



UNIVERSIDADE  
**LUSÓFONA**

# Solução para Pedidos Assíncronos e Monitorização em Tempo Real

## **Trabalho Final de curso**

Diogo Miguel Vieira Guiomar – nº 21302748

Orientador: Prof. Inês Oliveira

Co- Orientador: Miguel Moreno

Trabalho Final de Curso | LEI | junho 2016

[www.ulusofona.pt](http://www.ulusofona.pt)

## Contents

Índice de Figuras.....	III
Resumo .....	IV
Abstract.....	V
Agradecimentos .....	VI
Acrónimos.....	VII
1 Introdução.....	1
2 Enquadramento teórico .....	4
2.1 Azure.....	4
2.1.1 Azure Service Bus Queues.....	5
2.1.2 Azure Event Hubs .....	6
2.1.3 Azure SQL .....	8
2.1.4 Azure App Service .....	9
2.1.5 Azure WebJobs .....	9
2.2 .NET e C# .....	10
2.3 SignalR.....	10
2.4 AngularJS.....	11
2.5 MEF (Managed Extensibility Framework).....	11
2.6 JSON .....	12
2.7 Bootstrap .....	12
2.8 log4net .....	13
3 Arquitetura .....	14
4 Desenvolvimento.....	17
4.1 Orders.Signalr.Web .....	18
4.2 Orders.Api.....	20
4.3 Orders.EventHubWorker .....	21
4.4 Orders.SignalR.Api.....	22

4.5	Orders.WebJob .....	23
4.6	Orders.Common.....	24
4.7	Orders.Plugins.CustomersList .....	26
4.8	Orders.Plugins.CustomersSubscriptions.....	26
5	Resultados .....	28
6	Conclusões e Trabalho Futuro.....	30
7	Bibliografia.....	31
8	Anexos.....	33
8.1	Anexo A – Stored Procedure InsertEvents .....	33

## Índice de Figuras

Figura 1 – Serviços fornecidos pelo Microsoft Azure.....	4
Figura 2 – Ilustração básica de uma queue.....	5
Figura 3 – Arquitetura do Event Hub .....	7
Figura 4 - Propriedades da classe EventData .....	7
Figura 5 - Modelos de alojamento para SQL Server .....	8
Figura 6 – Os diferentes serviços do Azure App Service.....	9
Figura 7 - Fallback das técnicas de comunicação do SignalR.....	10
Figura 8 – Ilustração de um objecto em JSON.....	12
Figura 9 - Diagrama da solução.....	14
Figura 10 - Vista da solução no Visual Studio .....	18
Figura 11 - Routing do projeto em AngularJS .....	19
Figura 12 - Comunicação da REST Api.....	21
Figura 13 - Funcionamento da API do servidor do SignalR .....	23
Figura 14- Processamento dos pedidos provenientes da Service Bus Queue.....	28
Figura 15- Eventos lidos diretamente do Event Hub.....	28
Figura 16- Apresentação ao utilizador do estado dos pedidos .....	29
Figura 17 - Stored Procedure utilizado para Insert or Update dos eventos na base de dados .....	33

## Resumo

O objetivo deste projeto passou pelo desenvolvimento de uma solução para pedidos assíncronos com monitorização em tempo real. Solução esta, desenvolvida em tecnologias *Microsoft* e mais especificamente que fosse toda ela implementada na *cloud*, ou seja, fazendo uso dos serviços do *Azure*.

Mais concretamente, esta solução permite-nos receber pedidos a partir de um determinado *front end* coloca-los numa queue através de uma *Application Programming Interface* (API), mantendo-os numa fila de pedidos no *Azure Service Bus Queue*. Os pedidos podem ser de vários tipos, nomeadamente obter lista de clientes, obter subscrições de um determinado cliente, devido a isto, foi implementado um sistema de *plugins* para que seja invocado o *plugin* indicado para processamento do tipo de pedido em questão.

Para se garantir a monitorização em tempo real, durante todo este processo, são criados registos de eventos do decorrer de todo o processamento de cada pedido individualmente, eventos estes albergados no serviço *Azure Event Hub*. Esta monitorização é feita no *front end* utilizando o auxílio da biblioteca *SignalR*.

Durante o desenvolvimento deste trabalho final de curso, o contexto da aplicação desta solução foi alterado, para que este no futuro começasse a ser adaptado a um projeto em desenvolvimento na CreateIT.

**Palavras-chave:** *Cloud, Azure, Queue, Event Hub, API, Plugins, SignalR, Pedidos Assíncronos.*

## Abstract

The main objective of this project was the development of a solution that was capable to handle asynchronous requests with real time monitoring.

This solution should be developed using Microsoft technologies also, another key point was to implement all of it in the cloud, making use of the Azure services.

A testing front end will be making HTTP requests to a REST Api that is responsible to place those requests into a Service Bus Queue, so we can keep them in order.

We can have different types of requests, like getting a list of customers or getting customer's subscriptions. For this we implemented a plugin system, so we will match the kind of request with an available plugin, this way we will only invoke the plugin responsible to process that request.

To keep track of the Status of the requests process in real time, there are created individual events every time we got a new status for that request. Those requests are sent to Azure Event Hub service, after that the real time monitoring is done at the front end making use of the SignalR library.

While developing this solution the context of the application was changed, so in the future we could adapt it to a real project being developed by CreateIT.

**Keywords:** Cloud, Azure, Queue, Event Hub, API, Plugins, SignalR, Asynchronous Requests.

## Agradecimentos

Quero agradecer à Universidade Lusófona e aos docentes por me ajudarem a crescer na área de engenharia informática.

Agradeço à professora Inês Oliveira pela recomendação feita junto da empresa Create IT para que este trabalho final de curso fosse possível. Saliento ainda um enorme agradecimento pelo espaço e acompanhamento cedido pela Create IT e respetivos colaboradores.

Agradeço também à minha namorada Andreia Sousa pelo apoio incondicional ao longo do curso e neste caso durante o desenvolvimento do trabalho final de curso.

Por fim, agradeço também ao Sargento Ajudante para-quedista Rui Neves pelo apoio à conjugação da minha vida militar e académica.

## Acrónimos

<b>API</b>	<i>Application Programming Interface</i>
<b>CSS</b>	<i>Cascading Style Sheets</i>
<b>FIFO</b>	<i>First In, First Out</i>
<b>GUID</b>	<i>Globally Unique Identifier</i>
<b>HTML</b>	<i>HyperText Markup Language</i>
<b>HTTP(S)</b>	<i>Hypertext Transfer Protocol (Secure)</i>
<b>IaaS</b>	<i>Infrastructure as a Service</i>
<b>IDE</b>	<i>Integrated Development Environment</i>
<b>IIS</b>	<i>Internet Information Services</i>
<b>JSON</b>	<i>JavaScript Object Notation</i>
<b>MVC</b>	<i>Model View Controller</i>
<b>PaaS</b>	<i>Platform as a Service</i>
<b>REST</b>	<i>Representational State Transfer</i>
<b>SaaS</b>	<i>Software as a Service</i>
<b>SDK</b>	<i>Software Development Kit</i>
<b>URI</b>	<i>Uniform Resource Identifier</i>
<b>URL</b>	<i>Unified Resource Locator</i>
<b>XML</b>	<i>eXtensible Markup Language</i>



## 1 Introdução

Para a realização deste trabalho final de curso, parte integrante da avaliação para a Licenciatura em Engenharia Informática na Universidade Lusófona, foi-me proposta a possibilidade de o realizar junto da empresa CreateIT. A CreateIT foi fundada em 2001 com um princípio orientador – desenvolver para os seus clientes projetos tecnológicos assentes nas melhores práticas e tecnologias. A empresa fornece serviços e soluções multiplataforma, incluindo serviços de colaboração, *web* e de integração de sistemas. A CreateIT trabalha sobretudo sobre tecnologia *Microsoft*, com destaque para utilização de tecnologias como *Windows Azure*, *Office 365*, *SharePoint*, *Microsoft SQL* e *Microsoft BizTalk*.

Neste contexto, o objetivo principal do meu trabalho é desenvolver uma solução escalável para processamento de pedidos assíncronos com monitorização em tempo real, que permita o seu recebimento, processamento e também monitoria da sua execução. Esta solução deverá permitir que tenhamos um processamento distinto para cada tipo de pedido e que o utilizador possa acompanhar em tempo real o estado dos pedidos. A solução também terá em conta a manutenção do sistema consoante as necessidades do negócio e adaptabilidade da solução a diferentes cenários.

Cada pedido de execução consistirá num pedido REST a uma *Application Programming Interface* (API) pré-definida. Adicionalmente, deverá ser prevista ainda a criação de uma componente de processamento para cada tipo de pedido, através de um sistema de *plugins*, de forma que seja invocado o *plugin* correspondente ao tipo de pedido a processar. Isto permite que, caso sejam necessários processar novos tipos de pedidos já depois da solução estar implementada, apenas teremos de desenvolver um novo *plugin* que irá tratar do *business logic* do novo tipo de pedido, sem que tenhamos de alterar código em qualquer outro componente da solução. Ainda outra vantagem a evidenciar, é o facto de apenas ser invocado o *plugin* correspondente, não será gasto tempo de processamento extra para validações do tipo de pedido.

Estes pedidos deverão ter sempre origem num *front end*, mas podem atravessar várias fases de processamento que posteriormente podem originar novos pedidos. Por exemplo no caso de um pedido que tem vários *stages*, onde o pedido inicial possa ser a obtenção de uma lista de clientes e consequentemente os novos pedidos passam a ser a

obtenção das subscrições dos respetivos clientes. Aqui, a informação dos clientes para obter as respetivas subscrições já não será feita através do *front-end* mas sim do *plugin* que tratou do *stage* do pedido inicial.

Para monitoria do estado da execução de cada processo, pretende-se adicionalmente a disponibilização de um serviço de monitoria que permitirá a visualização do estado de cada pedido. Esta monitoria será feita num *dashboard* elaborado no *front end*.

A execução da aplicação deverá ser testada em dois cenários diferentes. (1) Primeiro, num site de comércio eletrónico, desenvolvido especificamente para este projeto final de curso. Este cenário deverá permitir simular pedidos de produtos cuja monitoria do estado do pedido seja feita através da biblioteca *SignalR*. (2) num projeto real, em desenvolvimento na CreateIT, que requer obter determinadas listas de clientes e respetivas subscrições.

Para testar a execução foi proposto a criação de um site de comércio eletrónico que permita simular pedidos de produtos cuja monitoria do estado do pedido fosse feito na página através da biblioteca *SignalR*. Contudo durante o desenvolvimento do projeto, foi proposto que este se adaptasse a um projeto em desenvolvimento na CreateIT, onde inicialmente o objetivo passa por obter determinadas listas de clientes e respetivas subscrições.

A implementação desta solução será feita utilizando tecnologias Microsoft, mais concretamente desenvolver sob a plataforma .NET e com o intuito que esta fosse implementada na *cloud*, neste caso utilizando o *Azure*, plataforma de serviços criada pela *Microsoft*. Este projeto apresenta ainda como grande aliciante a utilização de ferramentas muito recentes como *Azure Api Apps*, *Azure Service Bus Queues*, *Event Hubs*, *SignalR* e *AngularJS*. O facto de incorporar diversas tecnologias, grande parte delas completa novidade para a minha pessoa, tornou todo este projeto um grande desafio o qual aceitei de bom grado.

Este documento encontra-se organizado do seguinte modo:

Nesta seção apresentam-se os objetivos do projeto, o seu enquadramento, bem como a organização do documento,

Na seção 2, é feito todo o enquadramento teórico, mais concretamente é dado ênfase a todas as tecnologias utilizadas para o desenvolvimento desta solução. Aqui é feita uma breve descrição das tecnologias bem como detalhados pormenores relevantes para a utilização da mesma neste projeto.

Na seção 3 apresenta uma abordagem à arquitetura da solução, explicando de forma superficial a interligação dos diferentes componentes presentes na mesma.

A seção 4 descreve o ambiente de desenvolvimento utilizado ao longo da elaboração do projeto, descrevendo ferramentas utilizadas e o intuito da sua utilização.

Na seção 5 é feita uma abordagem pormenorizada do desenvolvimento dos diferentes componentes e projetos desenvolvidos. Aqui é explicado a um nível mais detalhado o papel e implementação de cada componente presente na solução.

## 2 Enquadramento teórico

Aqui são descritas as tecnologias utilizadas para o desenvolvimento desta solução, de forma a compreender melhor a utilização e importância das mesmas no ambiente que foi desenvolvido.

### 2.1 Azure

Com a necessidade de desenvolvimento rápido e fiável de aplicações, é importante que em certas alturas queiramos deixar de lado questões como o espaço físico para máquinas que tratem do alojamento das nossas soluções, a manutenção desse mesmo espaço físico, questões de rede, sistemas operativos, etc. com fim a dedicarmos o nosso tempo ao desenvolvimento dos nossos aplicativos.

Para tal, o *Microsoft Azure*<sup>1</sup>, plataforma *cloud* pública da *Microsoft*, funcionando como *Infrastructure as a Service* (IaaS)<sup>2</sup>, tanto como *Platform as a Service* (PaaS)<sup>3</sup>, fornece um vasto conjunto de serviços.

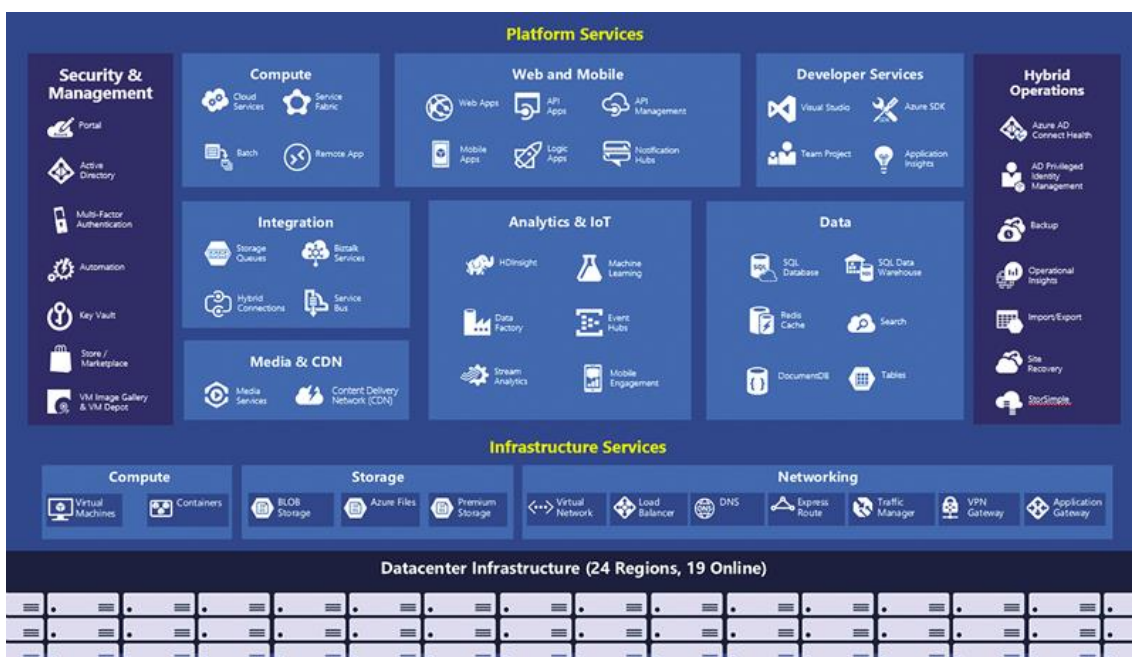


Figura 1 – Serviços fornecidos pelo Microsoft Azure

<sup>1</sup> <http://azure.microsoft.com/>

<sup>2</sup> Infrastructure as a Service (IaaS) ...

<sup>3</sup> Platform as a Service (PaaS) ...

Uma característica de realçar do *Azure* é que pagamos aquilo que usamos, ou seja, vamos pagar apenas pelos recursos que estamos efetivamente a utilizar, ou pela quantidade de banda que estamos a utilizar. Posto isto, é também de salientar que os recursos que estamos a utilizar podem ser escalados caso a nossa solução assim o necessite em determinadas alturas e posteriormente serem reajustados.

De seguida são abordados individualmente os serviços que foram utilizados no desenvolvimento desta solução.

### 2.1.1 Azure Service Bus Queues

Uma *queue* é uma estrutura de dados cujo princípio passa por implementar uma metodologia FIFO (*first in first out*), resumidamente, uma fila.

De forma a entendermos melhor o funcionamento de uma *queue* podemos utilizar um *call center* como analogia, onde várias pessoas ligam para o serviço, e este, mantém-nas em espera de forma ordenada até que um atendedor esteja livre.

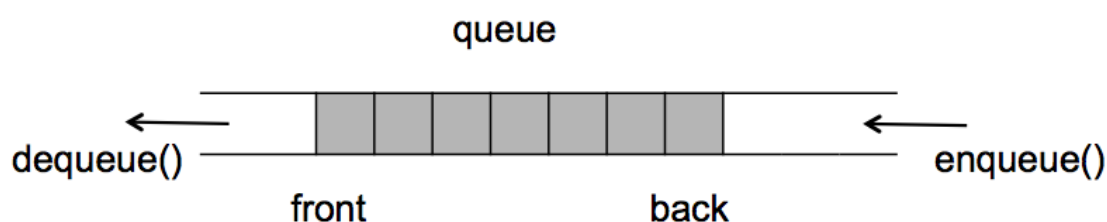


Figura 2 – Ilustração básica de uma *queue*

O *Azure Service Bus*<sup>4</sup> é uma das ferramentas utilizada no desenvolvimento desta solução. Esta, permite integrar várias aplicações na cloud, mais especificamente são utilizadas *queues* no *Azure Service Bus*.

As *queues* no *Service Bus* fornecem uma forma assíncrona de colocar mensagens na fila, isto permite que não seja necessária uma resposta “do outro lado” em como a mensagem foi processada para que continuem a ser enviadas e processadas mensagens. As *queues* são muito úteis em alguns casos, pois são uma forma de aplicações comunicarem mesmo que não estejam a correr em simultâneo, pois a comunicação não é

---

<sup>4</sup> <https://azure.microsoft.com/en-us/documentation/articles/service-bus-fundamentals-hybrid-solutions/>

feita diretamente pelas aplicações, a queue vai então funcionar como intermediário entre aplicações.

Existem duas maneiras de remover uma mensagem da *queue*, através do *RecieveAndDelete*<sup>5</sup>, este remove a mensagem da *queue* e apaga-a, o que pode não ser bom pois pode apagar a mensagem antes de esta ser processada. E a segunda opção é o *PeekLock*<sup>6</sup> que faz *lock* à mensagem, tornando-a invisível para os outros *recievers*. Após isto esperamos por um dos 3 eventos: se a mensagem for processada com sucesso, “*it calls Complete*”<sup>7</sup>, e apaga a mensagem da queue, se o *reciever* não conseguir processar a mensagem, “*it calls Abandon*”<sup>8</sup>, e é feito o *unlock* da mensagem ficando esta disponível para os outros *recievers*. Por último se após 60 segundos (tempo por defeito) não existir uma resposta do *reciever*, é considerado um *Abandon*.

#### 2.1.1.1 *BrokeredMessage*

Para a comunicação de mensagens na *Service Bus Queue*, é utilizada a classe *BrokeredMessage*. Resumidamente cada objeto instanciado é composto por um cabeçalho com diversas propriedades, como *MessageId*, *TimeToLive*, etc. e um corpo da mensagem onde irão estar os dados do pedido.

#### 2.1.2 Azure Event Hubs

Atualmente temos cada vez mais dados a serem gerados por inúmeros dispositivos e surge a necessidade de processarmos toda essa informação, principalmente em tempo real.

O *Azure Event Hubs*<sup>9</sup> vem então responder perante esta necessidade de processar largas quantidades de dados rapidamente e em tempo real. Trata-se de um sistema composto por partições cujo número é configurável entre 2 e 32, cada partição é independente e contém a sua própria sequência de dados, dados estes que expiram com uma base natural não sendo possível apaga-los manualmente, mesmo depois de lidos.

---

<sup>5</sup> <https://msdn.microsoft.com/library/azure/microsoft.servicebus.messaging.receiveivemode.aspx>

<sup>6</sup> <https://msdn.microsoft.com/library/azure/microsoft.servicebus.messaging.receiveivemode.aspx>

<sup>7</sup> <https://msdn.microsoft.com/en-us/library/microsoft.servicebus.messaging.brokeredmessage.complete.aspx>

<sup>8</sup> <https://msdn.microsoft.com/en-us/library/hh181837.aspx>

<sup>9</sup> <https://azure.microsoft.com/en-us/services/event-hubs/>

Os *Event Hubs* funcionam numa arquitetura *publisher/consumer*, sendo possível para quem envia eventos utilizar os protocolos HTTP(S) ou AMQP 1.0<sup>10</sup> o que permite que a maioria dos dispositivos possam enviar dados para o *Event Hub*, o que nos demonstra uma excelente característica que é a possibilidade de integração de diversos tipos de dispositivos.

Relativamente aos *consumers*, para estes podem ser criados grupos, caso o cenário assim o justifique, contudo, cada partição só pode ser lida por um *consumer* de cada vez.

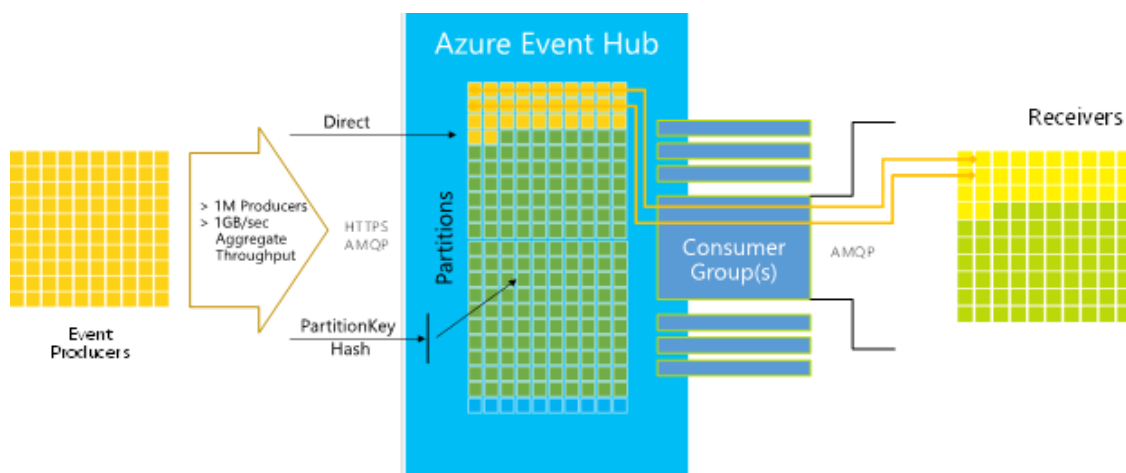


Figura 3 – Arquitetura do Event Hub

#### 2.1.2.1 *EventData*

Os eventos são enviados para o *Event Hub* utilizando a classe *EventData*<sup>11</sup>. Esta classe contém diferentes propriedades como podemos identificar na figura seguinte.



Figura 4 - Propriedades da classe *EventData*

Como podemos observar uma das propriedades é o *Body*, cujo objetivo é levar o conteúdo principal dos nossos eventos., podemos ainda definir determinadas

<sup>10</sup> <https://www.amqp.org/>

<sup>11</sup> <https://msdn.microsoft.com/library/azure/microsoft.servicebus.messaging.eventdata.aspx>

propriedades se necessitarmos e temos elementos cujo fim é mostrar a posição da mensagem na respetiva partição do *Event Hub*.

### 2.1.3 Azure SQL

Existem duas opções de alojamento respetivo ao *SQL Server*, *Azure SQL Database*<sup>12</sup>, uma base de dados SQL nativa na cloud, mais conhecida como base de dados *platform as a service* (PaaS) que é otimizada para o desenvolvimento de aplicações *software as a service* (SaaS), tendo compatibilidade com a maioria das funcionalidades de um *SQL Server*. Esta é implementada em hardware e software adquirido, alojado e mantido pela *Microsoft*, onde podemos desenvolver diretamente sob tal serviço utilizando funcionalidade pré-definidas. Aqui a utilização é feita segundo o padrão de *pay as u go*, com a opção de escalar consoante as necessidades sem interrupções ou alterações.

Outra opção é a instalação do *SQL Server* numa máquina virtual alojada na cloud, mais conhecido como *infrastructure as a service* (IaaS). Semelhante ao *Azure SQL*, é implementado em hardware mantido pela *Microsoft*. Contudo temos mais liberdade na configuração do *SQL Server*, mas também mais tempo despendido na administração do mesmo. Este cenário é indicado por exemplo para a construção de aplicações híbridas.

Em baixo temos ilustradamente as opções de alojamento para o *SQL Server*.

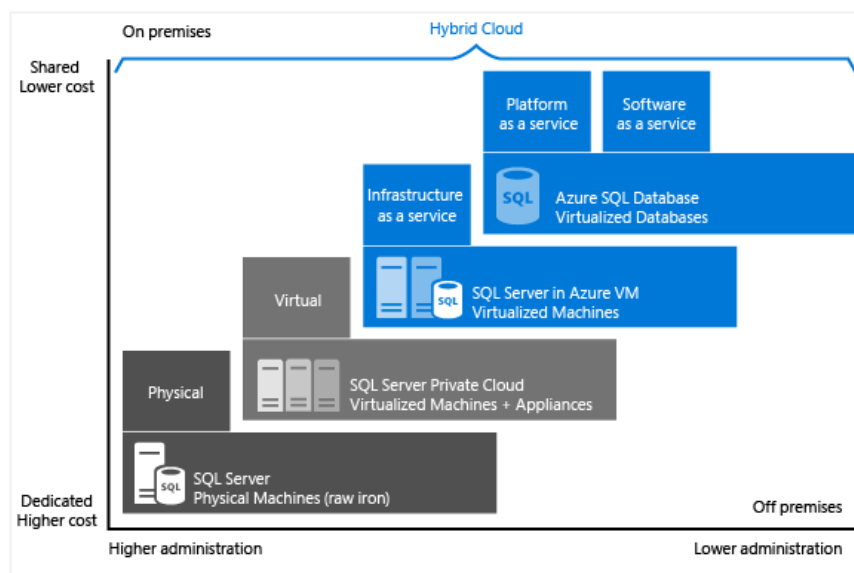


Figura 5 - Modelos de alojamento para SQL Server

<sup>12</sup> <https://azure.microsoft.com/en-us/services/sql-database/>



#### 2.1.4 Azure App Service

O *Azure App Service*<sup>13</sup> permite-nos colocar diversos tipos de aplicações alojados na *cloud*, começando pelo serviço *Web Apps*, que tem o intuito de alojar *websites* e *web applications*, o serviço *Mobile Apps* que nos permite alojar *back ends* de aplicações *mobile*, o serviço *Api Apps* para alojamento de APIs na *cloud* e o serviço de *Logic Apps* para automatizar o acesso e uso de dados e simplificar a integração de componentes.

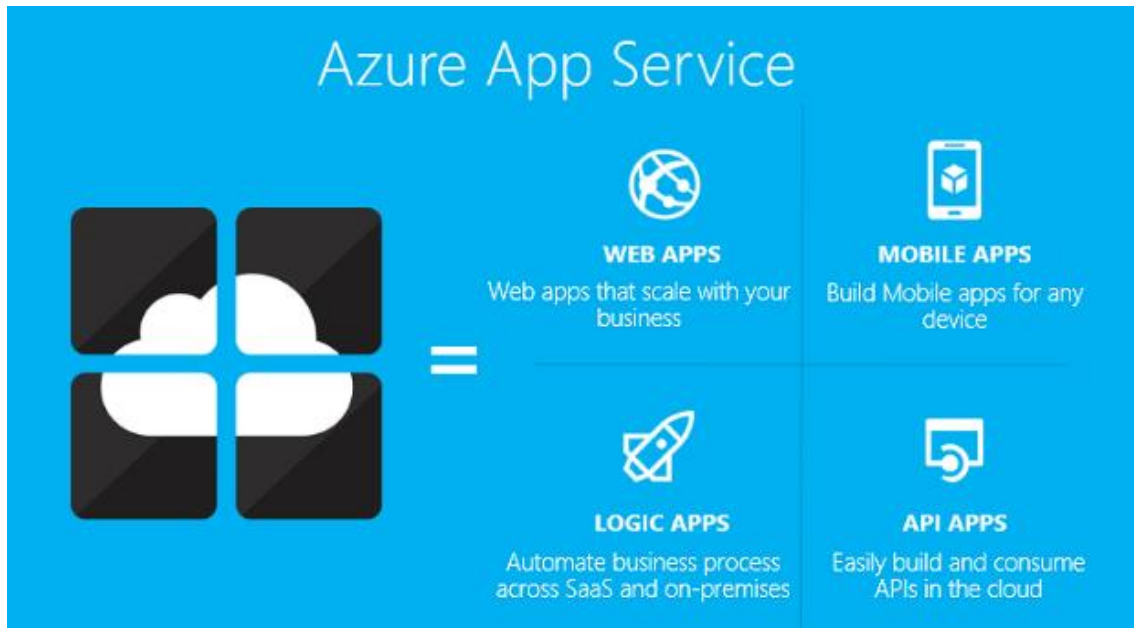


Figura 6 – Os diferentes serviços do Azure App Service

#### 2.1.5 Azure WebJobs

O *Azure WebJob SDK*<sup>14</sup> trata-se de uma *framework* que nos simplifica o desenvolvimento de código de processamento que queiramos desenvolver para a nossa solução, neste caso específico, o sistema de *trigger* que automaticamente invoca uma função sempre que algo é detetado numa *queue*.

Esta é uma solução que conseguimos colocar na *cloud* ao nível de um *website*, que em termos de custos se revela muito mais económica. Isto é, tendo em conta o serviço descrito no ponto anterior, este é alojado mais especificamente como uma *Web App*, cujo serviço está dentro da categoria *Azure App Service*.

<sup>13</sup> <https://azure.microsoft.com/en-us/services/app-service/>

<sup>14</sup> <https://github.com/Azure/azure-webjobs-sdk>

## 2.2 .NET e C#

Devido à natureza e ao contexto do projeto, todo este foi desenvolvido utilizando a plataforma .NET<sup>15</sup>, sendo esta desenvolvida pela *Microsoft*. Aqui impera a utilização da linguagem de programação C#, que se trata de uma linguagem orientada a objetos.

## 2.3 SignalR

O *SignalR*<sup>16</sup> é uma biblioteca para ASP.NET cujo objetivo passa por simplificar a implementação de funcionalidades *web* em tempo real permitindo comunicação bidirecional entre o servidor e o cliente.

Para tal, o *SignalR*, trata da gestão da conexão automaticamente e realiza o transporte via *WebSockets*<sup>17</sup> sempre que disponíveis. Contudo, caso não seja possível estabelecer uma ligação via *WebSocket* (por exemplo por incompatibilidade do *browser*), o *SignalR* vai verificando que tipo de ligação pode ser estabelecida pela seguinte ordem (sendo os dois primeiros dependentes do suporte do HTML5): *WebSocket*, seguido de *Server Sent Events*, *Forever Frame* e por fim *Long Polling*.

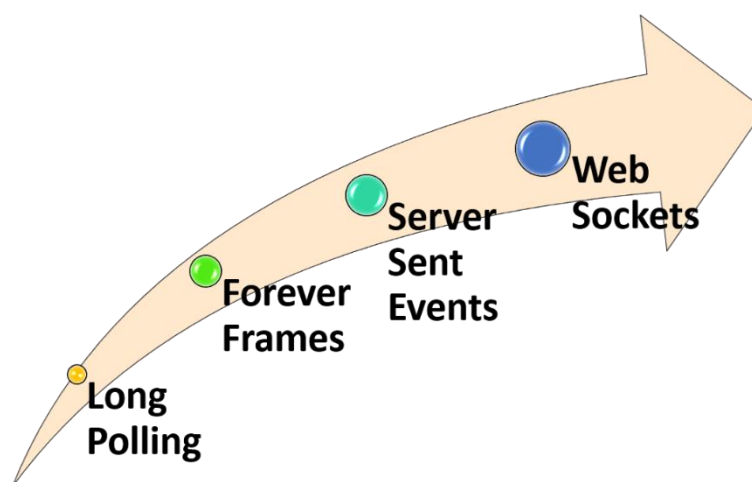


Figura 7 - Fallback das técnicas de comunicação do SignalR

A API do *SignalR* contém dois modelos de comunicação entre clientes e servidores, *Persistent Connections* e *Hubs*. As *persistent connections* é uma API de mais baixo nível, que nos dá acesso ao canal de comunicação semelhante ao que utilizamos quando estamos tradicionalmente a trabalhar com *sockets*<sup>18</sup>. No lado do servidor,

<sup>15</sup> <https://msdn.microsoft.com/en-us/library/z1zx9t92.aspx>

<sup>16</sup> <http://signalr.net/>

<sup>17</sup> <https://www.websocket.org/>

<sup>18</sup> [http://www.tutorialspoint.com/unix\\_sockets/what\\_is\\_socket.htm](http://www.tutorialspoint.com/unix_sockets/what_is_socket.htm)

podemos ser notificados quando são estabelecidas ligações ou quando são recebidos dados, tal como o envio de informação para clientes. Do lado do cliente, é possível iniciar e terminar ligações tal como enviar e receber dados.

A técnica dos *hubs* é uma API de mais alto nível que tem por base as *persistent connections*, estes têm uma implementação mais simplificada, permitindo que no nosso cliente consigamos invocar funções no servidor e vice-versa.

Esta tecnologia é útil para realizarmos a monitoria da atualização dos processos em tempo real na página principal, pois não precisamos de refrescar a página sucessivamente de forma a obter os novos estados dos pedidos.

## 2.4 AngularJS

Para o *front end*, foi escolhido *AngularJS*<sup>19</sup>, esta é uma *framework open source* e mantida pela *Google* cada vez mais utilizada no desenvolvimento de aplicações *web*. Utilizando JavaScript, permite-nos desenvolver aplicações seguindo os padrões MVC<sup>20</sup> (*Model View Controller*) de forma elegante do lado do cliente, esta, tem uma elevada utilização no desenvolvimento de SPA's (*Single Page Applications*)<sup>21</sup>.

## 2.5 MEF (Managed Extensibility Framework)

Com o intuito de serem criados *plugins* ou extensões para processar o tipo de pedidos, utilizou-se o MEF (*Managed Extensibility Framework*)<sup>22</sup> - uma biblioteca que faz parte da *framework* .NET. O MEF permite-nos definir onde é que a nossa aplicação pode ser estendida expondo módulos que podem ser ligados a componentes externos, permitindo que a nossa aplicação encontre componentes que possam ser interligados em execução através da composição de *parts*.

Basicamente existem três *parts* principais no MEF, temos o *[Import]* atributo definido numa propriedade no código que podemos interpretar como uma porta da nossa aplicação onde algo possa lá encaixar. O *[Export]* é o atributo que identifica por exemplo no *plugin* que este pode ser encaixado num determinado *import* da nossa aplicação. Por

---

<sup>19</sup> <https://angularjs.org/>

<sup>20</sup> Model-view-controller é um padrão de arquitetura de software

<sup>21</sup> Single-Page Application é um web site ou aplicação web contida numa única página com o intuito de fornecer uma experiência ao utilizador mais fluída

<sup>22</sup> [https://msdn.microsoft.com/en-us/library/dd460648\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/dd460648(v=vs.110).aspx)

fim a última *part* essencial é o *CompositionContainer* que irá estabelecer a conexão dos *exports* aos *imports*.

Outras duas partes importantes neste processo são os catálogos e o contrato, o primeiro tem a função de “descobrir” as *parts* por exemplo numa diretoria que contenha *.dll* dos *plugins* desenvolvidos. O contrato é simplesmente uma interface comum que a aplicação e os *plugins* entendem de forma a que possam comunicar.

## 2.6 JSON

JSON (*JavaScript Object Notation*) é um formato leve para troca de dados que pode ser escrito num ficheiro de texto, utilizado maioritariamente em alternativa ao XML<sup>23</sup>. Contudo não é necessária a utilização de JavaScript para tirarmos proveito do JSON.

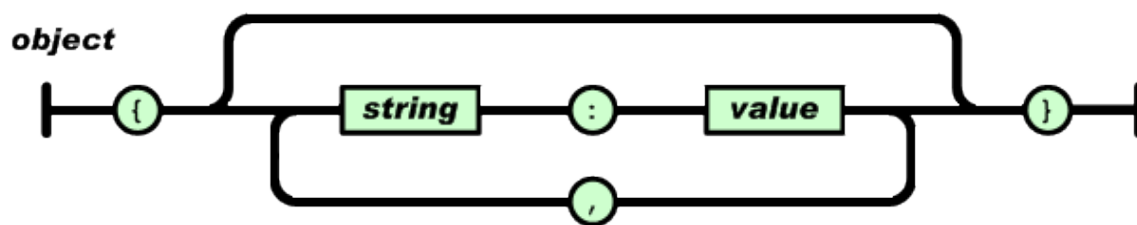


Figura 8 – Ilustração de um objecto em JSON

Podemos ainda separar uma só aplicação em vários serviços diferentes se assim o justificar.

## 2.7 Bootstrap

O *Bootstrap*<sup>24</sup> é uma *framework* de HTML, CSS e JavaScript, ou seja, para o *front end* que tem como objetivo principal o auxílio da construção de *websites* e *web applications* facilitando o desenvolvimento responsivo<sup>25</sup> das mesmas.

<sup>23</sup> <http://www.xml.com/>

<sup>24</sup> <http://getbootstrap.com/>

<sup>25</sup> Responsive Web Design é uma solução técnica para o desenvolvimento de um site de forma a que os elementos que o compõem se adaptem automaticamente às dimensões do ecrã do dispositivo onde este está a ser visualizado

## 2.8 log4net

Registar *logs* numa aplicação sobre as operações praticadas torna-se bastante útil não só para termos um registo do que foi feito, mas também como forma de diagnóstico. O *log4net* é uma biblioteca *open source* destinada ao registo de *logs* para a *framework* *Microsoft .NET*. Esta, permite-nos ter como output diferentes destinos, nomeadamente para ficheiros de texto, bases de dados, etc.

No *log4net* temos 5 níveis de mensagens: *FATAL*, *ERROR*, *WARN*, *INFO*, *DEBUG*, isto permite categorizarmos diferentes tipos de registos de mensagens. Também podemos definir o formato que as mensagens vão ser apresentadas, toda esta configuração, a par com os destinos dos *logs* é feita num ficheiro XML. De seguida observamos uma configuração do formato de uma mensagem e respetivo output.

```
<layout type="log4net.Layout.PatternLayout">  
  <conversionPattern value="%date %level %logger - %message%newline" />  
</layout>
```

```
2015-06-12 20:50:13,256 INFO log4netTutorial.Program - Application works
```

### 3 Arquitetura

Para a criação da solução genérica, foi necessário termos sempre em mente a adaptabilidade da solução a diferentes cenários como um site de e-commerce, uma plataforma que forneça serviços, ou mesmo um sistema que receba pedidos a serem decompostos em vários *stages*. De forma geral o sistema implementado permite-nos realizar um processamento específico para um determinado tipo de pedido, com o auxílio do MEF, ou seja, pegando no exemplo de um site que forneça serviços, podemos efetuar um pedido para usufruirmos de uma base de dados e requerermos alojamento para um website, estes dois pedidos são processados de forma diferente.

Na figura abaixo representada podemos observar o diagrama conceptual da solução. A componente *Azure SQL*, diferenciada das restantes em termos gráficos, foi apenas utilizada para na aplicação da solução no cenário de compras online, não tendo sido usada durante a adaptação da solução ao contexto do projeto real onde esta tem o intuito de processar os pedidos das listagens de clientes e posteriormente respetivas subscrições.

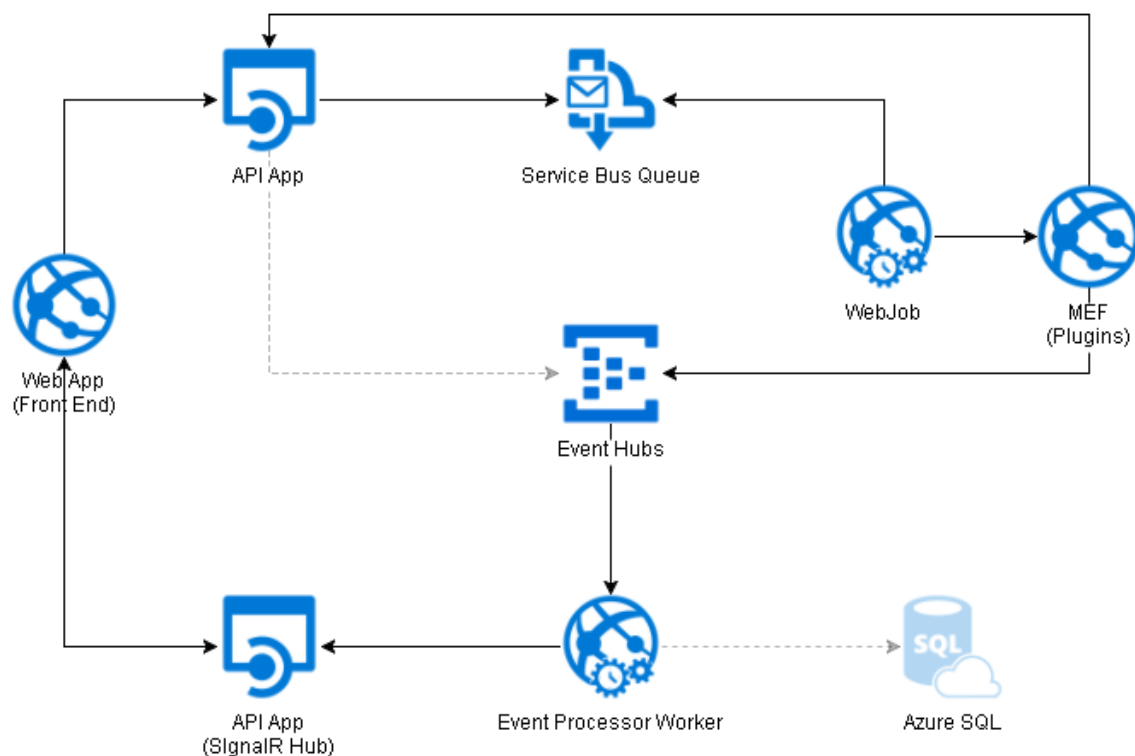


Figura 9 - Diagrama da solução

Cada componente que pode ser observado no diagrama na figura 9, está alojado num determinado serviço apropriado fornecido pelo *Azure*. Todo o desenvolvimento de código necessário foi realizado no *Visual Studio 2015 Community Edition*, criando uma solução que alberga todos os projetos necessários para a implementação dos diferentes componentes que podemos observar no diagrama.

A componente *Web App (front end)* representa o projeto do *web site* desenvolvido em *AngularJS*. Dada a emergente utilização desta *framework* para o desenvolvimento de *Single Page Applications* e a sua famosa organização do código no *front-end*, decidi introduzir e desenvolver conhecimento sobre a mesma, optando então pela sua utilização no *front-end*.

De forma a garantir que o sistema seja genérico e o possamos adaptar a diferentes cenários a *API App* que podemos observar mais no topo da figura 9, foi desenvolvida para receber dois parâmetros, o tipo do pedido e os dados do pedido (por exemplo um objeto em *JSON* no formato *string* que posteriormente é deserializado). Tomei esta opção pois assim os pedidos *REST* que são realizados pelo *front-end*, ou por qualquer outro componente da solução cujo objetivo seja enviar um pedido novo para a *Service Bus Queue*, podem enviar qualquer tipo de pedido. Todavia o processamento do tipo de pedido é feito, ou não, consoante os *plugins* desenvolvidos, isto é, se não existir um *plugin* cuja a *metadata* que o identifica corresponda ao tipo de pedido feito, este não irá ser processado.

Podemos observar pelas setas ligadas ao componente *WebJob*, que este é o componente que está à escuta da *Service Bus Queue* e que quando deteta um pedido, invoca o *plugin* correspondente ao pedido, caso exista. Por sua vez as setas que provêm do componente MEF, são subjetivas a cada tipo de *plugin* desenvolvido, isto é, nem todos os *plugins* têm a necessidade de voltar a colocar um novo pedido na queue. Existe ainda a ligação ao *Event Hub*, esta tem o intuito de registar todos os estados do decorrer do processamento do pedido. É de salientar que os *plugins* que tenham de colocar novos pedidos realizam também eles pedidos *REST* para a API principal, naturalmente esta opção foi tomada de forma a que quem esteja a desenvolver novos *plugins*, não tenha conhecimento ou se preocupe com o estabelecer uma ligação direta à *Service Bus Queue* e assim é também feita uma deteção de pedidos inválidos a colocar na queue por parte da *REST API*.

Idealmente o componente *Event Processor Worker* deveria apenas escrever para o *Azure SQL* e posteriormente a leitura em tempo real ser feita a partir deste. Todavia o *Azure SQL* não usufrui de todas as funcionalidades comparativamente a termos o *SQL Server* instalado localmente ou numa máquina virtual. A funcionalidade do *Service Broker*, que não está disponível *Azure SQL*, é necessária para que seja possível utilizarmos com sucesso a classe *SqlDependency*<sup>26</sup>, cujo funcionamento na implementação desta solução teria por base detetar alterações numa *query* pré-definida, ou seja, estar a “escuta” na base de dados sempre que alguma alteração fosse feita e iniciar um determinado evento, que neste caso seria o envio dos dados para ser feito o *update* nas páginas através do *SignalR*.

Dada tal limitação do serviço *Azure SQL*, tive que repensar a arquitetura relativa ao processamento de eventos, a monitoria em tempo real e o registo persistente de dados. A criação de dois *consumer groups* distintos para os eventos provenientes do *Event Hub*, ou a escrita em paralelo do mesmo evento para o *SignalR Hub* e para o *Azure SQL*, foram opções pensadas, mas que posteriormente causariam outro problema, imaginemos que por alguma razão a escrita para o *Azure SQL* poderia falhar, assim existiria incoerência naquilo que teríamos no armazenamento persistente e no que estaríamos a mostrar ao utilizador, para tal, a escrita é então feita primeiro para o armazenamento persistente e só depois de ter sucesso é então transmitida para o utilizador (*detalhes descritos no respetivo capítulo de desenvolvimento*).

Inicialmente, desenvolvi os componentes do *Event Processor Worker* e o *SignalR Hub* num só, contudo ao longo do desenvolver desta solução, verifiquei que fez mais sentido ter separado estas duas implementações por duas razões principais. Primeiro, o componente que está à escuta dos eventos (*Event Processor Worker*) convém ter o menos processamento possível que não seja a leitura do *Event Hub* de forma a minimizar os erros de leitura e em segundo lugar, se por alguma razão adaptarmos esta solução a um cenário que não necessite de monitoria em tempo real, podemos simplesmente excluir o componente do *SignalR Hub*, e assim será dependido muito menos tempo a adaptar o código desenvolvido a um cenário que não envolva esta monitorização.

---

<sup>26</sup> [https://msdn.microsoft.com/en-us/library/62xk7953\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/62xk7953(v=vs.110).aspx)



## 4 Desenvolvimento

Com o intuito de salvaguardar o código desenvolvido e de manter um controlo de versões, foi utilizada a ferramenta *Team Foundation Server* ao longo do desenvolvimento deste trabalho final de curso. Ferramenta esta que está integrada com o *Visual Studio 2015 Community Edition* que foi o IDE utilizado no decorrer do trabalho.

De forma a contextualizar-me com as diversas tecnologias e conceitos que fizeram parte do desenvolvimento desta solução, foi necessário um enorme estudo de todas elas, não só para tirar máximo proveito das respetivas capacidades e funcionalidades, mas também para a integração das mesmas.

Dada a solução ter sido criada de raiz e envolvendo diversos componentes, inicialmente para efeitos de teste, foram desenvolvidas *console applications* para consolidar e testar os conhecimentos adquiridos, por exemplo para escrita e leitura na *Service Bus Queue*. Durante o desenvolvimento da *REST API*, para efetuar pedidos à mesma, foi feita utilização da ferramenta *Fiddler*<sup>27</sup>, com o objetivo de realizar pedidos do tipo *POST* à API e observar também as respostas.

Ao longo do desenvolvimento desta solução fui desenvolvendo e integrando os diversos componentes presentes na arquitetura, até ao ponto em que tinha a solução funcional desde os pedidos à monitoria em tempo real no *web site*. Para efeitos de teste, os pedidos feitos inicialmente tinham um cenário idêntico ao funcionamento de um *site de e-commerce*, contendo um produto, um preço e um cliente.

Dada a natureza deste projeto passar por criar uma solução genérica, para ser aplicado num contexto mais real, o objetivo passou por realizar a recolha de informação relativa a clientes e respetivas subscrições. Aqui temos um cenário onde o pedido inicial foi decomposto em *stages* sendo inicialmente feito o pedido da lista de clientes para este processamento foi desenvolvido um *plugin* que recebe esta informação através de uma API externa e que assincronamente invoca a API principal da solução para que sejam colocados os clientes na *Service Bus Queue*. À medida que estes novos pedidos são detetados pelo *WebJob*, este invoca o *plugin* desenvolvido para processar o tipo de pedido classificado como “*Customer*”, que por sua vez obtém também através de APIs externas informações relativas às subscrições do cliente que está a ser processado. Toda esta

---

<sup>27</sup> <http://www.telerik.com/fiddler>

informação é monitorizada em tempo real com o intuito de acompanharmos o estado de cada cliente.

Na figura seguinte observados a solução com os vários projetos desenvolvidos no *Visual Studio*.

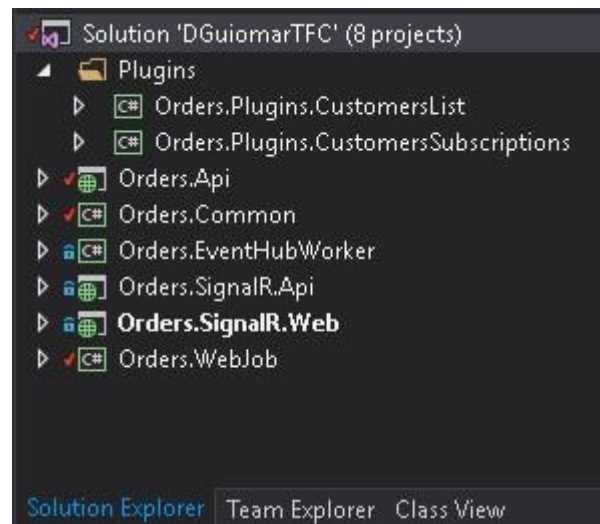


Figura 10 - Vista da solução no Visual Studio

De seguida serão abordados todos os projetos individualmente, evidenciando os detalhes de cada um deles na implementação desta solução, será também clarificada a associação de cada um deles aos componentes presentes na arquitetura.

#### 4.1 Orders.Signalr.Web

Este projeto consiste basicamente na implementação da *front page*. Apesar da complexidade deste ser mínimo do ponto de vista de conteúdo, foi utilizada a *framework AngularJS*, com HTML5 e CSS3 e é de evidenciar a utilização da biblioteca *Bootstrap*, esta teve o intuito de tornar a página responsiva. A utilização do *AngularJS* teve o intuito de me introduzir à mesma, dada a utilização cada vez mais popular e a eficiência que tem demonstrado.

Dado o contexto da criação de um site de comércio eletrónico ter sido abandonado, a página foi então desenvolvida de forma a que fosse simplesmente possível realizar um primeiro pedido REST a uma API pré-definida e que a monitoria de tudo o que daí desencadear seja feita de forma clara e em tempo real num *dashboard* da página. Aqui o

objetivo passa principalmente por demonstrar de forma simples no *dashboard* a monitorização em tempo real do estado dos pedidos.

No ficheiro `route.js` é estabelecido o encaminhamento do *url*, associando determinadas vistas a controladores, ou seja, ajuda-nos a dividir a nossa aplicação de forma lógica e vincular diferentes vistas a controladores. A figura seguinte demonstra como os *urls* estão associados com as vistas e respetivos controladores.

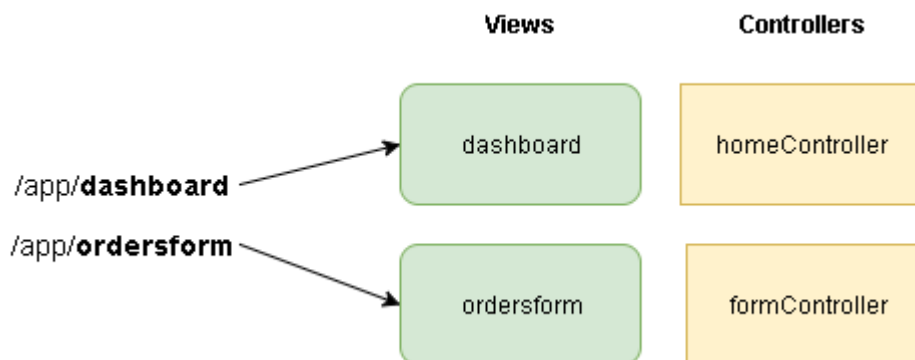


Figura 11 - Routing do projeto em AngularJS

No *homeController* são feitas essencialmente duas tarefas importantes, o recebimento em tempo real dos dados provenientes do servidor de *SignalR* e a função que permite realizar um pedido.

A *framework AngularJS* disponibiliza um serviço `$http`<sup>28</sup> que nos permite realizar pedidos REST, neste caso em concreto, é feito um pedido HTTP POST<sup>29</sup> à *Web API (Orders.Api)*. O encaminhamento do pedido é identificado através de um URI, neste caso ‘{url da api}/api/order’ e leva como corpo, dados que podem ser relevantes ao pedido, dados estes enviados no formato JSON. Neste caso em concreto o pedido leva o tipo de pedido “*GetCustomers*” e o corpo do pedido, neste caso uma mera informação do pedido realizado.

Dado o protocolo HTTP ser um sistema de pedido e resposta, ao fazermos o pedido eventualmente vamos obter uma resposta, neste caso de sucesso ou insucesso, onde também o corpo da resposta é retornado em formato JSON.

<sup>28</sup> [https://docs.angularjs.org/api/ng/service/\\$http](https://docs.angularjs.org/api/ng/service/$http)

<sup>29</sup> É um método de requisição de dados suportado pelo protocolo http, neste caso submete dados para serem processados para um destino específico

Outra tarefa importante a ser referida neste projeto e mais especificamente neste controlador, é a ligação ao servidor do *SignalR*, inicialmente é estabelecida uma ligação com a *Web Api (Orders.SignalR.Api)* que funciona como servidor do *SignalR*. No controlador é ainda implementado um método local que o objetivo de este ser chamado pelo servidor *hub* do *SignalR*. Isto é feito de forma a que a ligação via *WebSocket* estabelecida pelo *SignalR* possa invocar o método do lado do cliente e atualizar o estado dos pedidos.

Para além da utilização da *framework Bootstrap*, foi também criada uma folha de estilos *mystyles.css* para ajustar o *design* da página.

## 4.2 Orders.Api

Este projeto trata-se de uma *Web API* que recebe os pedidos REST e os coloca na *queue*. A *Web API* utiliza os conceitos de controlador e ação do MVC. Os recursos são mapeados diretamente para os controladores, que são os responsáveis por gerir os pedidos HTTP, esta configuração é feita no ficheiro *WebApiConfig.cs*. Neste caso temos apenas um controlador com um método, no ficheiro *OrdersController.cs* foi implementado um método que é identificado pelo atributo *[HttpPost]*. Dado tratar-se de um pedido do tipo POST, este recebe como argumento um JSON que contém os dados do pedido, ou seja, o tipo do pedido e o corpo, este corpo do pedido é uma *string* que pode conter um objeto em serializado em JSON, com dados úteis para o processamento feito no *plugin* respetivo.

Neste método tratamos também os tipos de respostas que devolvemos aos pedidos, por exemplo quando o corpo do pedido é *null* devolvemos uma resposta com o código do protocolo HTTP (neste caso *HttpStatusCode.BadRequest*) e uma descrição “*Invalid request*”.

Inicialmente, no contexto das compras online, este método tinha um comportamento mais complexo do que o atual, por isso é interessante descrever o mesmo. Recebia os dados do pedido, e tinha duas tarefas principais, enviar os dados do pedido para a *Service Bus Queue* e enviar um evento com o estado “Criado” para o *Event Hub*. A ordem de tarefas do método desenvolvido para tal era a seguinte:

- Gera um *CorrelationId*;
- Converte os dados do pedido para uma *string* que é composta por um JSON;

- É instanciado um objeto do tipo *EventData* e enviado para o *Event Hub*, onde o corpo será composto pelo *CorrelationId* e pelo *Status*;

- É instanciado um objeto do tipo *BrokeredMessage* e enviado para a *Service Bus Queue*, onde o corpo são os dados do pedido, juntamente com o *CorrelationId*;

O objetivo deste *CorrelationId* passa por manter uma relação entre o objeto com os detalhes do pedido e o objeto com os estados do mesmo. A serialização do JSON para uma *string* é feita com o auxílio da biblioteca *Newtonsoft.Json*<sup>30</sup>.

Atualmente esta API limita-se a criar uma *BrokeredMessage*, a estabelecer uma ligação à *Service Bus Queue* e enviar o pedido.



Figura 12 - Comunicação da REST Api

O objetivo disto é iniciar o processo para obter a lista de *Customers*. Esta API ficará ainda a funcionar na cloud através do serviço *Azure Api Apps*.

### 4.3 Orders.EventHubWorker

Neste projeto é feita a leitura de todos os eventos que vão chegando ao *Event Hub*, para tal fez-se uso da classe *EventProcessorHost*<sup>31</sup>, implementando a interface *IEventProcessor*. Quando eventos chegam ao *Event Hub*, mas especificamente a uma das partições, o método *ProcessEventsAsync* é chamado para processar essa determinada partição, bloqueando outras chamadas feitas à mesma. Cada chamada fornece-nos uma *collection* de eventos que vai ser iterada de forma a processar os eventos em questão. Aqui é ainda utilizado um sistema de *CheckPoint* de forma a estabelecer até onde foi feita a última leitura, basicamente trata-se de uma localização ou *offset* numa determinada partição do *Event Hub*.

<sup>30</sup> <http://www.newtonsoft.com/json>

<sup>31</sup> <https://msdn.microsoft.com/library/microsoft.servicebus.messaging.eventprocessorhost.aspx>

No contexto das compras online, estes eventos contendo o *CorrelationId* e o *Status* dos pedidos, eram escritos para uma base dados *Azure SQL* para armazenamento persistente. Para isto foi criado um *stored procedure*<sup>32</sup> com o intuito de fazer um “*Insert or Update*”, ou seja, cada vez que um evento é processado, é invocado o *stored procedure*.

De forma a manter coerência do que temos armazenado persistentemente com a monitorização em tempo real que será mostrada ao utilizador, temos de nos certificar que a escrita é feita com sucesso para a base de dados e então depois disponibilizada para a monitorização em tempo real, ou seja, para o servidor *hub* do *SignalR*. Através do valor de retorno do método *ExecuteNonQuery*, que nos indica o número de linhas afetadas na base de dados pelo *stored procedure*, foi feito um pequeno teste condicional e se tiver sido bem sucedido, a informação é enviada para o servidor do *SignalR*, que se trata do projeto *Orders.SignalR.Api*. Para tal, foi implementado um método cujo objetivo passa por realizar um pedido HTTP POST onde o corpo da mensagem é a informação retirada do *Event Hub*.

```
private static async Task<HttpResponseMessage> PostNotification(EventData
eventData)
{
    var httpClient = new HttpClient();
    var content = new
StringContent(Encoding.UTF8.GetString(eventData.GetBytes()), Encoding.UTF8,
"application/json");
    return await httpClient.PostAsync(new
Uri("http://tfccordersapisignalrhuh.azurewebsites.net/api/eventnotification"),
content);
}
```

No contexto atual da informação dos clientes, a escrita na base de dados não era necessária de momento, tendo sido descartada. Posto isto, à medida que os eventos são obtidos vão sendo enviados para o servidor do *SignalR*.

## 4.4 Orders.SignalR.Api

Este projeto tem a finalidade de servir como API contendo o servidor do *SignalR*, neste caso um *hub*, ou seja, temos um controlador onde foi implementado um método com o atributo *[HttpPost]* e uma classe que implementa o *hub* do *SignalR*. Para definir o

---

<sup>32</sup> Stored procedure

caminho que os clientes utilizam para se conectar ao *hub*, na classe *Startup* invocamos o método *MapSignalR* quando a aplicação inicia.

```
app.MapSignalR(new HubConfiguration { EnableJSONP = true });
```

Sempre que um evento é recebido vindo do processador de eventos, o *controller* da *Web API* faz uma chamada à função *SendData* da classe *OrdersHub* passando por argumento os dados a serem disponibilizados na página. A função *SendData* por sua vez invoca um método implementado nos clientes enviando os dados com a informação nova para ser atualizada na página.

Por outras palavras esta API servirá como servidor do *SignalR*, o facto de este funcionar como uma *Web API* permite que posteriormente, consoante o cenário, vários tipos de clientes (não apenas esta página de *front end*) possam receber as atualizações em tempo real.

De seguida podemos observar a interligação do processador de eventos, com o servidor do *SignalR* e o envio da informação para os clientes.

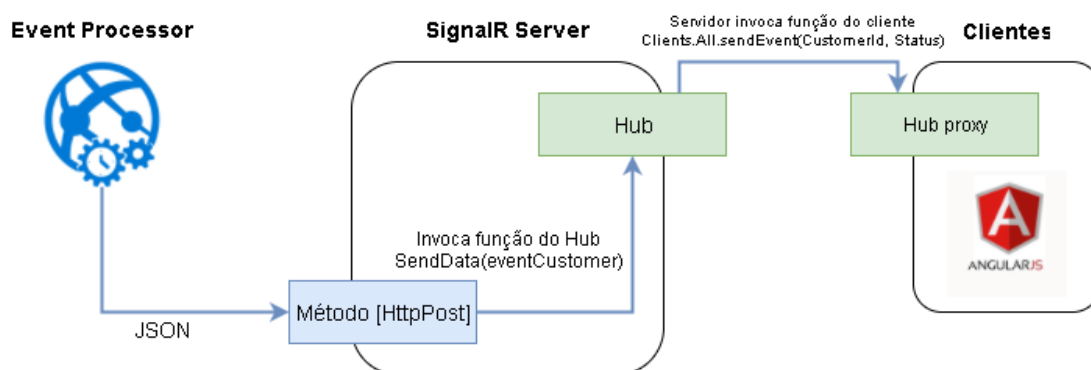


Figura 13 - Funcionamento da API do servidor do SignalR

## 4.5 Orders.WebJob

As duas tarefas principais desenvolvidas neste projeto são ler os pedidos que estão na *Service Bus Queue* e invocar o *plugin* indicado consoante o tipo de pedido.

Através do nome da *queue* e da *connection string*<sup>33</sup> definidos do ficheiro *App.config*, identificamos a *queue* que estaremos à escuta. Posto isto, sempre que existe algo a ler, é feito o *trigger* e é invocada a função que vai realizar o processamento dos

<sup>33</sup> String que especifica a informação sobre a fonte de dados e os meios para efetuar a ligação

dados recebidos, neste caso é uma *BrokeredMessage* que contém a informação do pedido no seu corpo. Relativamente aos vários campos de cabeçalho, destacam-se o *CorrelationId* e o tipo de pedido.

De forma a ser chamado o *plugin* correto para o processamento da mensagem recebida, é comparada a *property Type* do cabeçalho da mensagem e a *metadata* dos *plugins*, caso algum coincida, é invocado o método da interface *IMessageContract* que está implementada nos *plugins*.

O excerto de código demonstra a implementação do que foi acima descrito.

```
public async Task ProcessQueue([ServiceBusTrigger("tfccorderqueue")] BrokeredMessage
message, TextWriter logger)
{
    try
    {
        logger.WriteLine($"#### NEW MESSAGE - Type: {message.Properties["Type"]}");
        logger.WriteLine($"{plugins.messageTypes.Count()} plugin(s) loaded..");

        // search on available plugins for a match
        if (plugins.messageTypes.Count() > 0 && message.Properties != null)
        {
            foreach (var msgType in plugins.messageTypes)
            {
                if (msgType.Metadata.Message.Equals(message.Properties["Type"]))
                {
                    logger.WriteLine("Plugin Match Found, calling function from
plugin...");
                    await Task.Factory.StartNew(() =>
msgType.Value.GetMessageType(message.GetBody<string>()));
                }
            }
        }
        catch (Exception exception)
        {
            logger.WriteLine("Failed to process: " + message.CorrelationId + " - " +
exception.ToString());
        }
    }
}
```

## 4.6 Orders.Common

Este é o projeto que contém classes e interfaces comuns a toda a solução, de forma geral, este projeto está referenciado em todos os outros.

Mais concretamente, este projeto alberga classes para o antigo contexto como *Order.cs* (detalhes da compra), *OrderEvent.cs* (detalhes do estado da compra), e para o contexto atual tendo a classe *CustomerEvent.cs*, tendo como propriedades o *CustomerId* e o *Status* do respetivo pedido. Foi também criado um enumerado para os diferentes estados da informação do cliente que estamos a obter, ou seja, cada pedido irá passar por estes estados até estar concluído.



```
public enum Status
{
    GetCustomerDomains,
    GetCustomerSubscribedSkus,
    GetCustomerSubscriptions,
    GetSubscriptionsOffers,
    Finished
}
```

Ainda dentro deste projeto temos a interface que é implementada nos projetos referentes aos *plugins*. E por fim a classe que faz a composição desses mesmo *plugins* denominada por *MEFComposer*.

```
[ImportMany(typeof(IMessageContract))]
public IEnumerable<Lazy<IMessageContract, IMessageContractMetadata>>
messageTypes;
```

Dada a funcionalidade do MEF passar por estabelecer ligações entre *parts* (*imports/exports*) que implementam um determinado contrato, ou seja, uma interface comum. Em cima podemos verificar o atributo *ImportMany* que nos permite que vários *exports* possam ser “compatíveis” com aquele mesmo *import*. Neste caso aquela *collection* irá conter todos os *exports* que cumpram a implementação daquela interface.

Contudo para que sejam “descobertos” esses *exports* foi criado um catálogo, neste caso com o caminho para uma pasta que contém os ficheiros *.dll* dos *plugins* desenvolvidos. Este catálogo é utilizado na criação do *CompositionContainer* que fará tal como o nome indica a composição “*behind the scenes*” dos *plugins* assim que estes forem carregados.

```
public MEFComposer()
{
    // create catalog
    // adds all parts from Plugins directory
    var catalog = new AggregateCatalog();
    catalog.Catalogs.Add(new DirectoryCatalog(@"..\..\..\Plugins\"));

    // creates container with all parts in catalog
    container = new CompositionContainer(catalog);

    LoadPlugins();
}
```

## 4.7 Orders.Plugins.CustomersList

Este é um dos projetos que faz parte dos *plugins* referidos anteriormente. Como podemos verificar em baixo temos o atributo *export* e a interface *IMessageContract* implementada. Temos também ainda meta dados que servem para identificar o *plugin* perante os tipos de pedidos.

```
[Export(typeof(IMessageContract))]  
[ExportMetadata("Message", "GetCustomers")]  
public class CustomersList : IMessageContract
```

Aqui são feitos pedidos a uma API externa para obtermos uma lista de *customers*, aqui é também criada uma conexão à *Service Bus Queue* pois à medida que estamos a receber os dados dos clientes da API externa, estes estão a ser serializados seguindo o padrão utilizado ao longo do desenvolvimento da solução para ser enviada uma *BrokeredMessage*, com informação de cabeçalho identificando a mensagem como sendo do tipo “*Customer*” para a *queue* em questão. Em baixo podemos observar um excerto de código do projeto, onde é enviada a informação dos clientes para a *Service Bus Queue*.

```
foreach (var customer in batchOfCustomers.Items)  
{  
    // create BrokeredMessage with customer as body  
    // add header Type = Customer and send to Queue  
    var serializedObj = JsonConvert.SerializeObject(customer);  
    var message = new BrokeredMessage(serializedObj);  
    message.Properties["Type"] = MessageType.Customer.ToString();  
  
    queueClient.SendAsync(message);  
}
```

## 4.8 Orders.Plugins.CustomersSubscriptions

Na sequência dos envios realizados pelo *plugin* anterior, o *WebJob* que está a ler as mensagens da *queue* chama este *plugin* sempre que detetada uma mensagem que indica ser do tipo *Customer*.

O processamento de cada *customer* envolve a passagem por vários estados até estar concluído, estados estes demonstrados no enumerado criado no projeto *Orders.Common*. Neste projeto são feitas chamadas a APIs externas de forma a obter informações relativas a subscrições do cliente a ser processado. À medida que cada passo

é concluído, é enviado um evento para o *Event Hub*, de forma a atualizar o *Status* do processo visto na página *web*.

Em baixo podemos observar um excerto do código que demonstra o envio dos estados para o *Event Hub*.

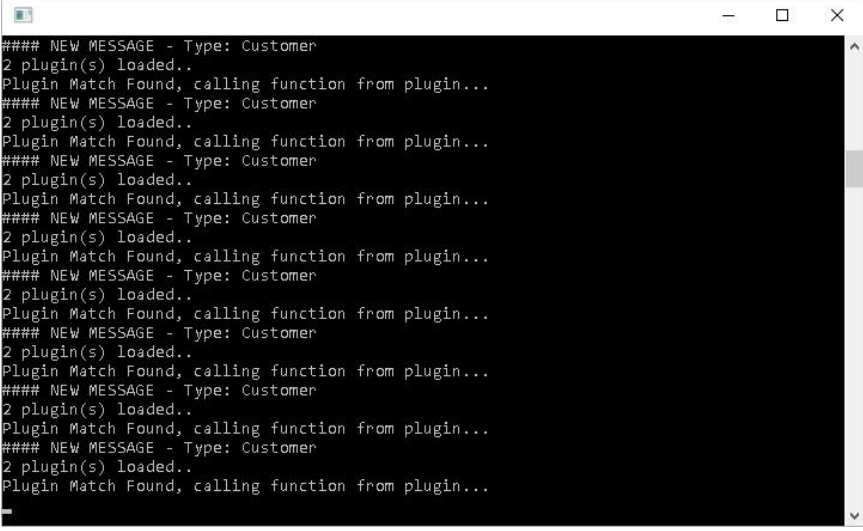
```
_logger.Debug("GetCustomerDomains");
eventCustomer.Status = Status.GetCustomerDomains.ToString();
await eventHubClient.SendAsync(new
EventData(Encoding.UTF8.GetBytes(JsonConvert.SerializeObject(eventCustomer))));
await domainsClient.GetCustomerDomains(Guid.Parse(customer.Id));

_logger.Debug("GetCustomerSubscribedSkus");
eventCustomer.Status = Status.GetCustomerSubscribedSkus.ToString();
await eventHubClient.SendAsync(new
EventData(Encoding.UTF8.GetBytes(JsonConvert.SerializeObject(eventCustomer))));
List<APIEntities.SubscribedSku> customerSubscribedSkus = await
customersClient.GetCustomerSubscribedSkusAsync(Guid.Parse(customer.Id));
```

## 5 Resultados

Neste capítulo observamos resultados provenientes da solução em pleno funcionamento, não só apresentação ao utilizador, mas também a monitoria de alguns componentes presentes no *back end* da solução.

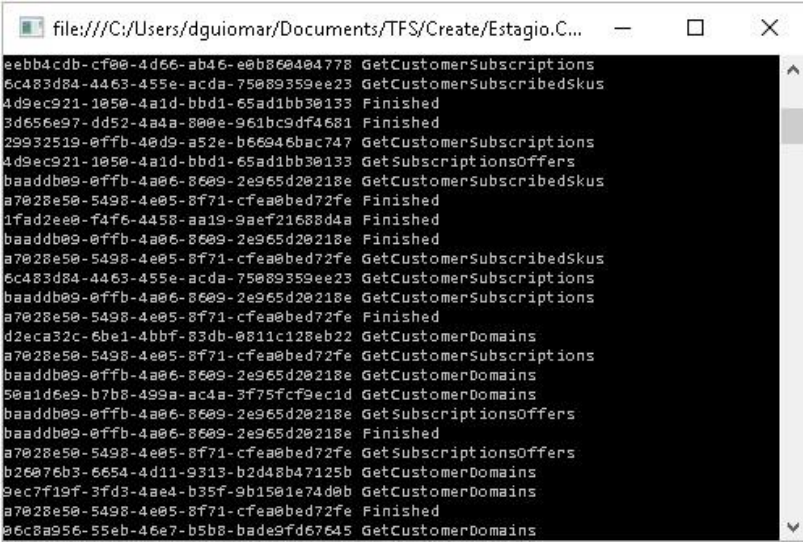
Na figura seguinte podemos observar as mensagens que estão a ser lidas na queue por parte do *WebJob*.



```
#### NEW MESSAGE - Type: Customer
2 plugin(s) loaded..
Plugin Match Found, calling function from plugin...
#### NEW MESSAGE - Type: Customer
2 plugin(s) loaded..
Plugin Match Found, calling function from plugin...
#### NEW MESSAGE - Type: Customer
2 plugin(s) loaded..
Plugin Match Found, calling function from plugin...
#### NEW MESSAGE - Type: Customer
2 plugin(s) loaded..
Plugin Match Found, calling function from plugin...
#### NEW MESSAGE - Type: Customer
2 plugin(s) loaded..
Plugin Match Found, calling function from plugin...
#### NEW MESSAGE - Type: Customer
2 plugin(s) loaded..
Plugin Match Found, calling function from plugin...
#### NEW MESSAGE - Type: Customer
2 plugin(s) loaded..
Plugin Match Found, calling function from plugin...
#### NEW MESSAGE - Type: Customer
2 plugin(s) loaded..
Plugin Match Found, calling function from plugin...
#### NEW MESSAGE - Type: Customer
2 plugin(s) loaded..
Plugin Match Found, calling function from plugin...
```

Figura 14- Processamento dos pedidos provenientes da Service Bus Queue

Durante a execução do processador de eventos podemos observar todos os eventos que estão a ser lidos diretamente do *Event Hub*.



```
eebb4cdb-cf0e-4d66-ab46-e0b860404778 GetCustomerSubscriptions
6c483d84-4463-455e-acda-75089359ee23 GetCustomerSubscribedSkus
4d9ec921-1050-4a1d-bbd1-65ad1bb30133 Finished
3d656e97-dd52-4a4a-800e-961bc9df4681 Finished
29932519-0ffb-40d9-a52e-b66046bac747 GetCustomerSubscriptions
4d9ec921-1050-4a1d-bbd1-65ad1bb30133 GetSubscriptionsOffers
baaddb09-0ffb-4a06-8609-2e965d20218e GetCustomerSubscribedSkus
a7028e50-5498-4e05-8f71-cfeabed72fe Finished
1fad2e00-f4f6-4458-aa19-9aef21688d4a Finished
baaddb09-0ffb-4a06-8609-2e965d20218e Finished
a7028e50-5498-4e05-8f71-cfeabed72fe GetCustomerSubscribedSkus
6c483d84-4463-455e-acda-75089359ee23 GetCustomerSubscriptions
baaddb09-0ffb-4a06-8609-2e965d20218e GetCustomerSubscriptions
a7028e50-5498-4e05-8f71-cfeabed72fe Finished
d2eca32c-6be1-4bbf-83db-0811c128eb22 GetCustomerDomains
a7028e50-5498-4e05-8f71-cfeabed72fe GetCustomerSubscriptions
baaddb09-0ffb-4a06-8609-2e965d20218e GetCustomerDomains
50a1d6e9-b7b8-499a-ac4a-3f75fcf9ec1d GetCustomerDomains
baaddb09-0ffb-4a06-8609-2e965d20218e GetSubscriptionsOffers
baaddb09-0ffb-4a06-8609-2e965d20218e Finished
a7028e50-5498-4e05-8f71-cfeabed72fe GetSubscriptionsOffers
b26076b3-6654-4d11-9313-b2d48b47125b GetCustomerDomains
9ec7f19f-3fd3-4ae4-b35f-9b1501e74d0b GetCustomerDomains
a7028e50-5498-4e05-8f71-cfeabed72fe Finished
06c8a956-55eb-46e7-b5b8-bade9fd67645 GetCustomerDomains
```

Figura 15- Eventos lidos diretamente do Event Hub

Apesar de nesta fase ainda não terem sido feitos intensos testes ao sistema ou comparações com outro tipo de processamento, pode ser já visto a partir do *front end* desenvolvido em *AngularJS* um *dashboard* e uma tabela com os estados dos vários pedidos, sendo estes alterados em tempo real.

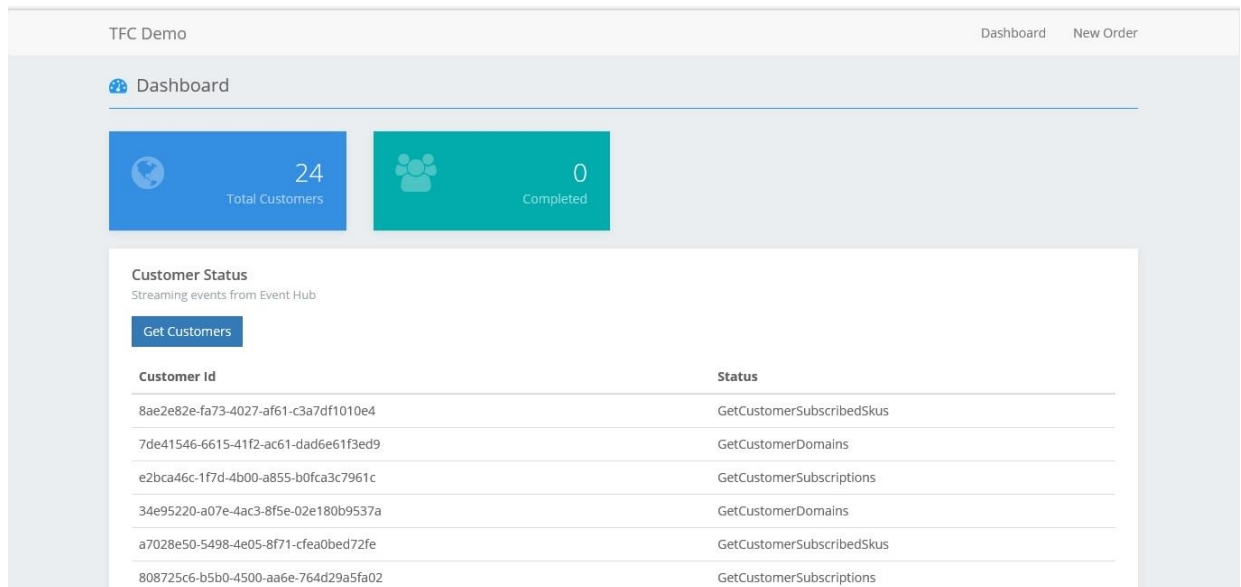


Figura 16- Apresentação ao utilizador do estado dos pedidos

## 6 Conclusões e Trabalho Futuro

Dada a envergadura e a adaptabilidade desta solução é difícil determinarmos a conclusão da mesma, especialmente dada a mudança de contexto durante o seu desenvolvimento. O facto de tal solução incorporar um vasto leque de tecnologias, implica em termos de tempo um grande investimento, não só para conhecimento, mas também para a integração das mesmas. Contudo, o retorno é sem dúvida uma mais valia, pois foram adquiridos diversos conhecimentos em tecnologias muito recentes.

De acordo com os objetivos iniciais do trabalho, apesar de não ter sido implementado o site de *e-commerce*, a solução desenvolvida responde às necessidades propostas. Foi desenvolvida uma API principal que recebe pedidos de vários clientes e que os coloca na *Service Bus Queue*. Foi também desenvolvido com sucesso o sistema de *plugins* com o auxílio da *framework* MEF, não para responder ao contexto de *e-commerce*, mas de forma a processar os *customers* e respetivas subscrições no contexto adotado a certa altura do desenvolvimento desta solução. Foi ainda implementado com sucesso a monitoria dos estados de todos os processos em tempo real, esta monitoria pode ser visualizada na página desenvolvida em *AngularJS*.

Em suma, a elaboração deste trabalho final de curso, permitiu-me elucidar-me pormenores de elevada importância no desenvolvimento de uma aplicação com diversos componentes como esta, que não tinham sido experienciados em projetos desenvolvidos previamente em contexto académico.

Apesar de ter desenvolvido aquilo que me foi proposto, mesmo com a enorme condicionante de tempo dado o meu estatuto, o entusiasmo foi crescendo durante o decorrer deste trabalho, e sinto que muito mais poderia ter sido feito para complementar ou simplesmente tirar partido desta solução.

Mais concretamente, com a utilização cada vez mais comum de *Machine Learning* em aplicações e soluções mais correntes, esta é uma área que me fascina e do meu ponto de vista, desenvolver um módulo de *Machine Learning* poderia tirar grande partido numa solução como esta, especialmente com o “apoio” dos eventos que são gerados no decorrer de todos os processamentos.

Em termos de trabalho futuro que será realizado com o apoio desta solução, este estará nas mãos da empresa CreateIT dar continuidade a adaptação aos seus cenários.

## 7 Bibliografia

- Alicea, A. (s.d.). *Learn and Understand AngularJS*. Obtido de UdeMy: <https://www.udemy.com/learn-angularjs/learn/v4/overview>
- Cleary, S. (Outubro de 2014). *Async Programming : Introduction to Async/Await on ASP.NET*. Obtido de MSDN: <https://msdn.microsoft.com/en-us/magazine/dn802603.aspx>
- Corey, T. (4 de Setembro de 2015). *log4net Tutorial*. Obtido de Code Project: <http://www.codeproject.com/Articles/140911/log-net-Tutorial>
- Dykstra, T. (5 de Maio de 2016). *Get started with API Apps, ASP.NET, and Swagger in Azure App Service*. Obtido de Microsoft Azure: <https://azure.microsoft.com/en-us/documentation/articles/app-service-api-dotnet-get-started/>
- Fischer, L. (9 de Janeiro de 2013). *A Beginner's Guide to HTTP and REST*. Obtido de envatotuts+: <http://code.tutsplus.com/tutorials/a-beginners-guide-to-http-and-rest--net-16340>
- Fletcher, P. (10 de Junho de 2014). *Introduction to SignalR*. Obtido de ASP.NET: <http://www.asp.net/signalr/overview/getting-started/introduction-to-signalr>
- Manheim, S. (4 de Abril de 2016). *Event Hubs programming guide*. Obtido de Microsoft Azure: <https://azure.microsoft.com/en-us/documentation/articles/event-hubs-programming-guide/>
- Microsoft. (s.d.). *Managed Extensibility Framework (MEF)*. Obtido de MSDN: [https://msdn.microsoft.com/en-us/library/dd460648\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/dd460648(v=vs.110).aspx)
- Microsoft. (s.d.). *Programming with C#*. Obtido de edX.org: <https://www.edx.org/course/programming-c-microsoft-dev204x-2>
- SQL Connection*. (s.d.). Obtido de dotnetperls: <http://www.dotnetperls.com/sqlconnection>
- Taubensee, J. (6 de Julho de 2016). *Get started with Service Bus Queues*. Obtido de Microsoft Azure: <https://azure.microsoft.com/en-us/documentation/articles/service-bus-dotnet-get-started-with-queues/>

Young, J. (3 de Fevereiro de 2015). *Sending Real-Time Sensor Data to Clients Using SignalR*. Obtido de ytechie: <http://www.ytechie.com/2015/02/sending-real-time-sensor-data-to-clients-using-signalr/>



## 8 Anexos

### 8.1 Anexo A – Stored Procedure InsertEvents

```
1 CREATE PROCEDURE InsertEvents
2 |
3 @CorrelationID VARCHAR(50),
4 @Status VARCHAR(50)
5
6 AS
7 BEGIN
8     SET NOCOUNT OFF;
9
10    IF EXISTS (SELECT 1 FROM OrderStatus WHERE CorrelationID = @CorrelationID)
11        UPDATE OrderStatus
12            SET Status = @Status
13    ELSE
14        INSERT INTO OrderStatus ([CorrelationID], [Status])
15            VALUES (@CorrelationID, @Status)
16    END
```

Figura 17 - Stored Procedure utilizado para Insert or Update dos eventos na base de dados