



Relatório do Trabalho Prático

Integração de Sistemas de Informação

Alunos:

18836 - Carlos Martins

15255 – Pedro Macedo

Professores: Luís Ferreira

Licenciatura em Engenharia de Sistemas Informáticos

Braga, Novembro, 2021

Índice de figuras

Figura 1 - Interface DBRest.....	6
Figura 2 - Classe Produto.....	7
Figura 3 - Classe encomenda.....	9
Figura 4 - Restful Services w/ Swagger.....	10
Figura 5 - Formulario criar produto	11
Figura 6 - Listar Produtos	12
Figura 7 – DeleteProduct.....	13
Figura 8 – MakeOrder	13
Figura 9 – AddProductToOrder	13
Figura 10 - GetEncomendas(false).....	14
Figura 11 - Resultado GetAllPessoas().....	15
Figura 12 – RegistInfected.....	16
Figura 13 - Contactos de Risco	16
Figura 14 – RegistIsolated.....	16
Figura 15 - GetMostOrderedProducts.....	19
Figura 16 – GetTestados	20
Figura 17 - Serviço externo COVID VOST.....	21

Índice

Conteúdo

Índice de figuras	2
Conteúdo.....	3
1. Introdução	4
1.1. Contextualização.....	4
1.2. Motivação e objetivos.....	4
1.3. Estrutura do Documento.....	4
1.4. Documentos de entrega	4
2. Implementação.....	5
2.1. Descrição do problema.....	5
2.2. Base de dados	5
2.3. Ponto 1 – API Restfull para gestão de operações CRUD	6
FindProduct.....	7
CreateProduct	7
ListProducts	8
DeleteProduct	8
MakeOrder e AddProductToOrder.....	9
Implementação utilizando Web API w/ Swagger.....	10
2.4. Ponto2 – Implementação das funções CRUD num cliente	11
CreateProduct	11
ListProducts	12
DeleteProduct	12
MakeOrder e AddProductToOrder.....	13
GetEncomendas.....	14
2.5. Ponto 3 – Implementação dos serviços SOAP	15
FindNif.....	15
GetAllPessoas.....	15
RegistInfected e RegistIsolated	16
2.6. Ponto 4 – Formulários de visita JSON/XML.....	17
RelatorioPSP.....	17
RelatorioGNR.....	18
2.7. Ponto 5 – Dashboard de estatísticas.....	18
GetMostOrderedProducts.....	19
GetTestados e GetInfetados.....	19
2.8. Ponto 6 – Implementação do OAuth.....	20
2.9. Ponto 7 - Dashboard com o serviço externo COVID-VOST	20
3. Publicação dos serviços no Microsoft Azure	21
4. Conclusão	22

Bibliografia	23
--------------------	----

1. Introdução

1.1. Contextualização

Este relatório foi feito devido a necessidade de mostrar toda a documentação que foi feita durante a realização do trabalho que foi proposto.

1.2. Motivação e objetivos

Desenvolver este trabalho proposto foi bastante produtivo, porque deu para perceber se estava, de facto, a entender a matéria dada na aula e se a consigo aplicar na totalidade. Surgiram várias dificuldades durante a realização do trabalho, mas com investigação e mesmo a ajuda dos colegas consegui ultrapassar as mesmas.

Os meus objetivos após a realização deste trabalho passam por cimentar os conhecimentos obtidos nesta cadeira noutras cadeiras, visto que na informática temos mil e uma maneiras de resolver um problema.

1.3. Estrutura do Documento

O documento encontra-se organizado em três capítulos:

- O capítulo introdutório, onde se faz uma abordagem ao contexto do problema, motivação e objetivos;
- O capítulo de implementação, onde é descrita toda a implementação do código e a sua devida explicação;
- O capítulo de conclusão, onde são retiradas as conclusões desta fase de desenvolvimento da aplicação;

1.4. Documentos de entrega

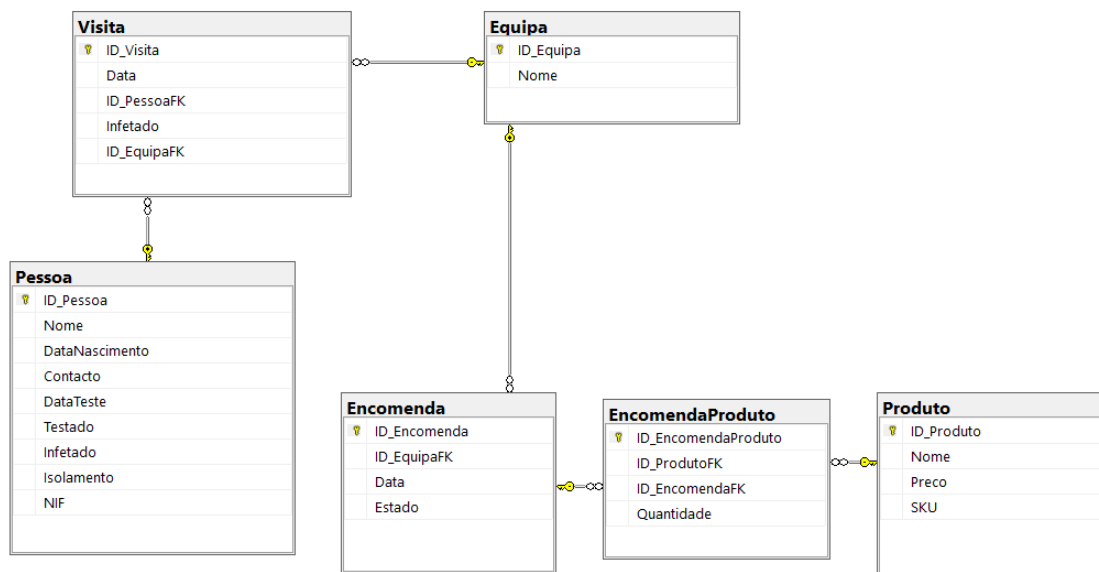
- Na pasta Cliente encontra-se o cliente referente ao programa.
- Na pasta Servidor encontram-se os serviços REST e SOAP desenvolvidos utilizando WCF Services.
- Na pasta FicheirosParaCarregar encontram-se os ficheiros XML e JSON referentes ao ponto 4 do trabalho.
- Na pasta OAuthRestApi encontram-se serviços CRUD REST que o professor pediu para desenvolver em WebAPI, de modo a conseguir testar o OAuth.

2. Implementação

2.1.Descrição do problema

Ao longo da leitura do enunciado, vamos obtendo informações sobre como devemos fazer o programa em si. Resumidamente, o enunciado pede-nos um sistema multifuncional para gestão de operações ligadas ao COVID, das quais, gestão de infetados/isolados, gestão de equipamento para as diversas equipas que visitam lares, diversas estatísticas que permitam verificar como se encontra a situação pandémica e, por fim, apresentar os números atuais da pandemia.

2.2. Base de dados



Para a realização deste trabalho optámos por desenvolver uma base de dados. É uma base de dados relativamente simples, com muitas oportunidade de melhoramento, mas que para o problema em questão serviu perfeitamente.

Temos duas áreas principais neste diagrama: a parte das Visitas em que identifica a pessoa que foi visitada e se estava infetada (se estiver infetada a mesma vai ter os seus estados “Testado”, “Infetado” e “Isolamento” alterados para TRUE), além disso, também guarda informações acerca da equipa que foi responsável por essa visita e em que dia é que ocorreu a mesma. De seguida temos a parte das encomendas (são as requisições faladas no enunciado) e optámos por três tabelas: a Produto, que tem os respetivos

produtos com as suas informações, a Encomenda que guarda o número de encomenda, a equipa que fez a encomenda e outros campos e por fim temos a EncomendaProduto que serve como uma “ponte” entre as duas, porque visto que uma encomenda pode ter vários produtos esta tabela guarda o ID_Encomenda “oficial” e os respetivos produtos.

Por fim, para realizar a conexão da base de dados com o programa bastou abrir o Server Explorer no Visual Studio, preencher com as respetivas informações e por último adicionar a connection string ao ficheiro Web.config.

2.3. Ponto 1 – API Restfull para gestão de operações CRUD

Neste ponto foi pedido que desenvolvessemos uma API Restfull que permitisse realizar operações CRUD necessárias à manutenção de produtos e requisições por parte das equipas. Para isso começamos por utilizar WCF Services e também desenvolvemos em WebAPI com Swagger para demonstrar a utilização do OAuth.

Como estamos a trabalhar com REST criamos a interface DBRest e definimo-la como **ServiceContract** chamado IDBRESt e nele estão definidos todos os métodos REST utilizados para a resolução deste enunciado. Ao definir um método na interface temos que definir o seu contrato, o seu formato de reposta e o seu URI. Vou começar por apresentar o diagrama da interface e de seguida explico cada uma das suas funções.

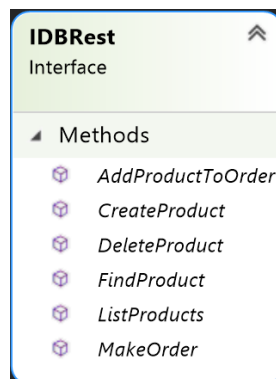


Figura 1 - Interface DBRest

FindProduct

```
[OperationContract]
[WebGet(ResponseFormat = WebMessageFormat.Json, UriTemplate = "FindProduct/{nome}")]
bool FindProduct(string nome);
```

O objetivo desta função é verificar se um certo produto já está presente na BD de forma a prevenir duplicados. Na primeira linha do código em cima, começamos por definir que a função é um `OperationContract` e isso indica que faz parte de um `ServiceContract`, neste caso faz parte do `IDBRest`. De seguida, na segunda linha, definimos é **um método GET** com o formato de **resposta JSON** e o seu **URI: FindProduct/<nome >** em que este nome é recebido por argumento.

Esta função começa por receber a connection string para estabelecer ligação à BD e recebe a seguinte query SQL `"SELECT * FROM Produto WHERE Nome = @nome"`. Como é possível ver, na clausula WHERE a query verifica se o nome já está presnte e se estiver seleciona-o. Por fim, para verificar se o produto existe ou não, utilizamos a função **SqlDataReader** para ler a tabela gerada pela query e depois verificamos se a mesma tem linhas ou não, se tiver o produto já existe, se não é porque não existe.

CreateProduct

```
[OperationContract]
[WebInvoke(Method="POST", ResponseFormat = WebMessageFormat.Json, UriTemplate =
"CreateProduct")]
bool CreateProduct(Produto p);
```

Como o nome diz, o objetivo desta função é criar um produto, ou seja, tudo indica que esta função utilize **um método POST**, utilizamos o **formato de resposta JSON** mais uma vez e o **URI** definido foi **CreateProduct**.

Para esta função tivemos que criar uma classe produto, porque o POST não permitia que passássemos mais que um argumento. Apresento a classe produto no diagrama abaixo:

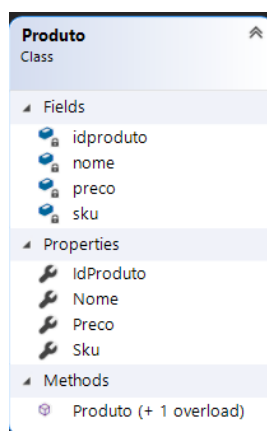


Figura 2 - Classe Produto

Para a sua implementação começamos por criar um **DataSet** que vai receber o Produto caso seja criado, de seguida fizemos a ligação à BD com a connection string e utilizamos a seguinte query: `"INSERT INTO Produto (Nome, Preco, Sku) VALUES(@Nome, @Preco, @Sku)"` depois executamo-la com o **SqlDataAdapter**(query, connectionstring), fizemos a respetiva parametrização dos values enviados na query (Esta parameterização é feita para prevenir ataques informáticos), e para finalizar utilizamos uma condição que recebe por argumento a função FindProduct e se o resultado da mesma for True, é porque o produto já existe então o programa retorna false, se não o produto é inserido na tabela Produto.

ListProducts

```
[OperationContract]
[WebGet(ResponseFormat = WebMessageFormat.Xml, UriTemplate = "ListProducts")]
List<Produto> ListProducts();
```

Esta função lista todos os produtos na base de dados, é um **método GET**, com o **formato resposta XML** e o seu **URI** é **ListProducts**. Começamos por definir um **ArrayList** para armazenar os produtos, conexão à base de dados e a query que seleciona todos os produtos presentes na mesma: `"SELECT Id_Produto, Sku, Nome, Preco FROM Produto"` depois utilizamos o método **SqlDataReader** para interpretar a tabela e até o SqlDataReader deixar de encontrar conteúdo na tabela vai ser inserido um novo Produto, com os respetivos campos na ArrayList, por fim converte o ArrayList numa lista de produtos e retorna a mesma.

DeleteProduct

```
[OperationContract]
[WebInvoke(Method="DELETE", ResponseFormat = WebMessageFormat.Xml, UriTemplate =
"DeleteProduct/{sku}")]
bool DeleteProduct(string sku);
```

Esta é a única função com um **método DELETE** presente neste trabalho, envia uma **resposta** com o **formato XML** e o seu URI é **DeleteProduct/<sku>** em que o SKU é recebido por argumento. O objetivo dela é receber um SKU e apagar o produto com o respetivo SKU.

Praticamente todas as funções REST deste trabalho seguem os mesmos passos e esta não é exceção. É feita a conexão com a base de dados, depois é criamos um comando com a query: `"DELETE FROM Produto WHERE Sku = @Sku"`, fazemos a parameterização do SKU depois se o comando retornar 1 a operação foi bem sucedida e o produto foi apagado e a função retorna true, se não retorna false.

MakeOrder e AddProductToOrder

```
[OperationContract]
[WebInvoke(Method = "POST", ResponseFormat = WebMessageFormat.Json, UriTemplate =
"MakeOrder")]
    bool MakeOrder(Encomenda e);

[OperationContract]
[WebInvoke(Method = "POST", ResponseFormat = WebMessageFormat.Json, UriTemplate =
"AddProductToOrder")]
    bool AddProductToOrder(Encomenda e);
```

Começo por dizer que esta esta foi a funcionalidade da aplicação mais complicada em termos de processo de desenvolvimento e talvez a maneira com que foi realizada não seja a melhor.

Como é possível ver ambas são utilizam o **método POST**, **retornam respostas JSON** e os seus **URIs** são **MakeOrder** e **AddProductToOrder**, respetivamente.

Escolhi apresentar estas duas funções em conjunto porque não fazem sentido ser separadas, visto que uma é encarregue de criar o Id da encomenda e a outra é encarregue de adicionar os produtos a esse mesmo Id. Ambas recebem a classe encomenda que apresento abaixo:

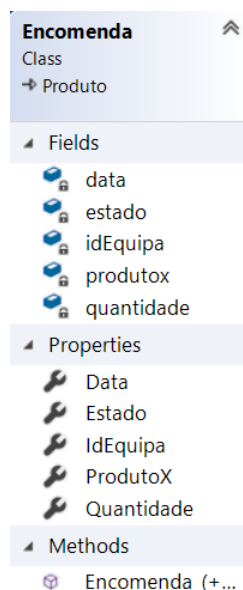


Figura 3 - Classe encomenda

Vou começar por explicar o funcionamento da função **MakeOrder**. Comecei por criar um DataSet para receber o resultado da query: **"INSERT INTO Encomenda(ID_EquipaFK, Data, Estado)VALUES(@Id_EquipaFK, @Data, 0) "**, também foi feita a ligação com a base de dados e a respetiva parametrização dos comandos Id_Equipa e Data. Com isto é adicionada uma nova linha à base de dados com o ID da encomenda, o ID da equipa que a fez, a data da encomenda e o estado da encomenda.

Passando agora para a **AddProductToOrder**, esta função tem exatamente o

mesmo processo que a anterior: criar um DataSet, estabelece ligação à BD, e executa a seguinte query “INSERT INTO EncomendaProduto(ID_ProdutoFK, ID_EncomendaFK, Quantidade) VALUES(@ID_Produto, (SELECT TOP 1 ID_Encomenda FROM Encomenda ORDER BY ID_Encomenda DESC), @Quantidade) “. Como é possível ver o resultado da query vai ser a última encomenda feita, ou seja, a encomenda criada em MakeOrder. Para garantir que o cliente insere nesta encomenda, no cliente, não permitimos que o cliente mexer noutras janelas que não a janela de inserir os produtos à encomenda.

Implementação utilizando Web API w/ Swagger

Como praticamente todos os serviços, presentes na imagem já estavam desenvolvidos, a implementação dos mesmos neste serviço foi simples.

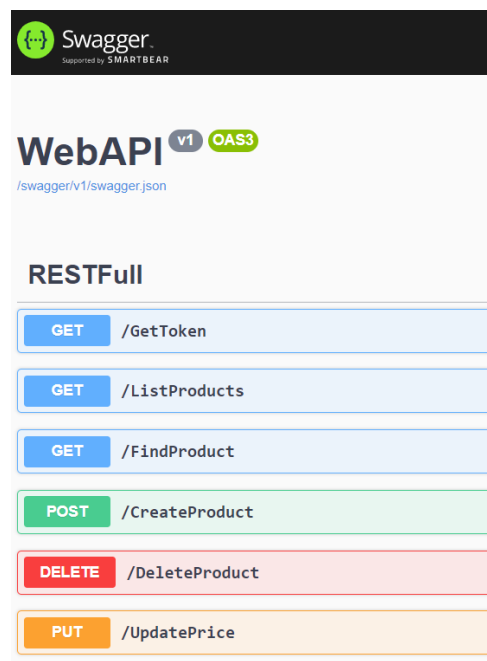


Figura 4 - Restful Services w/ Swagger

Foram realizadas apenas alterações mínimas como, por exemplo, ao nível da conexão com a base de dados que passo agora a explicar:

Nos WCF Services existe um ficheiro “**web.config**” em que colocamos a nossa connection string da seguinte maneira:

```
<connectionStrings>
  <add name="tpISIConnectionString"
connectionString="Server=(local)\SQLEXPRESS;Initial Catalog=TP_ISI;Integrated
Security=True" />
</connectionStrings>
```

Já na implementação numa API Rest tentamos meter a connection string no ficheiro “**appsettings.json**” da seguinte maneira:

```
"ConnectionStrings": {  
    "tpISISConnectionString": "(local)\\SQLEXPRESS; Database = TP_ISI; User Id =  
sa; Password = ola123"  
},
```

Esta implementação da connectionstring não funcionou, então para todos os serviços criados na Web API tivemos que chamá-la manualmente, ou seja:

```
// Ligação à BD  
string cs = "Data Source=localhost\\SQLEXPRESS;Initial Catalog=TP_ISI;Integrated  
Security=True";  
//2º OpenConnection  
SqlConnection con = new SqlConnection(cs);
```

Para além disso, foi criada uma função para verificar o método PUT em ação. A função chama-se **UpdatePrice** e o seu objeto é alterar o preço de um produto, dado o seu SKU

UpdatePrice

Apesar de não ser o mais correto, por se tratar de um método PUT e não de um método POST, esta função recebe dois argumentos: o SKU e o Preço. Quanto à sua implementação, inicialmente é criada uma conexão string e um comando do tipo **SqlDataAdapter** com a função de Update que recebe a seguinte query: "UPDATE Produto SET PRECO = @Preco WHERE SKU = @SKU ", de seguida é feita a parameterização dos seus parâmetros (SKU e Preço) e o resultado da mesma é enviado para um DataSet.

2.4. Ponto2 – Implementação das funções CRUD num cliente

CreateProduct

Para aceder a esta funcionalidade temos que seguir o seguinte caminho: Form Inicial -> Encomendas/Produtos -> Criar Produto. Seguindo este caminho vai aparecer uma certa janela com o seguinte formulário:

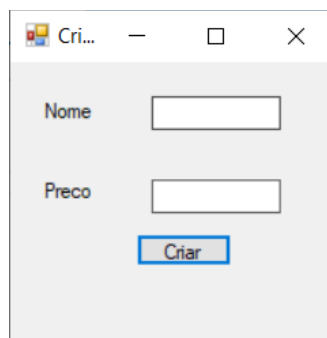
A image shows a small application window titled 'Cri...' with standard Windows window controls (minimize, maximize, close). Inside the window is a form with two text input fields. The first field is labeled 'Nome' and the second is labeled 'Preço'. Below these fields is a button labeled 'Criar'.

Figura 5 - Formulário criar produto

Na implementação deste produto criamos um objeto da classe Produto em que o

nome e o preço são recebidos por input do utilizador nas caixas de texto apresentadas em cima, já o SKU é gerado aleatoriamente por uma função chamada **SkuGenerator**, que foi baseada numa função apresenta nas referências. De seguida, é feita a serialização do produto em formato JSON e a mesma é guardada num **MemoryStream**. Para finalizar, criamos um WebClient que é um tipo de dados utilizado para enviar e receber dados e usamos o método **UploadString** com o URI da função, o método que neste caso é POST e o conteúdo que é o que o produto preencheu nas caixas de texto.

ListProducts

Para aceder a esta funcionalidade temos que seguir o seguinte caminho: Form Inicial -> Encomendas/Produtos -> Listar Produtos. O resultado é o seguinte:



	IdProduto	Nome	Preco	Sku
▶	1015	Desinfetante ...	1.49	SKU9438
	1016	Luvas ...	2.1	SKU7302
	1017	Agua ...	0.1	SKU6096
	1018	Mascaras ...	1.49	SKU2992
	1019	Batas ...	5	SKU4914
	1020	Aspirador Portatil ...	15	SKU6201
	1021	Toalhitas anti-co...	1	SKU8641
*				

Figura 6 - Listar Produtos

Como todas as implementações de serviços REST em clientes, esta também cria um **HttpRequest** e uma string com o URI do método em questão.

O **ListProducts** é uma função com o método GET, por isso para verificarmos se o GET falhou criei uma variável do tipo **HttpResponse** em que o seu conteúdo é igual à resposta ao **HttpRequest** inicial, depois verificamos se o código de resposta recebido não é o típico OK (200), se não for o programa envia uma exceção a avisar que o GET falhou, se tudo correr bem é criado um **DataSet** para receber o conteúdo recebido (XML) pelo GET num **DataSet** e utilizamos um **dataGrid** para mostrar o conteúdo recebido.

DeleteProduct

Para aceder a esta funcionalidade temos que seguir o seguinte caminho: Form Inicial -> Encomendas/Produtos. O formulário é o seguinte:



Figura 7 – DeleteProduct

O processo é o mesmo que o anterior, simplesmente definimos uma variável do tipo **HttpWebResponse**, guardamos o URI numa string, verificamos se o pedido foi bem sucedido, isto verificando se o código retornado foi o OK(200). Além disso, temos que definir manualmente que o método a utilizar é um DELETE o que é feito com a seguinte linha: `request.Method = "DELETE"`; Sem isto, o C# assume que o método em utilização é um GET e retorna 405 – Proibido.

MakeOrder e AddProductToOrder

Para aceder a esta funcionalidade temos que seguir o seguinte caminho: Form Inicial -> Encomendas/Produtos -> Inserir o ID da equipa e clicar em nova encomenda. O resultado é o seguinte:



Figura 8 – MakeOrder

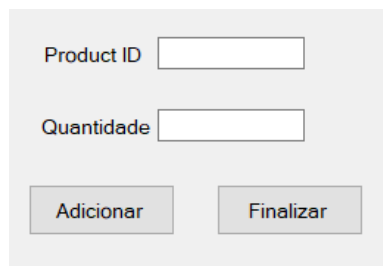


Figura 9 – AddProductToOrder

Ambos são métodos **POST** e a sua implementação é semelhante à implementação da função de criar um produto.

Se um ID de equipa válido for inserido na caixa de texto, ao pressionar o botão de nova Encomenda vai ser criado um novo objeto Encomenda, em que a data de encomenda vai ser obtida por `DateTime.Now`. De seguida, é feita a serialização do produto em formato JSON e a mesma é guardada num **MemoryStream**. Para finalizar, criamos um **WebClient** que é um tipo de dados utilizado para enviar e receber dados e usamos o método **UploadString** com o URI da função, o método que neste caso é POST e o conteúdo que é o que o produto preencheu nas caixas de texto. Com isto temos a função **MakeOrder** executada.

A função **AddProductToOrder** funciona exatamente da mesma maneira que a

MakeOrder com a exceção de que os parametros recebidos nas caixas de texto serem diferentes, neste caso recebe IdProduto e a Quantidade e quando é pressionado o botão adicionar o produto é adicionado à encomenda, quando é pressionado o botão finalizar a encomenda é enviada.

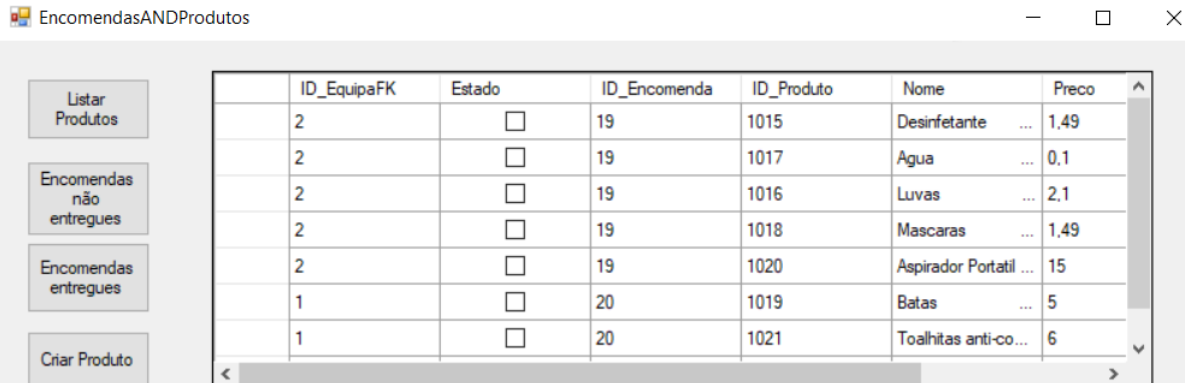
GetEncomendas

Para além da implementação de um cliente, também era pedida uma função que mostrasse as encomendas/requisições entregues e pendentes, para tal foi desenvolvida a função **GetEncomendas**. É uma **função SOAP** que **retorna um DataSet** e possui um **argumento** que é o **estado**. Este estado é um **booleano** e caracteriza o estado de uma encomenda, ou seja, se foi entregue (True) ou não (False). É relativamente simples de entender, simplesmente realiza uma conexão à BD e executa a seguinte query:

```
SELECT Encomenda.ID_EquipaFK, Encomenda.Estado, Encomenda.ID_Encomenda,
Produto.ID_Produto, Produto.Nome, Produto.Preco, Encomenda.Data
FROM Encomenda INNER JOIN
EncomendaProduto ON Encomenda.ID_Encomenda =
EncomendaProduto.ID_EncomendaFK INNER JOIN
Produto ON EncomendaProduto.ID_ProdutoFK = Produto.ID_Produto
WHERE Encomenda.Estado = @Estado"
```

Esta query seleciona algumas colunas das tabelas Encomenda, EncomendaProdutos e Produto e utiliza InnerJoins para obter os diversos parametros. Estes Inner Joins comparam as chaves estrangeiras com a chave primária e retornam os dados que possuem o estado enviado por argumento, ou seja, se o estado for true retornam as entregues.

Por fim deixo a sua implementação no cliente:



ID_EquipaFK	Estado	ID_Encomenda	ID_Produto	Nome	Preco
2	<input type="checkbox"/>	19	1015	Desinfetante ...	1,49
2	<input type="checkbox"/>	19	1017	Agua ...	0,1
2	<input type="checkbox"/>	19	1016	Luvas ...	2,1
2	<input type="checkbox"/>	19	1018	Mascaras ...	1,49
2	<input type="checkbox"/>	19	1020	Aspirador Portatil ...	15
1	<input type="checkbox"/>	20	1019	Batas ...	5
1	<input type="checkbox"/>	20	1021	Toalhitas anti-co...	6

Figura 10 - GetEncomendas(false)

2.5. Ponto 3 – Implementação dos serviços SOAP

Neste ponto era pedido para desenvolvermos serviços SOAP que basicamente conseguissem fazer uma gestão de infetados/isolados e para testar os mesmos foi também pedido o desenvolvimento de um cliente.

FindNif

Começamos por implementar esta função que atua como uma auxiliar a alguns serviços SOAP presentes neste trabalho. Como todas as outras funções neste trabalho que precisam de acesso à base de dados, esta também utiliza uma connection string. Depois de efetuada a ligação é executada uma query que procura todas as pessoas com o Nif recebido por argumento: `SELECT * FROM Pessoa WHERE Nif = @nif`. Depois, o resultado desta query é guardado numa variável do tipo **SqlDataReader**, por fim executamos o método associado chamado **ExecuteReader** e verificamos se o resultado retornou linhas, se sim é porque o Nif existe na base de dados.

GetAllPessoas

Como o nome diz, esta função é responsável por retornar todas as pessoas presentes na base de dados. É relativamente simples e o seu funcionamento passa por fazer uma query `SELECT * FROM Pessoa` e enviar o resultado para um DataSet.

No cliente a função é chamada através de um botão e o resultado é o seguinte:

ID_Pessoa	Nome	DataNascimento	Contacto
1	Manel	01/01/1940	910000001
2	Manuela	02/02/1943	910000002
3	Tone	03/03/1950	910000003
4	Joaquim	23/04/1955	910000004
5	Manuelinha	05/06/1945	910000005
6	Patricio	07/02/1951	910000006
7	Patricia	08/08/1959	910000007
8	Ana	07/07/1960	910000008

Figura 11 - Resultado GetAllPessoas()

RegistInfected e RegistIsolated

Estas duas funções são utilizadas em conjunto e seguem o mesmo processo mas com queries diferentes.

No esquema de base de dados apresentado em cima é possível verificar que na tabela Pessoa existem 3 campos: **Testado**, **Infetado** e **Isolado**. A **RegistInfected** altera todos esses campos para **True** (`UPDATE Pessoa SET Testado = 1, Infetado = 1, Isolamento = 1 WHERE Nif = @nif`) e a **RegistIsolated** altera apenas o campo **Isolado** (`UPDATE Pessoa SET Testado = 0, Infetado = 0, Isolamento = 1 WHERE Nif = @nif`).

Ambas fazem conexão com a base de dados e ambas possuem um UpdateCommand com queries de Update (cada uma com os diferentes parametros a atualizar), é feita a parameterização dos parametros passados nas diferentes queries e de seguida é verificado se o Nif introduzido existe na base de dados, se o Nif existir então os campos são alterados.

Como é que funciona isto no cliente? Quando um infetado é registado com sucesso aparece uma textbox a perguntar com quantas pessoas é que esteve em contacto, depois é pedido o contacto de cada uma dessas pessoas e a RegistIsolated é executada.

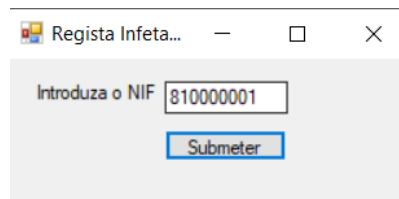


Figura 12 – RegistInfected

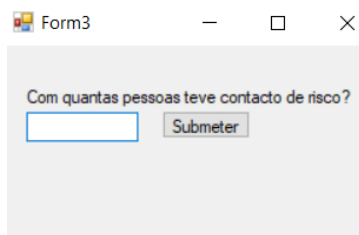


Figura 13 - Contactos de Risco

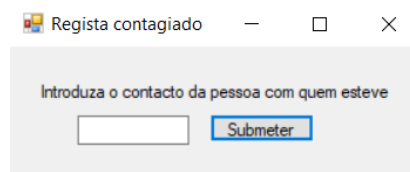


Figura 14 – RegistIsolated

2.6. Ponto 4 – Formulários de visita JSON/XML

Neste ponto era pedido que fosse criada uma função que conseguisse **importar ficheiros XML e JSON** com os dados das visitas efetuadas pela PSP e GNR respetivamente. A estrutura destes documentos era ao agrado do desenvolvedor então decidimos utilizar as seguintes estruturas:

JSON:

```
"Visita": [  
  {  
    "DataVisita": "2021-09-11T14:32",  
    "CodPessoa": 5,  
    "Infetado": 1,  
    "CodEquipa": 2  
  }  
]
```

XML:

```
<Visitas>  
  <Visita>  
    <DataVisita>2021-10-18 14:32</DataVisita>  
    <CodPessoa>1</CodPessoa>  
    <Infetado>1</Infetado>  
    <CodEquipa>1</CodEquipa>  
  </Visita>  
</Visitas>
```

Apresentadas as estruturas passo agora à apresentação das funções **RelatorioPSP** e **RelatorioGNR**.

RelatorioPSP

Como já foi dito, esta função tem como objetivo de interpretar documentos XML com a estrutura definida por nós e guardar a sua informação na base de dados dentro da tabela Visitas. Esta função recebe por parâmetro esse mesmo ficheiro.

Começamos por declarar um conjunto de variáveis que equivalem às tags presentes no ficheiro XML, depois criámos uma instância de um documento XML e carregamos o ficheiro recebido por argumento para essa mesma instância. De seguida, percorremos cada um dos nodes fazendo as respetivas atribuições e conversões, por exemplo, a dataVisita é o primeiro elemento da estrutura então para guardar o valor utilizamos:

```
dataVisita = Convert.ToDateTime(xmlNode.ChildNodes[0].InnerText);
```

Feitas todas atribuições para cada um dos parâmetros, temos, agora, que criar um

DataSet para receber todos estes dados. De seguida, seguimos o processo habitual de fazer conexão à base de dados com a respetiva connection string, usar a query “**INSERT INTO Visita (Data, ID_PessoaFK, Infetado, ID_EquipaFK) VALUES(@dataVisita, @idPessoa, @infetado, @idEquipa)**” e parametrizá-la. Além destes passos habituais, aproveitamos os dados recebidos no ficheiro para atualizar a condição da pessoa (Testado, Infetado, Isolado), por exemplo, se a pessoa tiver testado positivo então os campos são atualizados (o mesmo acontece caso tenha testado negativo).

RelatorioGNR

Tal como em muitas das funções criadas para este trabalho, esta foi baseada num dos exemplos fornecidos pelo professor Luís Ferreira, mais concretamente no exemplo de Serialização JSON Aula 9.

Semelhantemente ao anterior, este também recebe o ficheiro por argumento e também começamos por declarar variáveis que aos campos do ficheiro JSON. Depois obtemos a raiz do ficheiro JSON com a propriedade **RootElement**, com isto conseguimos obter a tag Visita e percorrer os seus nodes utilizando um foreach. Dentro do ciclo utilizamos a função `GetProperty` para obter os campos e guardá-los nas nossas variáveis efetuando as respetivas conversões, por exemplo:

```
JsonElement jsonData = temp.GetProperty("DataVisita");  
dataVisita = jsonData.GetDateTime();
```

Depois de obtidas todas as variáveis, seguimos o processo habitual de estabelecer ligação à base de dados, criar uma string com a respetiva query (**INSERT INTO Visita (ID_PessoaFK, Data, Infetado, ID_EquipaFK) VALUES(@idPessoa, @dataVisita, @infetado, @idEquipa)**), fazer a parametrização das variáveis da query e mais uma vez verificar o resultado do teste para atualizar os respetivos parâmetros.

2.7. Ponto 5 – Dashboard de estatísticas

Começo este ponto por dizer que este foi um dos pontos menos conseguido, existiram algumas dificuldades a nível de SQL e a base de dados tinha algumas falhas que já não tivemos tempo de melhorar a tempo sem comprometer outras funcionalidades, mas creio que se tivéssemos as queries/base de dados ideal o processo era simples de se implementar visto que era apenas repetir o que fizemos na função que realizamos para este ponto.

As dificuldades começaram logo na segunda estatística pedida que pedia para apresentar as equipas que mais gastaram. Nós efetivamente, conseguimos apresentar algo parecido com isto, mas por alguma razão numa encomenda nem todos os produtos entravam para o total, por exemplo, na encomenda 19 existiam dois totais de 80 e 40, em

vez de existir um total de 120. Na terceira query simplesmente não conseguimos obter valores em condições, estávamos a usar a função DateAdd do SQL e ou retornava todas as visitas efetuadas desde sempre, ou não retornava nenhuma.

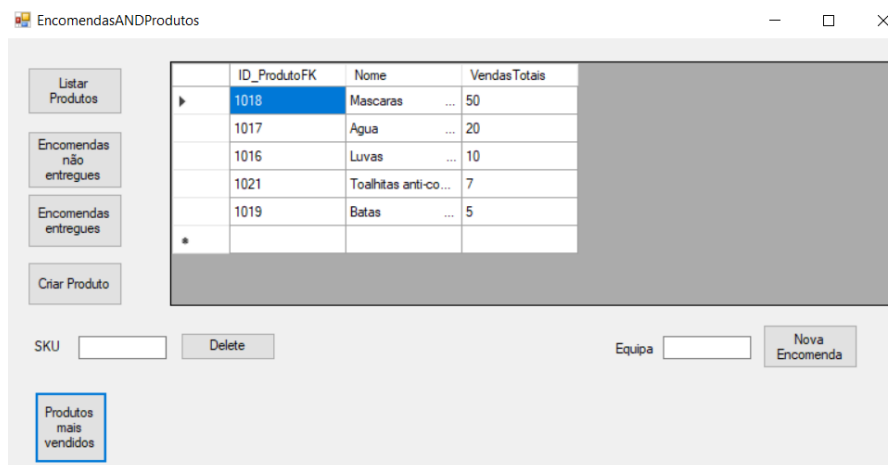
Passo agora a apresentar a função **GetMostOrderedProducts**.

GetMostOrderedProducts

É uma função SOAP com que retorna um DataSet e como todas as funções neste relatório também utiliza uma connection string para se ligar à base de dados. A query utilizada foi a seguinte:

```
SELECT TOP(5) EncomendaProduto.ID_ProdutoFK,
Produto.Nome,SUM(EncomendaProduto.Quantidade) As VendasTotais
FROM      EncomendaProduto INNER JOIN Produto ON EncomendaProduto.ID_ProdutoFK =
Produto.ID_Produto
GROUP BY ID_ProdutoFK, Produto.Nome
ORDER BY VendasTotais DESC"
```

Como é possível ver, vários parâmetros são seleccionados e é feita um somatório da quantidade de vezes em que um produto aparece em encomendas. Além disso é feito um INNER JOIN que compara Ids de produto de diferentes tabelas, para obter o Id do produto em questão. Por fim é retornado o DataSet com as informações obtidas.



ID_ProdutoFK	Nome	VendasTotais
1018	Mascaras	50
1017	Agua	20
1016	Luvas	10
1021	Toalhas anti-co...	7
1019	Batas	5

Figura 15 - GetMostOrderedProducts

GetTestados e GetInfetados

São funções que achamos relevantes estarem presentes na dashboard, apesar de serem relativamente simples e as suas implementações serem praticamente iguais com a diferença no campo utilizado no Count e no Where da query, numa é dado Count ao campo Testado quando este mesmo campo está preenchido com True e na outra é dado Count ao campo Infetado quando este mesmo campo está, também, a True.

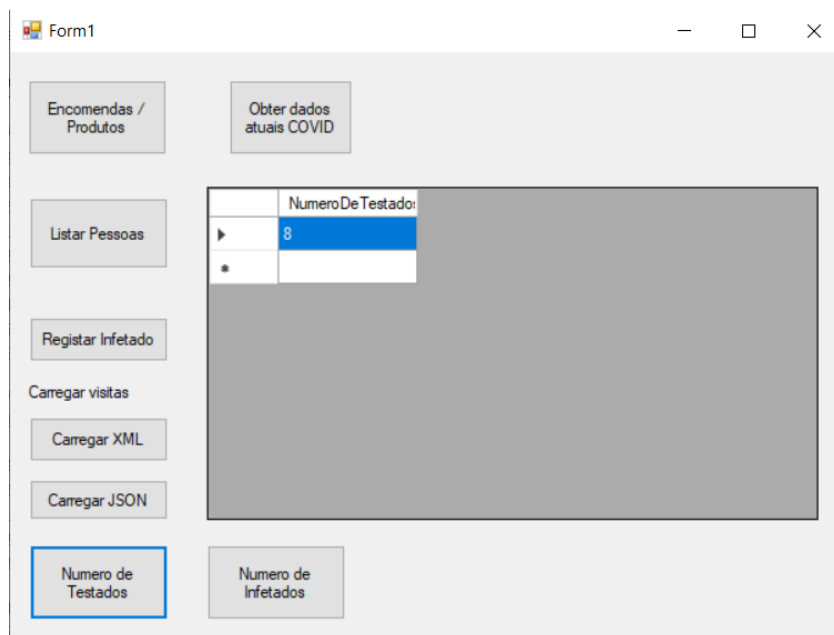


Figura 16 – GetTestados

2.8. Ponto 6 – Implementação do OAuth

Quanto a este ponto, foi um ponto que não conseguimos executar. Tentamos seguir o exemplo que o professor publicou no Moodle, mas mesmo assim não conseguimos. Creio que precisássemos de perder mais um ponto nesta tecnologia uma vez que não tivemos tanto tempo para a abordar. É uma tecnologia que certamente irei investigar mais no futuro e espero conseguir implementá-la no próximo trabalho prático se tiver possibilidade para tal.

Para esta tentativa de implementação começamos por importar as classes **AuthResponse**, **AuthRequest**, além destas classes também importamos os controladores **AuthController**, **JWTAuthManager** e **SecurityController**.

Tivemos também outra abordagem que foi tentada com uns colegas em que a opção **Authorize** aparecia assim como a textbox que recebe o **Id** e o **token**, mas a validação não era feita corretamente, porque qualquer string era autorizada. Além disso, nesta implementação, todos os serviços com a tag **[Authorize]** davam erro.

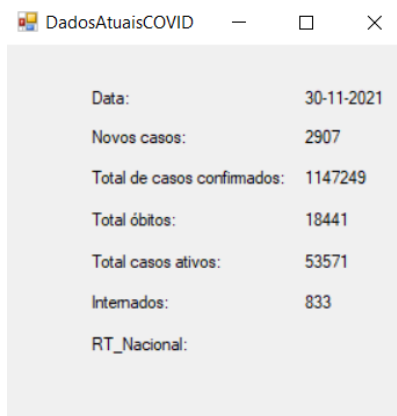
2.9. Ponto 7 - Dashboard com o serviço externo COVID-VOST

Neste ponto utilizamos serviços externos. Basicamente, a implementação deste serviço no cliente foi a mesma do que os serviços REST desenvolvidos para este trabalho.

Começamos por analisar os serviços disponibilizados por eles e chegámos à conclusão de que o mais pertinente para este trabalho era o **get_last_update**, então analisamos os campos retornados pelo serviço e criamos uma classe com todos os elementos retornados.

Tal como os outros serviços REST, o **URI** é guardado numa string e é criado um

WebRequest, depois verificamos a resposta desse WebRequest e se ele deu um código diferente de **OK (200)** é enviada uma exceção. De seguida, serializamos o JSON recebido numa classe Root e apresentamos os valores numa label.



A screenshot of a Windows application window titled "DadosAtuaisCOVID". The window has a standard title bar with a minimize button, a maximize button, and a close button. The content area of the window displays a list of COVID-related statistics in a two-column format. The labels are on the left and the corresponding values are on the right.

Data:	30-11-2021
Novos casos:	2907
Total de casos confirmados:	1147249
Total óbitos:	18441
Total casos ativos:	53571
Internados:	833
RT_Nacional:	

Figura 17 - Serviço externo COVID VOST

3. Publicação dos serviços no Microsoft Azure

Para finalizar este trabalho prático também publicamos os serviços REST desenvolvidos na WebAPI na Azure. Estes serviços encontram-se disponíveis em: <https://restapitpisi2021.azurewebsites.net>

4. Conclusão

A realização deste Trabalho prático permitiu-nos perceber melhor como funciona o mundo do desenvolvimento de serviços REST e SOAP, e o que implica publicar e aceder a serviços e bases de dados.

Com este trabalho passamos varias dificuldades na implementação dos serviços e também do cliente, também nos fez perceber que temos falhas noutras áreas de programação-

Este trabalho foi desenvolvido enquanto estavamos a aprender matéria nova para aplicar no mesmo, com isso os conceitos foram melhor assimilados e conseguimos ultrapassar algumas das dificuldades que foram aparecendo.

Para finalizar, podemos concluir que o trabalho foi uma mais valia porque conseguimos desenvolver as nossas capacidades ao nível da programação de várias tecnologias mas também melhoramos psicologicamente porque começamos a olhar para os problemas de outra forma.

Bibliografia

Powerpoints e exemplos de código disponibilizados pelo professor Luís Ferreira.

Sebenta da disciplina.

<https://www.scottbrady91.com/identity-server/aspnet-core-swagger-ui-authorization-using-identityserver4>

<https://www.youtube.com/watch?v=z46lqVOv1hQ>

<https://stackoverflow.com/questions/27826491/rest-api-error-code-500-handling>

<https://stackoverflow.com/questions/1344221/how-can-i-generate-random-alphanumeric-strings>

<https://docs.microsoft.com/en-us/visualstudio/ide/class-designer/designing-and-viewing-classes-and-types?view=vs-2022>

<https://www.youtube.com/watch?v=aWePkE2ReGw>

<https://stackoverflow.com/questions/40490538/sql-multi-line-query-in-c-sharp-using-sqlcommand>

https://www.w3schools.com/sql/sql_top.asp

<https://stackoverflow.com/questions/15601143/how-to-create-a-window-that-blocks-the-parent-in-winforms>