!string.IsNullOrEmpty(searchVar1) && !string.IsNullOrEmpty(searchVar2))
        {
            result = db.ModelSearch(tableName, columnName1, searchVar1, columnName2, searchVar2);
        }
        else if (!string.IsNullOrEmpty(searchVar1) && !string.IsNullOrEmpty(searchVar3))
        {
            result = db.ModelSearch(tableName, columnName1, searchVar1, columnName3, searchVar3);
        }
        else if (!string.IsNullOrEmpty(searchVar2) && !string.IsNullOrEmpty(searchVar3))
        {
            result = db.ModelSearch(tableName, columnName2, searchVar2, columnName3, searchVar3);
        }
        else if (!string.IsNullOrEmpty(searchVar1))
        {
            result = db.ModelSearch(tableName, columnName1, searchVar1);
        }
        else if (!string.IsNullOrEmpty(searchVar2))
        {
            result = db.ModelSearch(tableName, columnName2, searchVar2);
        }
        else if (!string.IsNullOrEmpty(searchVar3))
        {
            result = db.ModelSearch(tableName, columnName3, searchVar3);
```

```
        }


        return result;
    }
```

- Esta função, SearchInDatabase, é responsável por realizar uma pesquisa na base de dados. Ela recebe o nome da tabela e os nomes das colunas que serão utilizados como critérios de pesquisa. A função verifica se os valores nas variáveis searchVar1, searchVar2 e searchVar3 não estão vazios e, em seguida, executa a pesquisa de acordo com os valores disponíveis. Se um valor estiver presente numa variável, será utilizado como critério de pesquisa para a coluna correspondente. O resultado da pesquisa é armazenado numa tabela de dados e retornado. Se nenhuma pesquisa for possível devido à falta de valores, a função retorna null;

```
private void timer1_Tick(object sender, EventArgs e)
    {
        Data_textBox.Enabled = false;
        Hora_textBox.Enabled = false;
        Data_textBox.Text = DateTime.Now.ToString("dd/MM/yyyy");
        Hora_textBox.Text = DateTime.Now.ToString("HH:mm:ss");
    }
```

- Esta função é chamada periodicamente pelo temporizador timer1. Ela desabilita a edição dos campos de texto Data_textBox e Hora_textBox e, em seguida, define o texto desses campos com a data e hora atuais, formatadas como "dd/MM/yyyy" e "HH:mm:ss", respetivamente. Isso garante que esses campos mostrem sempre a data e hora atual e impeçam o utilizador de modificá-los manualmente;

```
private void checkBox1_CheckedChanged(object sender, EventArgs e)
    {
      if (checkBox1.Checked)
      {
        UpdateDataGrid();
        checkBox1.Checked = false;
      }
    }
```

- Esta função é um manipulador de evento para o evento CheckedChanged do controlo checkBox1. Ela é acionada sempre que o estado de seleção do checkBox1 muda. Quando o checkBox1 é marcado (checked), a função chama a função UpdateDataGrid() para atualizar a grelha de dados e, em seguida, desmarca automaticamente o checkBox1, evitando que ele permaneça marcado;

```
private void searchButton_Click(object sender, EventArgs e)
    {
        GetVarsSearch();

        string searchOption = chooseBd_comboBox.SelectedItem.ToString();

        DataTable tableData = null;

        if (chooseBd_comboBox.SelectedItem != null)
        {
            if (searchOption == "dbo.LeituraTag_FX96000")
            {
                tableData = SearchInDatabase(searchOption, "Tag_id", "EPC_id", "Antenna_id");
            }
            else
            {
                MessageBox.Show("Tabela não encontrada.");
            }
        }
        else
        {
            MessageBox.Show("Selecione uma tabela.");
        }

        if (tableData != null && tableData.Rows.Count > 0)
        {
            dataGrid_DB.DataSource = tableData;
        }
        else
        {
            MessageBox.Show("Nenhum resultado encontrado.");
        }
    }
```

- Esta função é um manipulador de evento para o clique no botão de pesquisa (searchButton). Quando o botão é clicado, a função obtém as variáveis de pesquisa (searchVar1, searchVar2 e searchVar3) chamando a função GetVarsSearch(). Em seguida, verifica se uma tabela foi selecionada no chooseBd_comboBox. Se uma tabela foi selecionada, verifica se é a tabela específica "dbo.LeituraTag_FX96000". Se for essa tabela, chama a função SearchInDatabase() para pesquisar na base de dados com base nos valores dos campos Tag_id, EPC_id e Antenna_id. Se a pesquisa retornar resultados, os dados são exibidos na grelha de dados (dataGrid_DB). Se nenhum resultado for encontrado ou se nenhuma tabela for selecionada, são exibidas mensagens apropriadas ao utilizador;

```
private void addButton_Click(object sender, EventArgs e)
    {
        //se os campos não estiverem todos preenchidos manda uma mensagem de erro
        if (string.IsNullOrEmpty(addVar1) || string.IsNullOrEmpty(addVar2) ||
string.IsNullOrEmpty(addVar3))
        {
            MessageBox.Show("Preencha todos os campos primeiro");
            return;
        }
        else
        {
            getVarsModelo();
            GetVarsAdd();
            getTagVars();
            addDbRfid();
            addDbRead();
        }
    }
```

- Esta função é um manipulador de evento para o clique no botão de adição (addButton). Quando o botão é clicado, verifica se os campos de entrada estão todos preenchidos. Se algum campo estiver vazio, exibe uma mensagem de erro e interrompe a execução. Caso contrário, obtém as variáveis de modelo chamando a função getVarsModelo(), as variáveis de adição chamando a função GetVarsAdd(), as variáveis de tag chamando a função getTagVars(). Em seguida, adiciona os dados à base de dados chamando as funções addDbRfid() e addDbRead();

```
private void deleteButton_Click(object sender, EventArgs e)
    {
        GetVarsDelete();

        string searchOption = chooseBd_comboBox.SelectedItem.ToString();

        DataTable tableData = null;

        if (chooseBd_comboBox.SelectedItem != null)
        {
            if (searchOption == "dbo.LeituraTag_FX96000")
            {
                tableData = DeleteInDb(searchOption, "Tag_id", "EPC_id", "Antenna_id");
                MessageBox.Show("Deletado com sucesso.");
            }
            else
            {
                MessageBox.Show("Tabela não encontrada.");
            }
        }
        else
        {
            MessageBox.Show("Selecione uma tabela.");
        }

        if (tableData != null && tableData.Rows.Count > 0)
        {
            dataGrid_DB.DataSource = tableData;
        }

        UpdateDataGrid();
    }
```

- Esta função é um manipulador de evento para o clique no botão de exclusão (deleteButton). Quando o botão é clicado, obtém as variáveis de exclusão chamando a função GetVarsDelete(). Em seguida, verifica se uma tabela foi selecionada no ComboBox. Se uma tabela foi selecionada, verifica se é a tabela correta (dbo.LeituraTag_FX96000). Se for, chama a função DeleteInDb() para excluir os registros da tabela especificada. Caso contrário, exibe uma mensagem informando que a tabela não foi encontrada. Se nenhum resultado for encontrado ou nenhuma tabela estiver selecionada, exibe uma mensagem correspondente. Por fim, atualiza o DataGridView chamando a função UpdateDataGrid();

```
private void getTagVars()
    {
       tagId = addVar1;
       epcId = addVar2;
       antenaId = Convert.ToInt32(addVar3);
    }
```

- Esta função getTagVars() é responsável por atribuir os valores dos campos de entrada (addVar1, addVar2 e addVar3) às variáveis tagId, epcId e antenaId, respectivamente;

```
private void addDbRfid()
    {
       data = DateTime.Now.ToString("dd/MM/yyyy");
       hora = DateTime.Now.ToString("HH:mm:ss");
       db.rfidAddToBd(antenaId, tagId, epcId, rssi, data, hora);
    }
```

- Nesta função addDbRfid(), são capturadas a data e a hora atuais e, em seguida, é chamada a função rfidAddToBd() do objeto db, passando como argumentos o antenaId, o tagId, o epcId, o rssi, a data e a hora;

```
private void addDbRead()
    {
        string postoString = postByAntenna(tagId);
        db.readAddToBd(infoIdSerie, tagId, postoString, infoIdOrdProd, antenaId, infoQuantidade, data,
hora);
    }
```

- Nesta função addDbRead(), é determinado o posto com base no tagId usando a função postByAntenna(). Em seguida, são passados os valores relevantes para a função readAddToBd() do objeto db, que adiciona esses dados à tabela de leitura na base de dados;

```
private string postByAntenna(string tag)
    {
        string posto = "";

        int antena = db.rfidGetAntenaId(tag);

        switch (antena)
        {
            case 1:
                posto = "LINHA 1 EXPEDIÇÃO";
                break;
            case 2:
                posto = "LINHA 2 EXPEDIÇÃO";
                break;
            case 3:
                posto = "LINHA 3 EXPEDIÇÃO";
                break;
            case 4:
                posto = "LINHA 4 EXPEDIÇÃO";
                break;
            default:
                break;
        }

        return posto;
    }
}
```

- Nesta função postByAntenna(tag), é determinado o posto com base na antena que leu a tag. Primeiro, é obtido o número da antena usando a função rfidGetAntenaId(tag) do objeto db. Depois, é feita uma verificação usando um switch-case para atribuir o posto correspondente ao número da antena. O posto é então retornado;

Form – “avisoForm”

Variáveis e instâncias:

dbHelper db = new dbHelper();

-Instância a classe "dbHelper" através do objeto "db";

public string idProdOrdem;

- Cria uma string "idProdOrdem";

public string artigo;

- Cria uma string "artigo";

public int quantidade;

- Cria um int "quantidade";

public int quantidadeMin;

- Cria um int "quantidadeMin";

public string idSerie;

- Cria uma string "idSerie";

Funções:

```
private void cancel_button_Click(object sender, EventArgs e)
    {
        db.modelDelete(idProdOrdem, idSerie);



        if (initialPage.instance == null)
        {
            initialPage form = new initialPage();
            form.Show();
        }
        else
        {
            initialPage.instance.Show();
        }


        this.Close();
    }
```

- Neste método cancel_button_Click, primeiro é chamado o método modelDelete do objeto db para excluir um modelo específico com base nos identificadores idProdOrdem e idSerie. Em seguida, é verificado se a instância de initialPage é nula. Se for, é criada uma nova instância e exibida. Caso contrário, a instância existente é exibida. Por fim, a janela atual é fechada;

```
private void getVars()
    {
       idProdOrdem = initialPage.instance.idProdOrdem;
       artigo = initialPage.instance.artigo;
       quantidade = initialPage.instance.quantidade;
       quantidadeMin = initialPage.instance.quantidadeMin;
       idSerie = initialPage.instance.idSerie;
    }
```

- Neste método getVars, os valores dos campos da instância de initialPage são atribuídos a variáveis locais para uso posterior. Esses valores incluem idProdOrdem, artigo, quantidade, quantidadeMin e idSerie;

```
private void rfid_button_Click(object sender, EventArgs e)
    {
       showRfidForm();
       this.Close();
    }
```

- No método rfid_button_Click, a função showRfidForm() é chamada para exibir um formulário relacionado ao RFID, e em seguida, a janela atual é fechada;

```
private void avisoForm_Load(object sender, EventArgs e)
    {
       getVars();
       message();
    }
```

- No evento avisoForm_Load, os valores das variáveis necessárias são obtidos utilizando a função getVars(). Em seguida, a função message() é chamada para exibir uma mensagem com base nos valores das variáveis obtidas. Este evento ocorre quando o formulário de aviso é carregado pela primeira vez, garantindo que a mensagem apropriada seja exibida imediatamente;

```
private void showRfidForm()
    {
       rfidReader reader = new rfidReader();
       reader.Show();
    }
```

- A função showRfidForm() cria uma nova instância do formulário rfidReader e, em seguida, o exibe ao usuário. Essa função é chamada quando o usuário clica em um botão ou executa alguma ação que requer a exibição do formulário de leitura RFID;

```
private void message()
    {
       aviso.Text = "Os valores são:" +
          "\n\nID ordem produção: " + idProdOrdem +
          "\nID serie: " + idSerie +
          "\nArtigo: " +  artigo +
          "\nQuantidade: " + quantidade +
          "\nQuantidade Min: " + quantidadeMin +
          "\n\nAo confirmar os valores serão inalteraveis";
    }
```

- A função message() é responsável por definir o texto exibido em um controle de texto chamado aviso. Esse texto inclui os valores das variáveis idProdOrdem, idSerie, artigo, quantidade e quantidadeMin, que são atribuídos anteriormente. O texto informa ao usuário sobre os valores atuais dessas variáveis e destaca que eles serão inalteráveis ao confirmar;

```
private void avisoForm_FormClosed(object sender, FormClosingEventArgs e)
    {
        db.modelDelete(idProdOrdem, idSerie);

        if (initialPage.instance == null)
        {
            initialPage form = new initialPage();
            form.Show();
        }
        else
        {
            initialPage.instance.Show();
        }
    }
```

- Esta função é chamada quando o formulário de aviso está prestes a ser fechado. Ela executa algumas ações antes do fechamento do formulário. Primeiro, deleta um modelo com base nos valores das variáveis idProdOrdem e idSerie na base de dados. Em seguida, verifica se há uma instância da página inicial (initialPage). Se não houver, cria uma nova instância e a exibe. Caso contrário, mostra a instância existente;